

Xilinx Device Drivers

[Driver Summary](#) [Copyright](#)

[Main Page](#) [Data Structures](#) [File List](#) [Data Fields](#) [Globals](#)

Xilinx Device Drivers Documentation

Generated on 29 May 2003 for Xilinx Device Drivers

Device Driver Summary

A summary of each device driver is provided together with links to its layer 1, high level header file and its layer 0, low level header file. A description of the device driver layers is provided in the Device Driver Programmer Guide. In addition, [building block](#) components are described below.

ATM Controller

The Asynchronous Transfer Mode (ATM) Controller driver resides in the *atmc* subdirectory. Details of the layer 1 high level driver can be found in the [xatmc.h](#) header file. Details of the layer 0 low level driver can be found in the [xatmc_1.h](#) header file.

Ethernet 10/100 MAC

The Ethernet 10/100 MAC driver resides in the *emac* subdirectory. Details of the layer 1 high level driver can be found in the [xemac.h](#) header file. Details of the layer 0 low level driver can be found in the [xemac_1.h](#) header file.

Ethernet 10/100 MAC Lite

The Ethernet 10/100 MAC Lite driver resides in the *emac_lite* subdirectory. Details of the layer 0 low level driver can be found in the [xemaclite_1.h](#) header file.

External Memory Controller

The External Memory Controller driver resides in the *emc* subdirectory. Details of the layer 1 high level driver can be found in the [xemc.h](#) header file. Details of the layer 0 low level driver can be found in the [xemc_1.h](#) header file.

General Purpose I/O

The General Purpose I/O driver resides in the *gpio* subdirectory. Details of the layer 1 high level driver can be found in the [xgpio.h](#) header file. Details of the layer 0 low level driver can be found in the [xgpio_1.h](#) header file.

Gigabit Ethernet MAC

The 1 Gigabit Ethernet MAC driver resides in the *gemac* subdirectory. Details of the layer 1 high level driver can be found in the [xgemac.h](#) header file. Details of the layer 0 low level driver can be found in the [xgemac_1.h](#) header file.

HDLC

The HDLC driver resides in the *hdlc* subdirectory. Details of the layer 1 high level driver can be found in the [xhdlc.h](#) header file. Details of the layer 0 low level driver can be found in the [xhdlc_1.h](#) header file.

Intel StrataFlash

The Intel StrataFlash driver resides in the *flash* subdirectory. Details of the layer 1 high level driver can be found in the [xflash.h](#) header file. Details of the layer 0 low level driver can be found in the [xflash_intel_1.h](#) header file.

Inter-Integrated Circuit (IIC)

The IIC driver resides in the *iic* subdirectory. Details of the layer 1 high level driver can be found in the [xiic.h](#) header file. Details of the layer 0 low level driver can be found in the [xiic_1.h](#) header file.

Interrupt Controller

The Interrupt Controller driver resides in the *intc* subdirectory. Details of the layer 1 high level driver can be found in the [xintc.h](#) header file. Details of the layer 0 low level driver can be found in the [xintc_1.h](#) header file.

OPB Arbiter

The OPB Arbiter driver resides in the *opbarb* subdirectory. Details of the layer 1 high level driver can be found in the [xopbarb.h](#) header file. Details of the layer 0 low level driver can be found in the [xopbarb_1.h](#) header file.

OPB to PLB Bridge

The OPB to PLB bridge driver resides in the *opb2plb* subdirectory. Details of the layer 1 high level driver can be found in the [xopb2plb.h](#) header file. Details of the layer 0 low level driver can

be found in the [xopb2plb_1.h](#) header file.

PCI Bridge

The PCI bridge driver resides in the *pci* subdirectory. Details of the layer 1 high level driver can be found in the [xpci.h](#) header file. Details of the layer 0 low level driver can be found in the [xpci_1.h](#) header file.

PLB Arbiter

The PLB arbiter driver resides in the *plbarb* subdirectory. Details of the layer 1 high level driver can be found in the [xplbarb.h](#) header file. Details of the layer 0 low level driver can be found in the [xplbarb_1.h](#) header file.

PLB to OPB Bridge

The PLB to OPB bridge driver resides in the *plb2opb* subdirectory. Details of the layer 1 high level driver can be found in the [xplb2opb.h](#) header file. Details of the layer 0 low level driver can be found in the [xplb2opb_1.h](#) header file.

Rapid I/O

The Rapid I/O driver resides in the *rapidio* subdirectory. Details of the layer 0 low level driver can be found in the [xrapidio_1.h](#) header file.

Serial Peripheral Interface (SPI)

The SPI driver resides in the *spi* subdirectory. Details of the layer 1 high level driver can be found in the [xspi.h](#) header file. Details of the layer 0 low level driver can be found in the [xspi_1.h](#) header file.

System ACE

The System ACE driver resides in the *sysace* subdirectory. Details of the layer 1 high level driver can be found in the [xsysace.h](#) header file. Details of the layer 0 low level driver can be found in the [xsysace_1.h](#) header file.

Timer/Counter

The Timer/Counter driver resides in the *tmrctr* subdirectory. Details of the layer 1 high level driver can be found in the [xtmrctr.h](#) header file. Details of the layer 0 low level driver can be found in the [xtmrctr_1.h](#) header file.

UART Lite

The UART Lite driver resides in the *uartlite* subdirectory. Details of the layer 1 high level driver can be found in the [xuartlite.h](#) header file. Details of the layer 0 low level driver can be found in the [xuartlite_1.h](#) header file.

UART 16450/16550

The UART 16450/16550 driver resides in the *uartns550* subdirectory. Details of the layer 1 high level driver can be found in the [xuartns550.h](#) header file. Details of the layer 0 low level driver can be found in the [xuartns550_1.h](#) header file.

Watchdog Timer/Timebase

The Watchdog Timer/Timebase driver resides in the *wdttb* subdirectory. Details of the layer 1 high level driver can be found in the [xwdttb.h](#) header file. Details of the layer 0 low level driver can be found in the [xwdttb_1.h](#) header file.

Building Block Components

Common

Common components reside in the *common* subdirectory and comprise a collection of header files and ".c" files that are commonly used by all device drivers and application code. Included in this collection are: [xstatus.h](#) , which contains the identifiers for Xilinx status codes; [xparameters.h](#) , which contains the identifiers for the driver configurations and memory map; and [xbasic_types.h](#) , which contains identifiers for primitive data types and commonly used constants.

CPU/CPU_PPC405

CPU components reside in the *cpu[_ppc405]* subdirectory and comprise I/O functions specific to a processor. These I/O functions are defined in *xio.h*. These functions are used by drivers and are not intended for external use.

IPIF

IPIF components reside in the *ipif* subdirectory and comprise functions related to the IP Interface (IPIF) interrupt control logic. Since most devices are built with IPIF, drivers utilize this common source code to prevent duplication of code within the drivers. These functions are used by drivers and are not intended for external use.

DMA

DMA components reside in the *dma* subdirectory and comprise functions used for Direct Memory Access (DMA). Both simple DMA and scatter-gather DMA are supported.

Packet FIFO

Packet FIFO components reside in the *packet_fifo* subdirectory and comprise functions used for packet FIFO control. Packet FIFOs are typically used by devices that process and potentially retransmit packets, such as Ethernet and ATM. These functions are used by drivers and are not intended for external use.

atmc/v1_00_c/src/xatmc.h File Reference

Detailed Description

The implementation of the **XAtmc** component, which is the driver for the Xilinx ATM controller.

The Xilinx ATM controller supports the following features:

- Simple and scatter-gather DMA operations, as well as simple memory mapped direct I/O interface (FIFOs).
- Independent internal transmit and receive FIFOs
- Internal loopback
- Header error check (HEC) generation and checking
- Cell buffering with or without header/User Defined
- Parity generation and checking
- Header generation for transmit cell payloads
- Physical interface (PHY) data path of 16 bits
- Basic statistics gathering such as long cells, short cells, parity errors, and HEC errors

The driver does not support all of the features listed above. Features not currently supported by the driver are:

- Simple DMA (in polled or interrupt mode)
- Direct I/O (FIFO) operations in interrupt mode (polled mode does use the FIFO directly)

It is the responsibility of the application get the interrupt handler of the ATM controller and connect it to the interrupt source.

The driver services interrupts and passes ATM cells to the upper layer software through callback functions. The upper layer software must register its callback functions during initialization. The driver requires callback functions for received cells, for confirmation of transmitted cells, and for asynchronous errors. The frequency of interrupts can be controlled with the packet threshold and packet wait bound features of the scatter-gather DMA engine.

The callback function which performs processing for scatter-gather DMA is executed in an interrupt context and is designed to allow the processing of the scatter-gather list to be passed to a thread context. The scatter-gather processing can require more processing than desired in an interrupt context. Functions are provided to be called from the callback function or thread context to get cells from the send and receive scatter-gather list.

Some errors that can occur in the device require a device reset. These errors are listed in the SetErrorHandler

function header. The upper layer's error handler is responsible for resetting the device and re-configuring it based on its needs (the driver does not save the current configuration).

DMA Support

The Xilinx ATMC device is available for both the IBM On-Chip Peripheral Bus (OPB) and Processor Local Bus (PLB). This driver works for both. However, a current limitation of the ATMC device on the PLB is that it does not support DMA. For this reason, the DMA scatter-gather functions (e.g., [XAtmc_SgSend\(\)](#)) of this driver will not function for the PLB version of the ATMC device.

Note:

Xilinx drivers are typically composed of two components, one is the driver and the other is the adapter. The driver is independent of OS and processor and is intended to be highly portable. The adapter is OS-specific and facilitates communication between the driver and the OS.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	JHL	07/31/01	First release
1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver an v1.23b of the IPIF driver

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xdma_channel.h"  
#include "xbasic_types.h"  
#include "xpacket_fifo_v2_00_a.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XAtmc  
struct XAtmc\_Config  
struct XAtmc\_Stats
```


Configuration options

These options are used in `XAtmc_SetOptions()` to configure the device.

```
#define XAT_LOOPBACK_OPTION
#define XAT_POLLED_OPTION
#define XAT_DISCARD_SHORT_OPTION
#define XAT_DISCARD_PARITY_OPTION
#define XAT_DISCARD_LONG_OPTION
#define XAT_DISCARD_HEC_OPTION
#define XAT_DISCARD_VXI_OPTION
#define XAT_PAYLOAD_ONLY_OPTION
#define XAT_NO_SEND_PARITY_OPTION
```

Cell status

These constants define the status values for a received cell. The status is available when polling to receive a cell or in the buffer descriptor after a cell is received using DMA scatter-gather.

```
#define XAT_CELL_STATUS_LONG
#define XAT_CELL_STATUS_SHORT
#define XAT_CELL_STATUS_BAD_PARITY
#define XAT_CELL_STATUS_BAD_HEC
#define XAT_CELL_STATUS_VXI_MISMATCH
#define XAT_CELL_STATUS_NO_ERROR
```

Typedefs for callbacks

Callback functions.

```
typedef void(* XAtmc_SgHandler )(void *CallBackRef, Xuint32 CellCount)
typedef void(* XAtmc_ErrorHandler )(void *CallBackRef, XStatus ErrorCode)
```

Functions

```
XStatus XAtmc_Initialize (XAtmc *InstancePtr, Xuint16 DeviceId)
XStatus XAtmc_Start (XAtmc *InstancePtr)
```

XStatus XAtmc_Stop (**XAtmc** *InstancePtr)

void **XAtmc_Reset** (**XAtmc** *InstancePtr)

XStatus XAtmc_SelfTest (**XAtmc** *InstancePtr)

XAtmc_Config * **XAtmc_LookupConfig** (**Xuint16** DeviceId)

XStatus XAtmc_SgSend (**XAtmc** *InstancePtr, XBufDescriptor *BdPtr)

XStatus XAtmc_SgRecv (**XAtmc** *InstancePtr, XBufDescriptor *BdPtr)

XStatus XAtmc_SgGetSendCell (**XAtmc** *InstancePtr, XBufDescriptor **PtrToBdPtr, int *BdCountPtr)

XStatus XAtmc_SgGetRecvCell (**XAtmc** *InstancePtr, XBufDescriptor **PtrToBdPtr, int *BdCountPtr)

XStatus XAtmc_PollSend (**XAtmc** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** ByteCount)

XStatus XAtmc_PollRecv (**XAtmc** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** *ByteCountPtr, **Xuint32** *CellStatusPtr)

XStatus XAtmc_SetOptions (**XAtmc** *InstancePtr, **Xuint32** Options)

Xuint32 XAtmc_GetOptions (**XAtmc** *InstancePtr)

XStatus XAtmc_SetPhyAddress (**XAtmc** *InstancePtr, **Xuint8** Address)

Xuint8 XAtmc_GetPhyAddress (**XAtmc** *InstancePtr)

XStatus XAtmc_SetHeader (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** Header)

Xuint32 XAtmc_GetHeader (**XAtmc** *InstancePtr, **Xuint32** Direction)

XStatus XAtmc_SetUserDefined (**XAtmc** *InstancePtr, **Xuint8** UserDefined)

Xuint8 XAtmc_GetUserDefined (**XAtmc** *InstancePtr)

XStatus XAtmc_SetPktThreshold (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint8** Threshold)

XStatus XAtmc_GetPktThreshold (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)

XStatus XAtmc_SetPktWaitBound (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)

XStatus XAtmc_GetPktWaitBound (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)

void **XAtmc_GetStats** (**XAtmc** *InstancePtr, **XAtmc_Stats** *StatsPtr)

void **XAtmc_ClearStats** (**XAtmc** *InstancePtr)

XStatus XAtmc_SetSgRecvSpace (**XAtmc** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

XStatus XAtmc_SetSgSendSpace (**XAtmc** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

void **XAtmc_InterruptHandler** (void *InstancePtr)

void **XAtmc_SetSgRecvHandler** (**XAtmc** *InstancePtr, void *CallBackRef, **XAtmc_SgHandler** FuncPtr)

void **XAtmc_SetSgSendHandler** (**XAtmc** *InstancePtr, void *CallBackRef, **XAtmc_SgHandler** FuncPtr)

void **XAtmc_SetErrorHandler** (**XAtmc** *InstancePtr, void *CallBackRef, **XAtmc_ErrorHandler** FuncPtr)

Define Documentation

```
#define XAT_CELL_STATUS_BAD_HEC
```

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define XAT_CELL_STATUS_BAD_PARITY

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define XAT_CELL_STATUS_LONG

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define XAT_CELL_STATUS_NO_ERROR

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define XAT_CELL_STATUS_SHORT

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define XAT_CELL_STATUS_VXI_MISMATCH

XAT_CELL_STATUS_LONG	Cell was too long
XAT_CELL_STATUS_SHORT	Cell was too short
XAT_CELL_STATUS_BAD_PARITY	Cell parity was not correct
XAT_CELL_STATUS_BAD_HEC	Cell HEC was not correct
XAT_CELL_STATUS_VXI_MISMATCH	Cell VPI/VCI fields didn't match the expected header values
XAT_CELL_STATUS_NO_ERROR	Cell received without errors

#define XAT_DISCARD_HEC_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define XAT_DISCARD_LONG_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define XAT_DISCARD_PARITY_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define XAT_DISCARD_SHORT_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define XAT_DISCARD_VXI_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define XAT_LOOPBACK_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define XAT_NO_SEND_PARITY_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define XAT_PAYLOAD_ONLY_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

#define XAT_POLLED_OPTION

XAT_LOOPBACK_OPTION	Enable sent data to be received
XAT_POLLED_OPTION	Enables polled mode (no interrupts)
XAT_DISCARD_SHORT_OPTION	Discard runt/short cells
XAT_DISCARD_PARITY_OPTION	Discard cells with parity errors
XAT_DISCARD_LONG_OPTION	Discard long cells
XAT_DISCARD_HEC_OPTION	Discard cells with HEC errors
XAT_DISCARD_VXI_OPTION	Discard cells which don't match in the VCI/VPI fields
XAT_PAYLOAD_ONLY_OPTION	Buffer payload only
XAT_NO_SEND_PARITY_OPTION	Disable parity for sent cells

Typedef Documentation

```
typedef void(* XAtmc_ErrorHandler)(void *CallbackRef, XStatus ErrorCode)
```

Callback when data is sent or received with scatter-gather DMA.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

ErrorCode indicates the error that occurred.

```
typedef void(* XAtmc_SgHandler)(void *CallbackRef, Xuint32 CellCount)
```

Callback when data is sent or received with scatter-gather DMA.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

CellCount is the number of cells sent or received.

Function Documentation

```
void XAtmc_ClearStats( XAtmc * InstancePtr)
```

Clears the [XAtmc_Stats](#) structure for this driver.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 XAtmc_GetHeader( XAtmc * InstancePtr,  
                        Xuint32 Direction  
                        )
```

Gets the send or receive ATM header in the ATM controller. The ATM controller attaches the send header to cells which are to be sent but contain only the payload.

If the ATM controller is configured appropriately, it will compare the header of received cells against the receive header and discard cells which don't match in the VCI and VPI fields of the header.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Direction indicates whether we're retrieving the send header or the receive header.

Returns:

The ATM header currently being used by the ATM controller for attachment to transmitted cells or the header which is being compared against received cells. An invalid specified direction will cause this function to return a value of 0.

Note:

None.

Xuint32 XAtmc_GetOptions(**XAtmc** * *InstancePtr*)

Gets Atmc driver/device options. The value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The 32-bit value of the Atmc options. The value is a bit-mask representing all options that are currently enabled. See [xatmc.h](#) for a detailed description of the options.

Note:

None.

Xuint8 XAtmc_GetPhyAddress(**XAtmc** * *InstancePtr*)

Gets the PHY address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The 5-bit PHY address (0 - 31) currently being used by the ATM controller.

Note:

None.

```
XStatus XAtmc_GetPktThreshold( XAtmc * InstancePtr,  
                               Xuint32 Direction,  
                               Xuint8 * ThreshPtr  
                               )
```

Gets the value of the packet threshold register for this driver/device. The packet threshold is used for interrupt coalescing when the ATM controller is configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus XAtmc_GetPktWaitBound( XAtmc * InstancePtr,  
                               Xuint32 Direction,  
                               Xuint32 * WaitPtr  
                               )
```

Gets the packet wait bound register for this driver/device. The packet wait bound is used for interrupt coalescing when the ATM controller is configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
void XAtmc_GetStats( XAtmc * InstancePtr,  
                    XAtmc_Stats * StatsPtr  
                    )
```

Gets a copy of the **XAtmc_Stats** structure, which contains the current statistics for this driver.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None. Although the output parameter will contain a copy of the statistics upon return from this function.

Note:

None.

```
Xuint8 XAtmc_GetUserDefined( XAtmc * InstancePtr)
```

Gets the 2nd byte of the User Defined data in the ATM controller for the channel which is sending data. The ATM controller will attach the header to all cells which are being sent and do not have a header. The header of a 16 bit Utopia interface contains the User Defined data which is two bytes. The first byte contains the HEC field and the second byte is available for user data. This function only allows the second byte to be retrieved.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The second byte of the User Defined data.

Note:

None.

```
XStatus XAtmc_Initialize( XAtmc * InstancePtr,  
                          Xuint16 DeviceId  
                          )
```

Initializes a specific ATM controller instance/driver. The initialization entails:

- Initialize fields of the **XAtmc** structure
- Clear the ATM statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- Configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists will be passed to the driver.
- Reset the ATM controller

The only driver function that should be called before this Initialize function is called is GetInstance.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XAtmc** instance. Passing in a device id associates the generic **XAtmc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started

Note:

None.

```
void XAtmc_InterruptHandler( void * InstancePtr)
```

Interrupt handler for the ATM controller driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: the ATM controller, the send packet FIFO, the receive packet FIFO, the send DMA channel, or the receive DMA channel. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the ATM controller.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the ATMC controller instance for the interrupt.

Returns:

None.

Note:

None.

XAtmc_Config* XAtmc_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table AtmcConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId contains the unique device ID that for the device. This ID is used to lookup the configuration.

Returns:

A pointer to the configuration for the specified device, or XNULL if the device could not be found.

Note:

None.

```
XStatus XAtmc_PollRecv( XAtmc * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 * ByteCountPtr,  
                        Xuint32 * CellStatusPtr  
                        )
```

Receives an ATM cell in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the cell directly from the ATM controller packet FIFO. This is a non-blocking receive, in that if there is no cell ready to be received at the device, the function returns with an error. The buffer into which the cell will be received must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received Atmc cell will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to the size of the buffer on entry into the function and the size the received cell on return from the function.

CellStatusPtr is both an input and an output parameter. It is a pointer to the status of the cell which is received. It is only valid if the return value indicates success. The status is necessary when cells with errors are not being discarded. This status is a bit mask which may contain one or more of the following values with the exception of XAT_CELL_STATUS_NO_ERROR which is mutually exclusive. The status values are:

- XAT_CELL_STATUS_NO_ERROR indicates the cell was received without any errors
- XAT_CELL_STATUS_BAD_PARITY indicates the cell parity was not correct
- XAT_CELL_STATUS_BAD_HEC indicates the cell HEC was not correct
- XAT_CELL_STATUS_SHORT indicates the cell was not the correct length
- XAT_CELL_STATUS_VXI_MISMATCH indicates the cell VPI/VCI fields did not match the expected

header values

Returns:

- XST_SUCCESS if the cell was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no cell to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the cell is too small for the cell waiting in the FIFO.

Note:

The input buffer must be big enough to hold the largest ATM cell. The buffer must also be 32-bit aligned.

```
XStatus XAtmc_PollSend( XAtmc * InstancePtr,  
                       Xuint8 * BufPtr,  
                       Xuint32 ByteCount  
                       )
```

Sends an ATM cell in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the cell directly to the ATM controller packet FIFO, then enters a loop checking the device status for completion or error. The buffer to be sent must be word-aligned.

It is assumed that the upper layer software supplies a correctly formatted ATM cell based upon the configuration of the ATM controller (attaching header or not).

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the ATM cell to be sent.

ByteCount is the size of the ATM cell. An ATM cell for a 16 bit Utopia interface is 54 bytes with a 6 byte header and 48 bytes of payload. This function may be used to send short cells with or without headers depending on the configuration of the ATM controller.

Returns:

- XST_SUCCESS if the cell was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_PFFIFO_NO_ROOM if there is no room in the FIFO for this cell
- XST_FIFO_ERROR if the FIFO was overrun or underrun

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread.

The input buffer must be big enough to hold the largest ATM cell. The buffer must also be 32-bit aligned.

```
void XAtmc_Reset( XAtmc * InstancePtr)
```

Resets the ATM controller. It resets the the DMA channels, the FIFOs, and the ATM controller. The reset does not remove any of the buffer descriptors from the scatter-gather list for DMA. Reset must only be called after the driver has been initialized.

The configuration after this reset is as follows:

- Disabled transmitter and receiver
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- PHY address of 0

The upper layer software is responsible for re-configuring (if necessary) and restarting the ATM controller after the reset.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

None.

Note:

The reset is accomplished by setting the IPIF reset register. This takes care of resetting all hardware blocks, including the ATM controller.

```
XStatus XAtmc_SelfTest( XAtmc * InstancePtr)
```

Performs a self-test on the ATM controller device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the ATM controller device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if self-test was successful
- XST_PFIPO_BAD_REG_VALUE if the FIFO failed register self-test
- XST_DMA_TRANSFER_ERROR if DMA failed data transfer self-test
- XST_DMA_RESET_REGISTER_ERROR if DMA control register value was incorrect after a reset
- XST_REGISTER_ERROR if the ATM controller failed register reset test
- XST_LOOPBACK_ERROR if the ATM controller internal loopback failed
- XST_IPIF_REG_WIDTH_ERROR if an invalid register width was passed into the function
- XST_IPIF_RESET_REGISTER_ERROR if the value of a register at reset was invalid
- XST_IPIF_DEVICE_STATUS_ERROR if a write to the device status register did not read back correctly
- XST_IPIF_DEVICE_ACK_ERROR if a bit in the device status register did not reset when acked
- XST_IPIF_DEVICE_ENABLE_ERROR if the device interrupt enable register was not updated correctly by the hardware when other registers were written to
- XST_IPIF_IP_STATUS_ERROR if a write to the IP interrupt status register did not read back correctly
- XST_IPIF_IP_ACK_ERROR if one or more bits in the IP status register did not reset when acked
- XST_IPIF_IP_ENABLE_ERROR if the IP interrupt enable register was not updated correctly when other registers were written to

Note:

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the self-test thread.


```
void XAtmc_SetErrorHandler( XAtmc *           InstancePtr,  
                           void *           CallbackRef,  
                           XAtmc_ErrorHandler FuncPtr  
                           )
```

Sets the callback function for handling errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback which should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable ATM controller error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_ATMC_ERROR_COUNT_MAX indicates the counters of the ATM controller have reached the maximum value and that the statistics of the ATM controller should be cleared.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetHeader( XAtmc * InstancePtr,  
                        Xuint32 Direction,  
                        Xuint32 Header  
                        )
```

Sets the send or receive ATM header in the ATM controller. If cells with only payloads are given to the controller to be sent, it will attach the header to the cells. If the ATM controller is configured appropriately, it will compare the header of received cells against the receive header and discard cells which don't match in the VCI and VPI fields of the header.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the direction, send(transmit) or receive, for the header to set.

Header contains the ATM header to be attached to each transmitted cell for cells with only payloads or the expected header for cells which are received.

Returns:

- XST_SUCCESS if the PHY address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus XAtmc_SetOptions( XAtmc * InstancePtr,  
                          Xuint32 OptionsFlag  
                          )
```

Set Atmc driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option. A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off. See [xatmc.h](#) for a detailed description of the available options.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

OptionsFlag is a bit-mask representing the Atmc options to turn on or off

Returns:

- XST_SUCCESS if options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
XStatus XAtmc_SetPhyAddress( XAtmc * InstancePtr,  
                             Xuint8 Address  
                             )
```

Sets the PHY address for this driver/device. The address is a 5-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Address contains the 5-bit PHY address (0 - 31).

Returns:

- XST_SUCCESS if the PHY address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus XAtmc_SetPktThreshold( XAtmc * InstancePtr,  
                               Xuint32 Direction,  
                               Xuint8 Threshold  
                               )
```

Sets the packet count threshold register for this driver/device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_DMA_SG_COUNT_EXCEEDED if the threshold must be equal to or less than the number of descriptors in the list
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus XAtmc_SetPktWaitBound( XAtmc * InstancePtr,  
                                Xuint32 Direction,  
                                Xuint32 TimerValue  
                                )
```

Sets the packet wait bound register for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold.

The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
void XAtmc_SetSgRecvHandler( XAtmc * InstancePtr,  
                             void * CallbackRef,  
                             XAtmc_SgHandler FuncPtr  
                             )
```

Sets the callback function for handling received cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are received. The number of received cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each received cell from the list and should attach a new buffer to each descriptor. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetSgRecvSpace( XAtmc * InstancePtr,  
                             Xuint32 * MemoryPtr,  
                             Xuint32 ByteCount  
                             )
```

Gives the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Atmc driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XAtmc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
void XAtmc_SetSgSendHandler( XAtmc * InstancePtr,  
                             void * CallbackRef,  
                             XAtmc_SgHandler FuncPtr  
                             )
```

Sets the callback function for handling confirmation of transmitted cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are sent. The number of sent cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each sent cell from the list and should also free the buffers attached to the descriptors if necessary. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetSgSendSpace( XAtmc * InstancePtr,  
                             Xuint32 * MemoryPtr,  
                             Xuint32 ByteCount  
                             )
```

Gives the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Atmc driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the `XAtmc_Initialize` function because the DMA channel components must be initialized before the memory space is set.

```
XStatus XAtmc_SetUserDefined( XAtmc * InstancePtr,
                             Xuint8  UserDefined
                             )
```

Sets the 2nd byte of the User Defined data in the ATM controller for the channel which is sending data. The ATM controller will attach the header to all cells which are being sent and do not have a header. The header of a 16 bit Utopia interface contains the User Defined data which is two bytes. The first byte contains the HEC field and the second byte is available for user data. This function only allows the second byte to be set.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

UserDefined contains the second byte of the User Defined data.

Returns:

- XST_SUCCESS if the user-defined data was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus XAtmc_SgGetRecvCell( XAtmc * InstancePtr,
                             XBufDescriptor ** PtrToBdPtr,
                             int * BdCountPtr
                             )
```

Gets the first buffer descriptor of the oldest cell which was received by the scatter-gather DMA channel of the ATM controller. This function is provided to be called from a callback function such that the buffer descriptors for received cells can be processed. The function should be called by the application repetitively for the number of cells indicated as an argument in the callback function. This function may also be used when only payloads are being sent and received by the ATM controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the cell. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the cell. This input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the cell pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- `XST_SUCCESS` if a descriptor was successfully returned to the driver.
- `XST_NOT_SGDMA` if the device is not in scatter-gather DMA mode.
- `XST_DMA_SG_NO_LIST` if the scatter gather list has not been created.
- `XST_DMA_SG_LIST_EMPTY` if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus XAtmc_SgGetSendCell( XAtmc * InstancePtr,  
                             XBufDescriptor ** PtrToBdPtr,  
                             int * BdCountPtr  
                             )
```

Gets the first buffer descriptor of the oldest cell which was sent by the scatter-gather DMA channel of the ATM controller. This function is provided to be called from a callback function such that the buffer descriptors for sent cells can be processed. The function should be called by the application repetitively for the number of cells indicated as an argument in the callback function. This function may also be used when only payloads are being sent and received by the ATM controller.

Parameters:

InstancePtr is a pointer to the `XAtmc` instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the cell. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the cell. this input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the cell pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- `XST_SUCCESS` if a descriptor was successfully returned to the driver.
- `XST_NOT_SGDMA` if the device is not in scatter-gather DMA mode.
- `XST_DMA_SG_NO_LIST` if the scatter gather list has not been created.
- `XST_DMA_SG_LIST_EMPTY` if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.


```
XStatus XAtmc_SgRecv( XAtmc * InstancePtr,  
                    XBufDescriptor * BdPtr  
                    )
```

Adds this descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of cells to replace filled buffers with empty buffers. The contents of the specified buffer descriptor are copied into the scatter-gather transmit list. This function can be called when the device is started or stopped.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

Note:

None.

```
XStatus XAtmc_SgSend( XAtmc * InstancePtr,  
                    XBufDescriptor * BdPtr  
                    )
```

Sends an ATM cell using scatter-gather DMA. The caller attaches the cell to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire ATM cell may or may not be contained within one descriptor. The contents of the buffer descriptor are copied into the scatter-gather transmit list. The caller is responsible for providing mutual exclusion to guarantee that a cell is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the cell, the inserts are committed, which means the descriptors for this cell are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted ATM cell based upon the configuration of the ATM controller (attaching header or not). The ATM controller must be started before calling this function.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Returns:

- XST_SUCCESS if the buffer was successfully sent
- XST_DEVICE_IS_STOPPED if the ATM controller has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

XStatus XAtmc_Start(XAtmc * InstancePtr)

Starts the ATM controller as follows:

- If not in polled mode enable interrupts
- Enable the transmitter
- Enable the receiver
- Start the DMA channels if the descriptor lists are not empty

It is necessary for the caller to connect the interrupt service routine of the ATM controller to the interrupt source, typically an interrupt controller, and enable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.
- XST_DMA_SG_LIST_EMPTY iff configured for scatter-gather DMA and no buffer descriptors have been put into the list for the receive channel.

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the

user is required to provide protection of this shared data (typically using a semaphore).

XStatus XAtmc_Stop(XAtmc * InstancePtr)

Stops the ATM controller as follows:

- Stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode

It is the callers responsibility to disconnect the interrupt handler of the ATM controller from the interrupt source, typically an interrupt controller, and disable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

(c) Copyright 2003 Xilinx Inc.
All rights reserved.

Xilinx Device Drivers Data Structures

Here are the data structures with brief descriptions:

XAtmc
XAtmc_Config
XAtmc_Stats
XEmac
XEmac_Config
XEmac_Stats
XEmc
XEmc_Config
XENV_TIME_STAMP
XFlash_Config
XFlashGeometry
XFlashPartID
XFlashProgCap
XFlashProperties
XFlashTag
XFlashTiming
XGemac
XGemac_Config
XGemac_SoftStats
XGemac_Stats
XGpio
XHdlc
XHdlc_Config
XHdlc_Stats

XIic
XIic_Config
XIicStats
XIntc
XIntc_Config
XOpb2Plb
XOpb2Plb_Config
XOpbArb
XOpbArb_Config
XPci
XPciError
XPlb2Opb
XPlb2Opb_Config
XPlbArb
XPlbArb_Config
XSpi
XSpi_Config
XSpi_Stats
XSysAce
XSysAce_CFPParameters
XSysAce_Config
XTmrCtr
XTmrCtr_Config
XTmrCtrStats
XUartLite
XUartLite_Buffer
XUartLite_Config
XUartLite_Stats
XUartNs550
XUartNs550_Config
XUartNs550Format
XUartNs550Stats

XWdtTb

XWdtTb_Config

Generated on 29 May 2003 for Xilinx Device Drivers

Xilinx Device Drivers File List

Here is a list of all documented files with brief descriptions:

atmc/v1_00_c/src/xatmc.c
atmc/v1_00_c/src/xatmc.h [code]
atmc/v1_00_c/src/xatmc_cfg.c
atmc/v1_00_c/src/xatmc_g.c
atmc/v1_00_c/src/xatmc_i.h [code]
atmc/v1_00_c/src/xatmc_l.c
atmc/v1_00_c/src/xatmc_l.h [code]
channel_fifo/v1_00_a/src/xchannel_fifo_v1_00_a.h [code]
common/v1_00_a/src/xbasic_types.c
common/v1_00_a/src/xbasic_types.h [code]
common/v1_00_a/src/xenv.h [code]
common/v1_00_a/src/xenv_linux.h [code]
common/v1_00_a/src/xenv_none.h [code]
common/v1_00_a/src/xenv_vxworks.h [code]
common/v1_00_a/src/xparameters.h [code]
common/v1_00_a/src/xstatus.h [code]
common/v1_00_a/src/xutil.h [code]
common/v1_00_a/src/xutil_memtest.c
cpu/v1_00_a/src/xio.c
cpu/v1_00_a/src/xio.h [code]
cpu_ppc405/v1_00_a/src/xio.c
cpu_ppc405/v1_00_a/src/xio.h [code]
cpu_ppc405/v1_00_a/src/xio_dcr.c
cpu_ppc405/v1_00_a/src/xio_dcr.h [code]

[emac/v1_00_c/src/xemac.c](#)

[emac/v1_00_c/src/xemac.h \[code\]](#)

[emac/v1_00_c/src/xemac_g.c](#)

[emac/v1_00_c/src/xemac_i.h \[code\]](#)

[emac/v1_00_c/src/xemac_intr.c](#)

[emac/v1_00_c/src/xemac_intr_dma.c](#)

[emac/v1_00_c/src/xemac_intr_fifo.c](#)

[emac/v1_00_c/src/xemac_l.c](#)

[emac/v1_00_c/src/xemac_l.h \[code\]](#)

[emac/v1_00_c/src/xemac_multicast.c](#)

[emac/v1_00_c/src/xemac_options.c](#)

[emac/v1_00_c/src/xemac_phy.c](#)

[emac/v1_00_c/src/xemac_polled.c](#)

[emac/v1_00_c/src/xemac_selftest.c](#)

[emac/v1_00_c/src/xemac_stats.c](#)

[emaclite/v1_00_a/src/xemaclite_l.c](#)

[emaclite/v1_00_a/src/xemaclite_l.h \[code\]](#)

[emc/v1_00_a/src/xemc.c](#)

[emc/v1_00_a/src/xemc.h \[code\]](#)

[emc/v1_00_a/src/xemc_g.c](#)

[emc/v1_00_a/src/xemc_i.h \[code\]](#)

[emc/v1_00_a/src/xemc_l.h \[code\]](#)

[emc/v1_00_a/src/xemc_selftest.c](#)

[flash/v1_00_a/src/xflash.c](#)

[flash/v1_00_a/src/xflash.h \[code\]](#)

[flash/v1_00_a/src/xflash_cfi.c](#)

[flash/v1_00_a/src/xflash_cfi.h \[code\]](#)

[flash/v1_00_a/src/xflash_g.c](#)

[flash/v1_00_a/src/xflash_geometry.c](#)

[flash/v1_00_a/src/xflash_geometry.h \[code\]](#)

[flash/v1_00_a/src/xflash_intel.c](#)

[flash/v1_00_a/src/xflash_intel.h \[code\]](#)

flash/v1_00_a/src/xflash_intel_l.c
flash/v1_00_a/src/xflash_intel_l.h <u>[code]</u>
flash/v1_00_a/src/xflash_properties.h <u>[code]</u>
gemac/v1_00_c/src/xgemac.c
gemac/v1_00_c/src/xgemac.h <u>[code]</u>
gemac/v1_00_c/src/xgemac_g.c
gemac/v1_00_c/src/xgemac_i.h <u>[code]</u>
gemac/v1_00_c/src/xgemac_intr.c
gemac/v1_00_c/src/xgemac_intr_dma.c
gemac/v1_00_c/src/xgemac_intr_fifo.c
gemac/v1_00_c/src/xgemac_l.h <u>[code]</u>
gemac/v1_00_c/src/xgemac_options.c
gemac/v1_00_c/src/xgemac_polled.c
gemac/v1_00_c/src/xgemac_selftest.c
gemac/v1_00_c/src/xgemac_stats.c
gpio/v1_00_a/src/xgpio.c
gpio/v1_00_a/src/xgpio.h <u>[code]</u>
gpio/v1_00_a/src/xgpio_extra.c
gpio/v1_00_a/src/xgpio_g.c
gpio/v1_00_a/src/xgpio_i.h <u>[code]</u>
gpio/v1_00_a/src/xgpio_l.h <u>[code]</u>
gpio/v1_00_a/src/xgpio_selftest.c
hdlc/v1_00_a/src/xhdlc.c
hdlc/v1_00_a/src/xhdlc.h <u>[code]</u>
hdlc/v1_00_a/src/xhdlc_dmasg.c
hdlc/v1_00_a/src/xhdlc_g.c
hdlc/v1_00_a/src/xhdlc_i.h <u>[code]</u>
hdlc/v1_00_a/src/xhdlc_l.c
hdlc/v1_00_a/src/xhdlc_l.h <u>[code]</u>
hdlc/v1_00_a/src/xhdlc_options.c
hdlc/v1_00_a/src/xhdlc_selftest.c
hdlc/v1_00_a/src/xhdlc_stats.c

[iic/v1_01_c/src/xiic.c](#)

[iic/v1_01_c/src/xiic.h](#) [\[code\]](#)

[iic/v1_01_c/src/xiic_g.c](#)

[iic/v1_01_c/src/xiic_i.h](#) [\[code\]](#)

[iic/v1_01_c/src/xiic_intr.c](#)

[iic/v1_01_c/src/xiic_l.c](#)

[iic/v1_01_c/src/xiic_l.h](#) [\[code\]](#)

[iic/v1_01_c/src/xiic_master.c](#)

[iic/v1_01_c/src/xiic_multi_master.c](#)

[iic/v1_01_c/src/xiic_options.c](#)

[iic/v1_01_c/src/xiic_selftest.c](#)

[iic/v1_01_c/src/xiic_slave.c](#)

[iic/v1_01_c/src/xiic_stats.c](#)

[intc/v1_00_b/src/xintc.c](#)

[intc/v1_00_b/src/xintc.h](#) [\[code\]](#)

[intc/v1_00_b/src/xintc_g.c](#)

[intc/v1_00_b/src/xintc_i.h](#) [\[code\]](#)

[intc/v1_00_b/src/xintc_intr.c](#)

[intc/v1_00_b/src/xintc_l.c](#)

[intc/v1_00_b/src/xintc_l.h](#) [\[code\]](#)

[intc/v1_00_b/src/xintc_lg.c](#)

[intc/v1_00_b/src/xintc_options.c](#)

[intc/v1_00_b/src/xintc_selftest.c](#)

[opb2plb/v1_00_a/src/xopb2plb.c](#)

[opb2plb/v1_00_a/src/xopb2plb.h](#) [\[code\]](#)

[opb2plb/v1_00_a/src/xopb2plb_g.c](#)

[opb2plb/v1_00_a/src/xopb2plb_i.h](#) [\[code\]](#)

[opb2plb/v1_00_a/src/xopb2plb_l.h](#) [\[code\]](#)

[opb2plb/v1_00_a/src/xopb2plb_selftest.c](#)

[opbarb/v1_02_a/src/xopbarb.c](#)

[opbarb/v1_02_a/src/xopbarb.h](#) [\[code\]](#)

[opbarb/v1_02_a/src/xopbarb_g.c](#)

[opbarb/v1_02_a/src/xopbarb_l.h \[code\]](#)

[packet_fifo/v2_00_a/src/xpacket_fifo_v2_00_a.h \[code\]](#)

[pci/v1_00_a/src/xpci.h \[code\]](#)

[pci/v1_00_a/src/xpci_config.c](#)

[pci/v1_00_a/src/xpci_g.c](#)

[pci/v1_00_a/src/xpci_intr.c](#)

[pci/v1_00_a/src/xpci_l.h \[code\]](#)

[pci/v1_00_a/src/xpci_selftest.c](#)

[pci/v1_00_a/src/xpci_v3.c](#)

[plb2opb/v1_00_a/src/xplb2opb.c](#)

[plb2opb/v1_00_a/src/xplb2opb.h \[code\]](#)

[plb2opb/v1_00_a/src/xplb2opb_g.c](#)

[plb2opb/v1_00_a/src/xplb2opb_i.h \[code\]](#)

[plb2opb/v1_00_a/src/xplb2opb_l.h \[code\]](#)

[plb2opb/v1_00_a/src/xplb2opb_selftest.c](#)

[plbarb/v1_01_a/src/xplbarb.c](#)

[plbarb/v1_01_a/src/xplbarb.h \[code\]](#)

[plbarb/v1_01_a/src/xplbarb_g.c](#)

[plbarb/v1_01_a/src/xplbarb_i.h \[code\]](#)

[plbarb/v1_01_a/src/xplbarb_l.h \[code\]](#)

[plbarb/v1_01_a/src/xplbarb_selftest.c](#)

[ps2_ref/v1_00_a/src/xps2.c](#)

[ps2_ref/v1_00_a/src/xps2.h \[code\]](#)

[ps2_ref/v1_00_a/src/xps2_g.c](#)

[ps2_ref/v1_00_a/src/xps2_i.h \[code\]](#)

[ps2_ref/v1_00_a/src/xps2_intr.c](#)

[ps2_ref/v1_00_a/src/xps2_l.c](#)

[ps2_ref/v1_00_a/src/xps2_l.h \[code\]](#)

[ps2_ref/v1_00_a/src/xps2_options.c](#)

[ps2_ref/v1_00_a/src/xps2_stats.c](#)

[rapidio/v1_00_a/src/xrapidio_l.c](#)

[rapidio/v1_00_a/src/xrapidio_l.h \[code\]](#)

[spi/v1_00_b/src/xspi.c](#)

[spi/v1_00_b/src/xspi.h](#) [\[code\]](#)

[spi/v1_00_b/src/xspi_g.c](#)

[spi/v1_00_b/src/xspi_i.h](#) [\[code\]](#)

[spi/v1_00_b/src/xspi_l.h](#) [\[code\]](#)

[spi/v1_00_b/src/xspi_options.c](#)

[spi/v1_00_b/src/xspi_selftest.c](#)

[spi/v1_00_b/src/xspi_stats.c](#)

[sysace/v1_00_a/src/xsysace.c](#)

[sysace/v1_00_a/src/xsysace.h](#) [\[code\]](#)

[sysace/v1_00_a/src/xsysace_compactflash.c](#)

[sysace/v1_00_a/src/xsysace_g.c](#)

[sysace/v1_00_a/src/xsysace_intr.c](#)

[sysace/v1_00_a/src/xsysace_jtagcfg.c](#)

[sysace/v1_00_a/src/xsysace_l.c](#)

[sysace/v1_00_a/src/xsysace_l.h](#) [\[code\]](#)

[sysace/v1_00_a/src/xsysace_selftest.c](#)

[tmrctr/v1_00_b/src/xtmrctr.c](#)

[tmrctr/v1_00_b/src/xtmrctr.h](#) [\[code\]](#)

[tmrctr/v1_00_b/src/xtmrctr_g.c](#)

[tmrctr/v1_00_b/src/xtmrctr_i.h](#) [\[code\]](#)

[tmrctr/v1_00_b/src/xtmrctr_intr.c](#)

[tmrctr/v1_00_b/src/xtmrctr_l.c](#)

[tmrctr/v1_00_b/src/xtmrctr_l.h](#) [\[code\]](#)

[tmrctr/v1_00_b/src/xtmrctr_options.c](#)

[tmrctr/v1_00_b/src/xtmrctr_selftest.c](#)

[tmrctr/v1_00_b/src/xtmrctr_stats.c](#)

[touchscreen_ref/v1_00_a/src/xtouchscreen.c](#)

[touchscreen_ref/v1_00_a/src/xtouchscreen.h](#) [\[code\]](#)

[touchscreen_ref/v1_00_a/src/xtouchscreen_g.c](#)

[touchscreen_ref/v1_00_a/src/xtouchscreen_i.h](#) [\[code\]](#)

[touchscreen_ref/v1_00_a/src/xtouchscreen_intr.c](#)

[touchscreen_ref/v1_00_a/src/xtouchscreen_l.c](#)

[touchscreen_ref/v1_00_a/src/xtouchscreen_l.h](#) [\[code\]](#)

[uartlite/v1_00_b/src/xuartlite.c](#)

[uartlite/v1_00_b/src/xuartlite.h](#) [\[code\]](#)

[uartlite/v1_00_b/src/xuartlite_g.c](#)

[uartlite/v1_00_b/src/xuartlite_i.h](#) [\[code\]](#)

[uartlite/v1_00_b/src/xuartlite_intr.c](#)

[uartlite/v1_00_b/src/xuartlite_l.c](#)

[uartlite/v1_00_b/src/xuartlite_l.h](#) [\[code\]](#)

[uartlite/v1_00_b/src/xuartlite_selftest.c](#)

[uartlite/v1_00_b/src/xuartlite_stats.c](#)

[uartns550/v1_00_b/src/xuartns550.c](#)

[uartns550/v1_00_b/src/xuartns550.h](#) [\[code\]](#)

[uartns550/v1_00_b/src/xuartns550_format.c](#)

[uartns550/v1_00_b/src/xuartns550_g.c](#)

[uartns550/v1_00_b/src/xuartns550_i.h](#) [\[code\]](#)

[uartns550/v1_00_b/src/xuartns550_intr.c](#)

[uartns550/v1_00_b/src/xuartns550_l.c](#)

[uartns550/v1_00_b/src/xuartns550_l.h](#) [\[code\]](#)

[uartns550/v1_00_b/src/xuartns550_options.c](#)

[uartns550/v1_00_b/src/xuartns550_selftest.c](#)

[uartns550/v1_00_b/src/xuartns550_stats.c](#)

[wdttb/v1_00_b/src/xwdttb.c](#)

[wdttb/v1_00_b/src/xwdttb.h](#) [\[code\]](#)

[wdttb/v1_00_b/src/xwdttb_g.c](#)

[wdttb/v1_00_b/src/xwdttb_i.h](#) [\[code\]](#)

[wdttb/v1_00_b/src/xwdttb_l.h](#) [\[code\]](#)

[wdttb/v1_00_b/src/xwdttb_selftest.c](#)

Xilinx Device Drivers Data Fields

[a](#) | [b](#) | [c](#) | [d](#) | [e](#) | [f](#) | [h](#) | [i](#) | [l](#) | [m](#) | [n](#) | [p](#) | [r](#) | [s](#) | [t](#) | [u](#) | [v](#) | [w](#) | [x](#)

Here is a list of all documented struct and union fields with links to the structures/unions they belong to:

- a -

- AbsoluteBlock : [XFlashGeometry](#)
- AbsoluteOffset : [XFlashGeometry](#)
- AckBeforeService : [XIntc_Config](#)
- ArbitrationLost : [XIicStats](#)
- AtmcInterrupts : [XAtmc_Stats](#)

- b -

- BaseAddr : [XWdtTb_Config](#), [XFlash_Config](#)
- BaseAddress : [XUartNs550_Config](#), [XTmrCtr_Config](#), [XSysAce_Config](#), [XSpi_Config](#), [XPlbArb_Config](#), [XPlb2Opb_Config](#), [XOpbArb_Config](#), [XOpb2Plb_Config](#), [XIntc_Config](#), [XIic_Config](#), [XHdlc_Config](#), [XGemac_Config](#), [XFlashGeometry](#), [XEmac_Config](#), [XAtmc_Config](#)
- BaudRate : [XUartNs550Format](#), [XUartLite_Config](#)
- BufferSize : [XSysAce_CFPParameters](#)
- BufferType : [XSysAce_CFPParameters](#)
- BusBusy : [XIicStats](#)
- BytesTransferred : [XSpi_Stats](#)

- c -

- Capabilities : [XSysAce_CFPParameters](#)
- CharactersReceived : [XUartNs550Stats](#), [XUartLite_Stats](#)
- CharactersTransmitted : [XUartNs550Stats](#), [XUartLite_Stats](#)

- CommandSet : **XFlashPartID**
- CurNumCylinders : **XSysAce_CFPParameters**
- CurNumHeads : **XSysAce_CFPParameters**
- CurSectorsPerCard : **XSysAce_CFPParameters**
- CurSectorsPerTrack : **XSysAce_CFPParameters**

- d -

- DataBits : **XUartNs550Format, XUartLite_Config**
- DblWord : **XSysAce_CFPParameters**
- DeviceID : **XFlashPartID**
- DeviceId : **XWdtTb_Config, XUartNs550_Config, XUartLite_Config, XTmrCtr_Config, XSysAce_Config, XSpi_Config, XPlbArb_Config, XPlb2Opb_Config, XOpbArb_Config, XOpb2Plb_Config, XIntc_Config, XIic_Config, XHdlc_Config, XGemac_Config, XFlash_Config, XEmc_Config, XEmac_Config, XAtmc_Config**
- DeviceSize : **XFlashGeometry**
- DmaErrors : **XHdlc_Stats, XGemac_SoftStats, XGemac_Stats, XEmac_Stats, XAtmc_Stats**
- DmaMode : **XSysAce_CFPParameters**
- DmaRegBaseAddr : **XPci**
- DmaType : **XPci**

- e -

- EmacInterrupts : **XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- EraseBlock_Ms : **XFlashTiming**
- EraseChip_Ms : **XFlashTiming**
- EraseQueueSize : **XFlashProgCap**

- f -

- FifoErrors : **XHdlc_Stats, XGemac_SoftStats, XGemac_Stats, XEmac_Stats, XAtmc_Stats**
- FwVersion : **XSysAce_CFPParameters**

- h -

- Has10BitAddr : **XIic_Config**
- HasCounters : **XEmac_Config**
- HasFifos : **XSpi_Config**

- HasMii : **XGemac_Config, XEmac_Config**
- HdlcInterrupts : **XHdlc_Stats**

- i -

- IicInterrupts : **XIicStats**
- InputClockHz : **XUartNs550_Config**
- Interrupts : **XTmrCtrStats**
- IpIfDmaConfig : **XHdlc_Config, XGemac_Config, XEmac_Config, XAtmc_Config**
- IsError : **XPciError**
- IsReady : **XPci**

- l -

- LbaSectors : **XSysAce_CFPParameters**
- LocalBusReadAddr : **XPciError**
- LocalBusReason : **XPciError**
- LocalBusWriteAddr : **XPciError**

- m -

- ManufacturerID : **XFlashPartID**
- MaxSectors : **XSysAce_CFPParameters**
- MemoryLayout : **XFlashGeometry**
- ModeFaults : **XSpi_Stats**
- ModelNo : **XSysAce_CFPParameters**
- ModemInterrupts : **XUartNs550Stats**
- MultipleSectors : **XSysAce_CFPParameters**

- n -

- NumBanks : **XEmc_Config**
- Number : **XFlashGeometry**
- NumBlocks : **XFlashGeometry**
- NumBytesPerSector : **XSysAce_CFPParameters**
- NumBytesPerTrack : **XSysAce_CFPParameters**
- NumCylinders : **XSysAce_CFPParameters**
- NumEccBytes : **XSysAce_CFPParameters**

- NumEraseRegions : **XFlashGeometry**
- NumHeads : **XSysAce_CFPParameters**
- NumInterrupts : **XSpi_Stats**
- NumMasters : **XPlbArb_Config, XPlb2Opb_Config, XOpbArb_Config**
- NumParts : **XFlash_Config**
- NumSectorsPerCard : **XSysAce_CFPParameters**
- NumSectorsPerTrack : **XSysAce_CFPParameters**
- NumSlaveBits : **XSpi_Config**

- p -

- Parity : **XUartNs550Format**
- ParityOdd : **XUartLite_Config**
- PartID : **XFlashProperties**
- PartMode : **XFlash_Config**
- PartWidth : **XFlash_Config**
- PciReadAddr : **XPciError**
- PciReason : **XPciError**
- PciSerrReadAddr : **XPciError**
- PciSerrReason : **XPciError**
- PciSerrWriteAddr : **XPciError**
- PciWriteAddr : **XPciError**
- PioMode : **XSysAce_CFPParameters**
- PowerDesc : **XSysAce_CFPParameters**
- ProgCap : **XFlashProperties**

- r -

- ReceiveBreakDetected : **XUartNs550Stats**
- ReceiveFifoSize : **XHdlc_Config**
- ReceiveFramingErrors : **XUartNs550Stats, XUartLite_Stats**
- ReceiveInterrupts : **XUartNs550Stats, XUartLite_Stats**
- ReceiveOverrunErrors : **XUartNs550Stats, XUartLite_Stats**
- ReceiveParityErrors : **XUartNs550Stats, XUartLite_Stats**
- RecvAbortedFrames : **XHdlc_Stats**
- RecvAlignmentErrors : **XHdlc_Stats, XEmac_Stats**
- RecvBytes : **XIicStats, XHdlc_Stats, XGemac_Stats, XEmac_Stats**
- RecvBytesMSL : **XGemac_Stats**

- RecvCells : **XAtmc_Stats**
- RecvCollisionErrors : **XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- RecvCollisionErrorsMSL : **XGemac_Stats**
- RecvFcsErrors : **XHdlc_Stats, XGemac_Stats, XEmac_Stats**
- RecvFcsErrorsMSL : **XGemac_Stats**
- RecvFrames : **XHdlc_Stats, XGemac_Stats, XEmac_Stats**
- RecvFramesMSL : **XGemac_Stats**
- RecvHecErrors : **XAtmc_Stats**
- RecvInterrupts : **XIicStats, XHdlc_Stats, XGemac_SoftStats, XGemac_Stats, XEmac_Stats, XAtmc_Stats**
- RecvLengthFieldErrors : **XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- RecvLongCells : **XAtmc_Stats**
- RecvLongErrors : **XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- RecvMissedFrameErrors : **XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- RecvOverrunErrors : **XHdlc_Stats, XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- RecvOverruns : **XSpi_Stats**
- RecvParityErrors : **XAtmc_Stats**
- RecvShortCells : **XAtmc_Stats**
- RecvShortErrors : **XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- RecvUnderrunErrors : **XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- RecvUnexpectedHeaders : **XAtmc_Stats**
- RegBaseAddr : **XUartLite_Config, XPCI, XEmc_Config**
- RepeatedStarts : **XIicStats**

- S -

- SecurityStatus : **XSysAce_CFPParameters**
- SendBytes : **XIicStats**
- SendInterrupts : **XIicStats**
- SerialNo : **XSysAce_CFPParameters**
- Signature : **XSysAce_CFPParameters**
- Size : **XFlashGeometry**
- SlaveModeFaults : **XSpi_Stats**
- SlaveOnly : **XSpi_Config**
- SlotLengthErrors : **XGemac_SoftStats, XGemac_Stats**
- StatusInterrupts : **XUartNs550Stats**
- StopBits : **XUartNs550Format**

- t -

- TimeMax : **XFlashProperties**
- TimeTypical : **XFlashProperties**
- TotalIntrs : **XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- TranslationValid : **XSysAce_CFPParameters**
- TransmitFifoSize : **XHdlc_Config**
- TransmitInterrupts : **XUartNs550Stats, XUartLite_Stats**
- TxErrors : **XIicStats**

- u -

- UseParity : **XUartLite_Config**

- v -

- VendorUnique : **XSysAce_CFPParameters**
- VendorUniqueBytes : **XSysAce_CFPParameters**

- w -

- WriteBuffer_Us : **XFlashTiming**
- WriteBufferAlignmentMask : **XFlashProgCap**
- WriteBufferSize : **XFlashProgCap**
- WriteSingle_Us : **XFlashTiming**

- x -

- XmitBytes : **XHdlc_Stats, XGemac_Stats, XEmac_Stats**
- XmitBytesMSL : **XGemac_Stats**
- XmitCells : **XAtmc_Stats**
- XmitExcessDeferral : **XGemac_Stats, XEmac_Stats**
- XmitExcessDeferralMSL : **XGemac_Stats**
- XmitFrames : **XHdlc_Stats, XGemac_Stats, XEmac_Stats**
- XmitFramesMSL : **XGemac_Stats**
- XmitInterrupts : **XHdlc_Stats, XGemac_SoftStats, XGemac_Stats, XEmac_Stats, XAtmc_Stats**
- XmitLateCollisionErrors : **XGemac_Stats, XEmac_Stats**

- XmitLateCollisionErrorsMSL : **XGemac_Stats**
- XmitOverrunErrors : **XGemac_SoftStats, XGemac_Stats, XEmac_Stats**
- XmitUnderrunErrors : **XGemac_Stats, XEmac_Stats**
- XmitUnderrunErrorsMSL : **XGemac_Stats**
- XmitUnderruns : **XSpi_Stats**

Generated on 29 May 2003 for Xilinx Device Drivers

Xilinx Device Drivers Globals

[1](#) | [x](#)

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- l -

- LOOPBACK_BYTE_COUNT : [xhdlc_selftest.c](#)

- x -

- XAssert() : [xbasic_types.h](#), [xbasic_types.c](#)
- XASSERT_NONVOID : [xbasic_types.h](#)
- XASSERT_NONVOID_ALWAYS : [xbasic_types.h](#)
- XASSERT_VOID : [xbasic_types.h](#)
- XASSERT_VOID_ALWAYS : [xbasic_types.h](#)
- XAssertCallback : [xbasic_types.h](#)
- XAssertSetCallback() : [xbasic_types.h](#), [xbasic_types.c](#)
- XAssertStatus : [xbasic_types.h](#), [xbasic_types.c](#)
- XAT_CELL_STATUS_BAD_HEC : [xatmc.h](#)
- XAT_CELL_STATUS_BAD_PARITY : [xatmc.h](#)
- XAT_CELL_STATUS_LONG : [xatmc.h](#)
- XAT_CELL_STATUS_NO_ERROR : [xatmc.h](#)
- XAT_CELL_STATUS_SHORT : [xatmc.h](#)
- XAT_CELL_STATUS_VXI_MISMATCH : [xatmc.h](#)
- XAT_DISCARD_HEC_OPTION : [xatmc.h](#)
- XAT_DISCARD_LONG_OPTION : [xatmc.h](#)
- XAT_DISCARD_PARITY_OPTION : [xatmc.h](#)
- XAT_DISCARD_SHORT_OPTION : [xatmc.h](#)
- XAT_DISCARD_VXI_OPTION : [xatmc.h](#)
- XAT_LOOPBACK_OPTION : [xatmc.h](#)

- XAT_NO_SEND_PARITY_OPTION : [xatmc.h](#)
- XAT_PAYLOAD_ONLY_OPTION : [xatmc.h](#)
- XAT_POLLED_OPTION : [xatmc.h](#)
- XAtmc_ClearStats() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_ConfigTable : [xatmc_i.h](#), [xatmc_g.c](#)
- XAtmc_ErrorHandler : [xatmc.h](#)
- XAtmc_GetHeader() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetOptions() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetPhyAddress() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetPktThreshold() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetPktWaitBound() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetStats() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_GetUserDefined() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_Initialize() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_InterruptHandler() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_LookupConfig() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_mDisable : [xatmc_l.h](#)
- XAtmc_mEnable : [xatmc_l.h](#)
- XAtmc_mIsRxEmpty : [xatmc_l.h](#)
- XAtmc_mIsSgDma : [xatmc_i.h](#)
- XAtmc_mIsTxDone : [xatmc_l.h](#)
- XAtmc_mReadReg : [xatmc_l.h](#)
- XAtmc_mWriteReg : [xatmc_l.h](#)
- XAtmc_PollRecv() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_PollSend() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_RecvCell() : [xatmc_l.h](#), [xatmc_l.c](#)
- XAtmc_Reset() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SelfTest() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SendCell() : [xatmc_l.h](#), [xatmc_l.c](#)
- XAtmc_SetErrorHandler() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetHeader() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetOptions() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetPhyAddress() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetPktThreshold() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetPktWaitBound() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetSgRecvHandler() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetSgRecvSpace() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetSgSendHandler() : [xatmc_cfg.c](#), [xatmc.h](#)

- XAtmc_SetSgSendSpace() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SetUserDefined() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_SgGetRecvCell() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_SgGetSendCell() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_SgHandler : [xatmc.h](#)
- XAtmc_SgRecv() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_SgSend() : [xatmc.h](#), [xatmc.c](#)
- XAtmc_Start() : [xatmc_cfg.c](#), [xatmc.h](#)
- XAtmc_Stop() : [xatmc_cfg.c](#), [xatmc.h](#)
- Xboolean : [xbasic_types.h](#)
- XEM_BROADCAST_OPTION : [xemac.h](#)
- XEM_FDUPLEX_OPTION : [xemac.h](#)
- XEM_FLOW_CONTROL_OPTION : [xemac.h](#)
- XEM_INSERT_ADDR_OPTION : [xemac.h](#)
- XEM_INSERT_FCS_OPTION : [xemac.h](#)
- XEM_INSERT_PAD_OPTION : [xemac.h](#)
- XEM_LOOPBACK_OPTION : [xemac.h](#)
- XEM_OVWRT_ADDR_OPTION : [xemac.h](#)
- XEM_POLLED_OPTION : [xemac.h](#)
- XEM_PROMISC_OPTION : [xemac.h](#)
- XEM_STRIP_PAD_FCS_OPTION : [xemac.h](#)
- XEM_UNICAST_OPTION : [xemac.h](#)
- XEmac_ClearStats() : [xemac_stats.c](#), [xemac.h](#)
- XEmac_ConfigTable : [xemac_i.h](#), [xemac_g.c](#)
- XEmac_ErrorHandler : [xemac.h](#)
- XEmac_FifoHandler : [xemac.h](#)
- XEmac_FifoRecv() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_FifoSend() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_GetInterframeGap() : [xemac_options.c](#), [xemac.h](#)
- XEmac_GetMacAddress() : [xemac.h](#), [xemac.c](#)
- XEmac_GetOptions() : [xemac_options.c](#), [xemac.h](#)
- XEmac_GetPktThreshold() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_GetPktWaitBound() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_GetStats() : [xemac_stats.c](#), [xemac.h](#)
- XEmac_Initialize() : [xemac.h](#), [xemac.c](#)
- XEmac_IntrHandlerDma() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_IntrHandlerFifo() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_LookupConfig() : [xemac.h](#), [xemac.c](#)

- XEmac_mDisable : [xemac_1.h](#)
- XEmac_mEnable : [xemac_1.h](#)
- XEmac_mIsDma : [xemac.h](#)
- XEmac_mIsRxEmpty : [xemac_1.h](#)
- XEmac_mIsSgDma : [xemac.h](#)
- XEmac_mIsSimpleDma : [xemac.h](#)
- XEmac_mIsTxDone : [xemac_1.h](#)
- XEmac_mPhyReset : [xemac_1.h](#)
- XEmac_mReadReg : [xemac_1.h](#)
- XEmac_mSetControlReg : [xemac_1.h](#)
- XEmac_mSetMacAddress : [xemac_1.h](#)
- XEmac_MulticastAdd() : [xemac_multicast.c](#), [xemac.h](#)
- XEmac_MulticastClear() : [xemac_multicast.c](#), [xemac.h](#)
- XEmac_mWriteReg : [xemac_1.h](#)
- XEmac_PhyRead() : [xemac_phy.c](#), [xemac.h](#)
- XEmac_PhyWrite() : [xemac_phy.c](#), [xemac.h](#)
- XEmac_PollRecv() : [xemac_polled.c](#), [xemac.h](#)
- XEmac_PollSend() : [xemac_polled.c](#), [xemac.h](#)
- XEmac_RecvFrame() : [xemac_1.h](#), [xemac_1.c](#)
- XEmac_Reset() : [xemac.h](#), [xemac.c](#)
- XEmac_SelfTest() : [xemac_selftest.c](#), [xemac.h](#)
- XEmac_SendFrame() : [xemac_1.h](#), [xemac_1.c](#)
- XEmac_SetErrorHandler() : [xemac_intr.c](#), [xemac.h](#)
- XEmac_SetFifoRecvHandler() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_SetFifoSendHandler() : [xemac_intr_fifo.c](#), [xemac.h](#)
- XEmac_SetInterframeGap() : [xemac_options.c](#), [xemac.h](#)
- XEmac_SetMacAddress() : [xemac.h](#), [xemac.c](#)
- XEmac_SetOptions() : [xemac_options.c](#), [xemac.h](#)
- XEmac_SetPktThreshold() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetPktWaitBound() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetSgRecvHandler() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetSgRecvSpace() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetSgSendHandler() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SetSgSendSpace() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SgHandler : [xemac.h](#)
- XEmac_SgRecv() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_SgSend() : [xemac_intr_dma.c](#), [xemac.h](#)
- XEmac_Start() : [xemac.h](#), [xemac.c](#)

- XEmac_Stop() : [xemac.h](#), [xemac.c](#)
- XEmacLite_mIsRxEmpty : [xemaclite_1.h](#)
- XEmacLite_mIsTxDone : [xemaclite_1.h](#)
- XEmacLite_RecvFrame() : [xemaclite_1.h](#), [xemaclite_1.c](#)
- XEmacLite_SendFrame() : [xemaclite_1.h](#), [xemaclite_1.c](#)
- XEmacLite_SetMacAddress() : [xemaclite_1.h](#), [xemaclite_1.c](#)
- XEmc_GetAccessSpeed() : [xemc.h](#), [xemc.c](#)
- XEmc_GetPageMode() : [xemc.h](#), [xemc.c](#)
- XEmc_Initialize() : [xemc.h](#), [xemc.c](#)
- XEmc_LookupConfig() : [xemc.h](#), [xemc.c](#)
- XEmc_mDisableFastAccess : [xemc_1.h](#)
- XEmc_mDisablePageMode : [xemc_1.h](#)
- XEmc_mEnableFastAccess : [xemc_1.h](#)
- XEmc_mEnablePageMode : [xemc_1.h](#)
- XEmc_mGetControlReg : [xemc_1.h](#)
- XEmc_mGetOffset : [xemc_1.h](#)
- XEmc_mSetControlReg : [xemc_1.h](#)
- XEmc_SelfTest() : [xemc_selftest.c](#), [xemc.h](#)
- XEmc_SetAccessSpeed() : [xemc.h](#), [xemc.c](#)
- XEmc_SetPageMode() : [xemc.h](#), [xemc.c](#)
- XENV_MEM_COPY : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_MEM_FILL : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_TIME_STAMP : [xenv_none.h](#), [xenv_linux.h](#)
- XENV_TIME_STAMP_DELTA_MS : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_TIME_STAMP_DELTA_US : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_TIME_STAMP_GET : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XENV_USLEEP : [xenv_vxworks.h](#), [xenv_none.h](#), [xenv_linux.h](#)
- XFALSE : [xbasic_types.h](#)
- XFL_CFI_ADVANCE_PTR16 : [xflash_cfi.h](#)
- XFL_CFI_ADVANCE_PTR8 : [xflash_cfi.h](#)
- XFL_CFI_POSITION_PTR : [xflash_cfi.h](#)
- XFL_CFI_READ16 : [xflash_cfi.h](#)
- XFL_CFI_READ8 : [xflash_cfi.h](#)
- XFL_GEOMETRY_BLOCK_DIFF : [xflash_geometry.h](#)
- XFL_GEOMETRY_BLOCKS_LEFT : [xflash_geometry.h](#)
- XFL_GEOMETRY_INCREMENT : [xflash_geometry.h](#)
- XFL_GEOMETRY_IS_ABSOLUTE_VALID : [xflash_geometry.h](#)
- XFL_GEOMETRY_IS_BLOCK_VALID : [xflash_geometry.h](#)

- XFL_MANUFACTURER_ID_INTEL : [xflash.h](#)
- XFL_MAX_ERASE_REGIONS : [xflash_geometry.h](#)
- XFL_NON_BLOCKING_ERASE_OPTION : [xflash.h](#)
- XFL_NON_BLOCKING_WRITE_OPTION : [xflash.h](#)
- XFlash : [xflash.h](#)
- XFlash_ConfigTable : [xflash_g.c](#)
- XFlash_DeviceControl() : [xflash.h](#), [xflash.c](#)
- XFlash_Erase() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseBlockResume() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseBlockSuspend() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseChip() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseResume() : [xflash.h](#), [xflash.c](#)
- XFlash_EraseSuspend() : [xflash.h](#), [xflash.c](#)
- XFlash_GetBlockStatus() : [xflash.h](#), [xflash.c](#)
- XFlash_GetGeometry() : [xflash.h](#), [xflash.c](#)
- XFlash_GetOptions() : [xflash.h](#), [xflash.c](#)
- XFlash_GetProperties() : [xflash.h](#), [xflash.c](#)
- XFlash_GetStatus() : [xflash.h](#), [xflash.c](#)
- XFlash_Initialize() : [xflash.h](#), [xflash.c](#)
- XFlash_IsReady() : [xflash.c](#)
- XFlash_Lock() : [xflash.h](#), [xflash.c](#)
- XFlash_LockBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_LookupConfig() : [xflash.h](#), [xflash.c](#)
- XFlash_Read() : [xflash.h](#), [xflash.c](#)
- XFlash_ReadBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_Reset() : [xflash.h](#), [xflash.c](#)
- XFlash_SelfTest() : [xflash.h](#), [xflash.c](#)
- XFlash_SetOptions() : [xflash.h](#), [xflash.c](#)
- XFlash_Unlock() : [xflash.h](#), [xflash.c](#)
- XFlash_UnlockBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_Write() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteBlock() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteBlockResume() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteBlockSuspend() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteResume() : [xflash.h](#), [xflash.c](#)
- XFlash_WriteSuspend() : [xflash.h](#), [xflash.c](#)
- XFlashCFI_ReadCommon() : [xflash_cfi.h](#), [xflash_cfi.c](#)

- XFlashGeometry_ConvertLayout() : [xflash_geometry.h](#), [xflash_geometry.c](#)
- XFlashGeometry_ToAbsolute() : [xflash_geometry.h](#), [xflash_geometry.c](#)
- XFlashGeometry_ToBlock() : [xflash_geometry.h](#), [xflash_geometry.c](#)
- XFlashIntel_DeviceControl() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_Erase() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_EraseBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseBlockResume() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseBlockSuspend() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseChip() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseResume() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_EraseSuspend() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetBlockStatus() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetGeometry() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetOptions() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetProperties() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_GetStatus() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_Initialize() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_Lock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_LockAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_LockBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_mSendCmd : [xflash_intel_l.h](#)
- XFlashIntel_Read() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_ReadAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_ReadBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_Reset() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_SelfTest() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_SetOptions() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_Unlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_UnlockAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_UnlockBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WaitReady() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_Write() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WriteAddr() : [xflash_intel_l.h](#), [xflash_intel_l.c](#)
- XFlashIntel_WriteBlock() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WriteBlockResume() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WriteBlockSuspend() : [xflash_intel.h](#), [xflash_intel.c](#)
- XFlashIntel_WriteResume() : [xflash_intel.h](#), [xflash_intel.c](#)

- XFlashIntel_WriteSuspend() : [xflash_intel.h](#), [xflash_intel.c](#)
- Xfloat32 : [xbasic_types.h](#)
- Xfloat64 : [xbasic_types.h](#)
- XGE_AUTO_NEGOTIATE_OPTION : [xgemac.h](#)
- XGE_BROADCAST_OPTION : [xgemac.h](#)
- XGE_FDUPLEX_OPTION : [xgemac.h](#)
- XGE_FLOW_CONTROL_OPTION : [xgemac.h](#)
- XGE_INSERT_FCS_OPTION : [xgemac.h](#)
- XGE_INSERT_PAD_OPTION : [xgemac.h](#)
- XGE_LOOPBACK_OPTION : [xgemac.h](#)
- XGE_POLLED_OPTION : [xgemac.h](#)
- XGE_PROMISC_OPTION : [xgemac.h](#)
- XGE_STRIP_PAD_FCS_OPTION : [xgemac.h](#)
- XGE_UNICAST_OPTION : [xgemac.h](#)
- XGEmac_ClearStats() : [xgemac_stats.c](#), [xgemac.h](#)
- XGEmac_ConfigTable : [xgemac_i.h](#), [xgemac_g.c](#)
- XGEmac_ErrorHandler : [xgemac.h](#)
- XGEmac_FifoHandler : [xgemac.h](#)
- XGEmac_FifoRecv() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGEmac_FifoSend() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGEmac_GetInterframeGap() : [xgemac_options.c](#), [xgemac.h](#)
- XGEmac_GetMacAddress() : [xgemac.h](#), [xgemac.c](#)
- XGEmac_GetOptions() : [xgemac_options.c](#), [xgemac.h](#)
- XGEmac_GetPktThreshold() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGEmac_GetPktWaitBound() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGEmac_GetStats() : [xgemac_stats.c](#), [xgemac.h](#)
- XGEmac_Initialize() : [xgemac.h](#), [xgemac.c](#)
- XGEmac_IntrHandlerDma() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGEmac_IntrHandlerFifo() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGEmac_LookupConfig() : [xgemac.h](#), [xgemac.c](#)
- XGEmac_mDisable : [xgemac_l.h](#)
- XGEmac_mEnable : [xgemac_l.h](#)
- XGEmac_mIsDma : [xgemac.h](#)
- XGEmac_mIsRxEmpty : [xgemac_l.h](#)
- XGEmac_mIsSgDma : [xgemac.h](#)
- XGEmac_mIsSimpleDma : [xgemac.h](#)
- XGEmac_mIsTxDone : [xgemac_l.h](#)
- XGEmac_mPhyReset : [xgemac_l.h](#)

- XGmac_mReadReg : [xgemac_1.h](#)
- XGmac_mSetControlReg : [xgemac_1.h](#)
- XGmac_mSetMacAddress : [xgemac_1.h](#)
- XGmac_mWriteReg : [xgemac_1.h](#)
- XGmac_PollRecv() : [xgemac_polled.c](#), [xgemac.h](#)
- XGmac_PollSend() : [xgemac_polled.c](#), [xgemac.h](#)
- XGmac_Reset() : [xgemac.h](#), [xgemac.c](#)
- XGmac_SelfTest() : [xgemac_selftest.c](#), [xgemac.h](#)
- XGmac_SetErrorHandler() : [xgemac_intr.c](#), [xgemac.h](#)
- XGmac_SetFifoRecvHandler() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGmac_SetFifoSendHandler() : [xgemac_intr_fifo.c](#), [xgemac.h](#)
- XGmac_SetInterframeGap() : [xgemac_options.c](#), [xgemac.h](#)
- XGmac_SetMacAddress() : [xgemac.h](#), [xgemac.c](#)
- XGmac_SetOptions() : [xgemac_options.c](#), [xgemac.h](#)
- XGmac_SetPktThreshold() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGmac_SetPktWaitBound() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGmac_SetSgRecvHandler() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGmac_SetSgRecvSpace() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGmac_SetSgSendHandler() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGmac_SetSgSendSpace() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGmac_SgHandler : [xgemac.h](#)
- XGmac_SgRecv() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGmac_SgSend() : [xgemac_intr_dma.c](#), [xgemac.h](#)
- XGmac_Start() : [xgemac.h](#), [xgemac.c](#)
- XGmac_Stop() : [xgemac.h](#), [xgemac.c](#)
- XGpio_ConfigTable : [xgpio_i.h](#), [xgpio_g.c](#)
- XGPIO_DATA_OFFSET : [xgpio_1.h](#)
- XGpio_DiscreteClear() : [xgpio_extra.c](#), [xgpio.h](#)
- XGpio_DiscreteRead() : [xgpio.h](#), [xgpio.c](#)
- XGpio_DiscreteSet() : [xgpio_extra.c](#), [xgpio.h](#)
- XGpio_DiscreteWrite() : [xgpio.h](#), [xgpio.c](#)
- XGpio_Initialize() : [xgpio.h](#), [xgpio.c](#)
- XGpio_LookupConfig() : [xgpio.h](#), [xgpio.c](#)
- XGpio_mGetDataReg : [xgpio_1.h](#)
- XGpio_mReadReg : [xgpio_1.h](#)
- XGpio_mSetDataReg : [xgpio_1.h](#)
- XGpio_mWriteReg : [xgpio_1.h](#)
- XGpio_SelfTest() : [xgpio_selftest.c](#), [xgpio.h](#)

- XGpio_SetDataDirection() : [xgpio.h](#), [xgpio.c](#)
- XGPIO_TRI_OFFSET : [xgpio_1.h](#)
- XHD_OPTION_CRC_32 : [xhdlc.h](#)
- XHD_OPTION_CRC_DISABLE : [xhdlc.h](#)
- XHD_OPTION_LOOPBACK : [xhdlc.h](#)
- XHD_OPTION_POLLED : [xhdlc.h](#)
- XHD_OPTION_RX_16_ADDR : [xhdlc.h](#)
- XHD_OPTION_RX_BROADCAST : [xhdlc.h](#)
- XHD_OPTION_RX_FILTER_ADDR : [xhdlc.h](#)
- XHD_OPTION_RX_REMOVE_ADDR : [xhdlc.h](#)
- XHdlc_ClearStats() : [xhdlc_stats.c](#), [xhdlc.h](#)
- XHdlc_ConfigTable : [xhdlc_i.h](#), [xhdlc_g.c](#)
- XHdlc_ErrorHandler : [xhdlc.h](#)
- XHdlc_GetAddress() : [xhdlc_options.c](#), [xhdlc.h](#)
- XHdlc_GetOptions() : [xhdlc_options.c](#), [xhdlc.h](#)
- XHdlc_GetStats() : [xhdlc_stats.c](#), [xhdlc.h](#)
- XHdlc_Initialize() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_InterruptHandler() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_LookupConfig() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_mDisable : [xhdlc_1.h](#)
- XHdlc_mEnable : [xhdlc_1.h](#)
- XHdlc_mReadReg : [xhdlc_1.h](#)
- XHdlc_mSetAddressReg : [xhdlc_1.h](#)
- XHdlc_mSetRxControlReg : [xhdlc_1.h](#)
- XHdlc_mSetTxControlReg : [xhdlc_1.h](#)
- XHdlc_mWriteReg : [xhdlc_1.h](#)
- XHdlc_Recv() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_RecvFrame() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_Reset() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_SelfTest() : [xhdlc_selftest.c](#), [xhdlc.h](#)
- XHdlc_Send() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_SendFrame() : [xhdlc_1.h](#), [xhdlc_1.c](#)
- XHdlc_SetAddress() : [xhdlc_options.c](#), [xhdlc.h](#)
- XHdlc_SetErrorHandler() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SetOptions() : [xhdlc_options.c](#), [xhdlc.h](#)
- XHdlc_SetSgRecvHandler() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SetSgRecvSpace() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SetSgSendHandler() : [xhdlc_dmasg.c](#), [xhdlc.h](#)

- XHdlc_SetSgSendSpace() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SgGetRecvFrame() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SgGetSendFrame() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SgHandler : [xhdlc.h](#)
- XHdlc_SgRecv() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_SgSend() : [xhdlc_dmasg.c](#), [xhdlc.h](#)
- XHdlc_Start() : [xhdlc.h](#), [xhdlc.c](#)
- XHdlc_Stop() : [xhdlc.h](#), [xhdlc.c](#)
- XII_ADDR_TO_RESPOND_TYPE : [xiic.h](#)
- XII_ADDR_TO_SEND_TYPE : [xiic.h](#)
- XII_ARB_LOST_EVENT : [xiic.h](#)
- XII_BUS_NOT_BUSY_EVENT : [xiic.h](#)
- XII_GENERAL_CALL_EVENT : [xiic.h](#)
- XII_GENERAL_CALL_OPTION : [xiic.h](#)
- XII_MASTER_READ_EVENT : [xiic.h](#)
- XII_MASTER_WRITE_EVENT : [xiic.h](#)
- XII_REPEATED_START_OPTION : [xiic.h](#)
- XII_SEND_10_BIT_OPTION : [xiic.h](#)
- XII_SLAVE_NO_ACK_EVENT : [xiic.h](#)
- XIic_ClearStats() : [xiic_stats.c](#), [xiic.h](#)
- XIic_ConfigTable : [xiic_i.h](#), [xiic_g.c](#)
- XIic_GetAddress() : [xiic.h](#), [xiic.c](#)
- XIic_GetOptions() : [xiic_options.c](#), [xiic.h](#)
- XIic_GetStats() : [xiic_stats.c](#), [xiic.h](#)
- XIic_Handler : [xiic.h](#)
- XIic_Initialize() : [xiic.h](#), [xiic.c](#)
- XIic_InterruptHandler() : [xiic_intr.c](#), [xiic.h](#)
- XIic_IsSlave() : [xiic.c](#)
- XIic_LookupConfig() : [xiic.h](#), [xiic.c](#)
- XIic_MasterRecv() : [xiic_master.c](#), [xiic.h](#)
- XIic_MasterSend() : [xiic_master.c](#), [xiic.h](#)
- XIic_MultiMasterInclude() : [xiic_multi_master.c](#), [xiic.h](#)
- XIic_Recv() : [xiic_l.h](#), [xiic_l.c](#)
- XIic_Reset() : [xiic.h](#), [xiic.c](#)
- XIic_SelfTest() : [xiic_selftest.c](#), [xiic.h](#)
- XIic_Send() : [xiic_l.h](#), [xiic_l.c](#)
- XIic_SetAddress() : [xiic.h](#), [xiic.c](#)
- XIic_SetOptions() : [xiic_options.c](#), [xiic.h](#)

- XIic_SetRecvHandler() : [xiic.h](#), [xiic.c](#)
- XIic_SetSendHandler() : [xiic.h](#), [xiic.c](#)
- XIic_SetStatusHandler() : [xiic.h](#), [xiic.c](#)
- XIic_Start() : [xiic.h](#), [xiic.c](#)
- XIic_StatusHandler : [xiic.h](#)
- XIic_Stop() : [xiic.h](#), [xiic.c](#)
- XIN_REAL_MODE : [xintc.h](#)
- XIN_SIMULATION_MODE : [xintc.h](#)
- XIN_SVC_ALL_ISRS_OPTION : [xintc.h](#)
- XIN_SVC_SGL_ISR_OPTION : [xintc.h](#)
- Xint16 : [xbasic_types.h](#)
- Xint32 : [xbasic_types.h](#)
- Xint8 : [xbasic_types.h](#)
- XIntc_Acknowledge() : [xintc.h](#), [xintc.c](#)
- XIntc_ConfigTable : [xintc_i.h](#), [xintc_g.c](#)
- XIntc_Connect() : [xintc.h](#), [xintc.c](#)
- XIntc_DefaultHandler() : [xintc_l.h](#), [xintc_l.c](#)
- XIntc_Disable() : [xintc.h](#), [xintc.c](#)
- XIntc_Disconnect() : [xintc.h](#), [xintc.c](#)
- XIntc_Enable() : [xintc.h](#), [xintc.c](#)
- XIntc_GetOptions() : [xintc_options.c](#), [xintc.h](#)
- XIntc_Initialize() : [xintc.h](#), [xintc.c](#)
- XIntc_InterruptHandler() : [xintc_intr.c](#), [xintc.h](#)
- XIntc_LookupConfig() : [xintc.h](#), [xintc.c](#)
- XIntc_LowLevelInterruptHandler() : [xintc_l.h](#), [xintc_l.c](#)
- XIntc_mAckIntr : [xintc_l.h](#)
- XIntc_mDisableIntr : [xintc_l.h](#)
- XIntc_mEnableIntr : [xintc_l.h](#)
- XIntc_mGetIntrStatus : [xintc_l.h](#)
- XIntc_mMasterDisable : [xintc_l.h](#)
- XIntc_mMasterEnable : [xintc_l.h](#)
- XIntc_SelfTest() : [xintc_selftest.c](#), [xintc.h](#)
- XIntc_SetOptions() : [xintc_options.c](#), [xintc.h](#)
- XIntc_SimulateIntr() : [xintc_selftest.c](#), [xintc.h](#)
- XIntc_Start() : [xintc.h](#), [xintc.c](#)
- XIntc_Stop() : [xintc.h](#), [xintc.c](#)
- XIntc_VoidInterruptHandler() : [xintc_intr.c](#), [xintc.h](#)
- XInterruptHandler : [xbasic_types.h](#)

- XIo_Address : [xio.h](#)
- XIo_DcrIn() : [xio_dcr.h](#), [xio_dcr.c](#)
- XIo_DcrOut() : [xio_dcr.h](#), [xio_dcr.c](#)
- XIo_EndianSwap16() : [xio.h](#), [xio.c](#)
- XIo_EndianSwap16OLD() : [xio.c](#)
- XIo_EndianSwap32() : [xio.h](#), [xio.c](#)
- XIo_EndianSwap32OLD() : [xio.c](#)
- XIo_In16 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_In32 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_In8 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_InSwap16() : [xio.h](#), [xio.c](#)
- XIo_InSwap32() : [xio.h](#), [xio.c](#)
- XIo_Out16 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_Out32 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_Out8 : [xio.h](#), [xio.c](#), [xio.h](#)
- XIo_OutSwap16() : [xio.h](#), [xio.c](#)
- XIo_OutSwap32() : [xio.h](#), [xio.c](#)
- XNULL : [xbasic_types.h](#)
- XO2P_NO_ERROR : [xopb2plb.h](#)
- XO2P_READ_ERROR : [xopb2plb.h](#)
- XO2P_WRITE_ERROR : [xopb2plb.h](#)
- XOA_DYNAMIC_PRIORITY_OPTION : [xopbarb.h](#)
- XOA_PARK_BY_ID_OPTION : [xopbarb.h](#)
- XOA_PARK_ENABLE_OPTION : [xopbarb.h](#)
- XOpb2Plb_ClearErrors() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_ConfigTable : [xopb2plb_i.h](#), [xopb2plb_g.c](#)
- XOpb2Plb_DisableInterrupt() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_DisableLock() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_EnableInterrupt() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_EnableLock() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_GetErrorAddress() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_GetErrorStatus() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_Initialize() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_IsError() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_LookupConfig() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_mGetBearReg : [xopb2plb_l.h](#)
- XOpb2Plb_mGetBesrReg : [xopb2plb_l.h](#)
- XOpb2Plb_mGetControlReg : [xopb2plb_l.h](#)

- XOpb2Plb_mSetControlReg : [xopb2plb_1.h](#)
- XOpb2Plb_Reset() : [xopb2plb.h](#), [xopb2plb.c](#)
- XOpb2Plb_SelfTest() : [xopb2plb_selftest.c](#), [xopb2plb.h](#)
- XOpbArb_ConfigTable : [xopbarb_g.c](#)
- XOpbArb_GetOptions() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_GetParkId() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_GetPriorityLevel() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_Initialize() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_LookupConfig() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_mClearParkMasterNot : [xopbarb_1.h](#)
- XOpbArb_mClearPriorityRegsValid : [xopbarb_1.h](#)
- XOpbArb_mDisableDynamic : [xopbarb_1.h](#)
- XOpbArb_mDisableParking : [xopbarb_1.h](#)
- XOpbArb_mEnableDynamic : [xopbarb_1.h](#)
- XOpbArb_mEnableParking : [xopbarb_1.h](#)
- XOpbArb_mGetControlReg : [xopbarb_1.h](#)
- XOpbArb_mGetPriorityReg : [xopbarb_1.h](#)
- XOpbArb_mSetControlReg : [xopbarb_1.h](#)
- XOpbArb_mSetParkedMasterId : [xopbarb_1.h](#)
- XOpbArb_mSetParkMasterNot : [xopbarb_1.h](#)
- XOpbArb_mSetPriorityReg : [xopbarb_1.h](#)
- XOpbArb_mSetPriorityRegsValid : [xopbarb_1.h](#)
- XOpbArb_ResumePriorityLevels() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SelfTest() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SetOptions() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SetParkId() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SetPriorityLevel() : [xopbarb.h](#), [xopbarb.c](#)
- XOpbArb_SuspendPriorityLevels() : [xopbarb.h](#), [xopbarb.c](#)
- XP2O_DRIVING_BEAR_MASK : [xplb2opb.h](#)
- XP2O_ERROR_READ_MASK : [xplb2opb.h](#)
- XP2O_ERROR_TYPE_MASK : [xplb2opb.h](#)
- XP2O_LOCK_ERR_MASK : [xplb2opb.h](#)
- XPCI_AckRead() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_AckSend() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_BUSNO_BUS_MASK : [xpci_1.h](#)
- XPCI_BUSNO_OFFSET : [xpci_1.h](#)
- XPCI_BUSNO_SUBBUS_MASK : [xpci_1.h](#)
- XPCI_CAR_OFFSET : [xpci_1.h](#)

- XPCI_CDR_OFFSET : [xpci_1.h](#)
- XPCI_ConfigIn() : [xpci.h](#)
- XPCI_ConfigIn16() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigIn32() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigIn8() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigOut() : [xpci.h](#)
- XPCI_ConfigOut16() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigOut32() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigOut8() : [xpci_config.c](#), [xpci.h](#)
- XPCI_ConfigPack() : [xpci.h](#)
- XPCI_DMA_TYPE_NONE : [xpci_1.h](#)
- XPCI_DMA_TYPE_SG : [xpci_1.h](#)
- XPCI_DMA_TYPE_SIMPLE : [xpci_1.h](#)
- XPCI_ErrorClear() : [xpci.h](#)
- XPCI_ErrorGet() : [xpci.h](#)
- XPCI_GetBusNumber() : [xpci.h](#)
- XPCI_GetDmaImplementation() : [xpci.h](#)
- XPCI_HDR_BAR0 : [xpci_1.h](#)
- XPCI_HDR_BAR1 : [xpci_1.h](#)
- XPCI_HDR_BAR2 : [xpci_1.h](#)
- XPCI_HDR_BAR3 : [xpci_1.h](#)
- XPCI_HDR_BAR4 : [xpci_1.h](#)
- XPCI_HDR_BAR5 : [xpci_1.h](#)
- XPCI_HDR_BAR_ADDR_MASK : [xpci_1.h](#)
- XPCI_HDR_BAR_PREFETCH_NO : [xpci_1.h](#)
- XPCI_HDR_BAR_PREFETCH_YES : [xpci_1.h](#)
- XPCI_HDR_BAR_SPACE_IO : [xpci_1.h](#)
- XPCI_HDR_BAR_SPACE_MEMORY : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_ABOVE_4GB : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_BELOW_1MB : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_BELOW_4GB : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_MASK : [xpci_1.h](#)
- XPCI_HDR_BAR_TYPE_RESERVED : [xpci_1.h](#)
- XPCI_HDR_BIST : [xpci_1.h](#)
- XPCI_HDR_CACHE_LINE_SZ : [xpci_1.h](#)
- XPCI_HDR_CAP_PTR : [xpci_1.h](#)
- XPCI_HDR_CARDBUS_PTR : [xpci_1.h](#)
- XPCI_HDR_CLASSCODE : [xpci_1.h](#)

- XPCI_HDR_COMMAND : [xpci_1.h](#)
- XPCI_HDR_DEVICE : [xpci_1.h](#)
- XPCI_HDR_INT_LINE : [xpci_1.h](#)
- XPCI_HDR_INT_PIN : [xpci_1.h](#)
- XPCI_HDR_LATENCY : [xpci_1.h](#)
- XPCI_HDR_MAX_LAT : [xpci_1.h](#)
- XPCI_HDR_MIN_GNT : [xpci_1.h](#)
- XPCI_HDR_NUM_BAR : [xpci_1.h](#)
- XPCI_HDR_REVID : [xpci_1.h](#)
- XPCI_HDR_ROM_BASE : [xpci_1.h](#)
- XPCI_HDR_STATUS : [xpci_1.h](#)
- XPCI_HDR_SUB_DEVICE : [xpci_1.h](#)
- XPCI_HDR_SUB_VENDOR : [xpci_1.h](#)
- XPCI_HDR_TYPE : [xpci_1.h](#)
- XPCI_HDR_VENDOR : [xpci_1.h](#)
- XPCI_IAR_OFFSET : [xpci_1.h](#)
- XPCI_INHIBIT_LOCAL_BUS_R : [xpci_1.h](#)
- XPCI_INHIBIT_LOCAL_BUS_W : [xpci_1.h](#)
- XPCI_INHIBIT_MASK : [xpci_1.h](#)
- XPCI_INHIBIT_OFFSET : [xpci_1.h](#)
- XPCI_INHIBIT_PCI_R : [xpci_1.h](#)
- XPCI_INHIBIT_PCI_W : [xpci_1.h](#)
- XPci_InhibitAfterError() : [xpci.h](#)
- XPci_Initialize() : [xpci.h](#)
- XPci_InterruptClear() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptDisable() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptEnable() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptGetEnabled() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptGetHighestPending() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptGetPending() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptGetStatus() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptGlobalDisable() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptGlobalEnable() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptPciClear() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptPciDisable() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptPciEnable() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptPciGetEnabled() : [xpci_intr.c](#), [xpci.h](#)
- XPci_InterruptPciGetStatus() : [xpci_intr.c](#), [xpci.h](#)

- XPCI_IR_LM_BR_W : [xpci_1.h](#)
- XPCI_IR_LM_BRANGE_W : [xpci_1.h](#)
- XPCI_IR_LM_BRD_W : [xpci_1.h](#)
- XPCI_IR_LM_BRT_W : [xpci_1.h](#)
- XPCI_IR_LM_MA_W : [xpci_1.h](#)
- XPCI_IR_LM_PERR_R : [xpci_1.h](#)
- XPCI_IR_LM_PERR_W : [xpci_1.h](#)
- XPCI_IR_LM_SERR_R : [xpci_1.h](#)
- XPCI_IR_LM_SERR_W : [xpci_1.h](#)
- XPCI_IR_LM_TA_R : [xpci_1.h](#)
- XPCI_IR_LM_TA_W : [xpci_1.h](#)
- XPCI_IR_MASK : [xpci_1.h](#)
- XPCI_IR_PI_SERR_R : [xpci_1.h](#)
- XPCI_IR_PI_SERR_W : [xpci_1.h](#)
- XPCI_LMA_R_OFFSET : [xpci_1.h](#)
- XPCI_LMA_W_OFFSET : [xpci_1.h](#)
- XPCI_LMADDR_BR_W : [xpci_1.h](#)
- XPCI_LMADDR_BRANGE_W : [xpci_1.h](#)
- XPCI_LMADDR_BRD_W : [xpci_1.h](#)
- XPCI_LMADDR_BRT_W : [xpci_1.h](#)
- XPCI_LMADDR_MA_W : [xpci_1.h](#)
- XPCI_LMADDR_MASK : [xpci_1.h](#)
- XPCI_LMADDR_OFFSET : [xpci_1.h](#)
- XPCI_LMADDR_PERR_R : [xpci_1.h](#)
- XPCI_LMADDR_PERR_W : [xpci_1.h](#)
- XPCI_LMADDR_SERR_R : [xpci_1.h](#)
- XPCI_LMADDR_SERR_W : [xpci_1.h](#)
- XPCI_LMADDR_TA_R : [xpci_1.h](#)
- XPCI_LMADDR_TA_W : [xpci_1.h](#)
- XPci_LookupConfig() : [xpci.h](#)
- XPci_mAckRead : [xpci_1.h](#)
- XPci_mAckSend : [xpci_1.h](#)
- XPci_mConfigIn : [xpci_1.h](#)
- XPci_mConfigOut : [xpci_1.h](#)
- XPci_mIntrClear : [xpci_1.h](#)
- XPci_mIntrDisable : [xpci_1.h](#)
- XPci_mIntrEnable : [xpci_1.h](#)
- XPci_mIntrGlobalDisable : [xpci_1.h](#)

- XPCI_mIntrGlobalEnable : [xpci_1.h](#)
- XPCI_mIntrPciClear : [xpci_1.h](#)
- XPCI_mIntrPciDisable : [xpci_1.h](#)
- XPCI_mIntrPciEnable : [xpci_1.h](#)
- XPCI_mIntrPciReadIER : [xpci_1.h](#)
- XPCI_mIntrPciReadISR : [xpci_1.h](#)
- XPCI_mIntrPciWriteISR : [xpci_1.h](#)
- XPCI_mIntrReadID : [xpci_1.h](#)
- XPCI_mIntrReadIER : [xpci_1.h](#)
- XPCI_mIntrReadIPR : [xpci_1.h](#)
- XPCI_mIntrReadISR : [xpci_1.h](#)
- XPCI_mIntrWriteISR : [xpci_1.h](#)
- XPCI_mLocal2Pci : [xpci_1.h](#)
- XPCI_mPci2Local : [xpci_1.h](#)
- XPCI_mReadReg : [xpci_1.h](#)
- XPCI_mReset : [xpci_1.h](#)
- XPCI_mSpecialCycle : [xpci_1.h](#)
- XPCI_mWriteReg : [xpci_1.h](#)
- XPCI_PIA_R_OFFSET : [xpci_1.h](#)
- XPCI_PIA_W_OFFSET : [xpci_1.h](#)
- XPCI_PIADDR_ERRACK_R : [xpci_1.h](#)
- XPCI_PIADDR_ERRACK_W : [xpci_1.h](#)
- XPCI_PIADDR_MASK : [xpci_1.h](#)
- XPCI_PIADDR_OFFSET : [xpci_1.h](#)
- XPCI_PIADDR_RANGE_W : [xpci_1.h](#)
- XPCI_PIADDR_RETRY_W : [xpci_1.h](#)
- XPCI_PIADDR_TIMEOUT_W : [xpci_1.h](#)
- XPCI_PREOVRD_OFFSET : [xpci_1.h](#)
- XPCI_Reset() : [xpci.h](#)
- XPCI_SC_DATA_OFFSET : [xpci_1.h](#)
- XPCI_SelfTest() : [xpci_selftest.c](#), [xpci.h](#)
- XPCI_SERR_R_OFFSET : [xpci_1.h](#)
- XPCI_SERR_W_OFFSET : [xpci_1.h](#)
- XPCI_SetBusNumber() : [xpci.h](#)
- XPCI_SpecialCycle() : [xpci_intr.c](#), [xpci.h](#)
- XPCI_STATCMD_66MHZ_CAP : [xpci_1.h](#)
- XPCI_STATCMD_BACK_EN : [xpci_1.h](#)
- XPCI_STATCMD_BUSM_EN : [xpci_1.h](#)

- XPCI_STATCMD_DEVSEL_FAST : [xpci_1.h](#)
- XPCI_STATCMD_DEVSEL_MED : [xpci_1.h](#)
- XPCI_STATCMD_DEVSEL_MSK : [xpci_1.h](#)
- XPCI_STATCMD_ERR_MASK : [xpci_1.h](#)
- XPCI_STATCMD_INT_DISABLE : [xpci_1.h](#)
- XPCI_STATCMD_INT_STATUS : [xpci_1.h](#)
- XPCI_STATCMD_IO_EN : [xpci_1.h](#)
- XPCI_STATCMD_MEM_EN : [xpci_1.h](#)
- XPCI_STATCMD_MEMWR_INV_EN : [xpci_1.h](#)
- XPCI_STATCMD_MPERR : [xpci_1.h](#)
- XPCI_STATCMD_MSTABRT_RCV : [xpci_1.h](#)
- XPCI_STATCMD_OFFSET : [xpci_1.h](#)
- XPCI_STATCMD_PARITY : [xpci_1.h](#)
- XPCI_STATCMD_PERR_DET : [xpci_1.h](#)
- XPCI_STATCMD_SERR_EN : [xpci_1.h](#)
- XPCI_STATCMD_SERR_SIG : [xpci_1.h](#)
- XPCI_STATCMD_SPECIALCYC : [xpci_1.h](#)
- XPCI_STATCMD_STEPPING : [xpci_1.h](#)
- XPCI_STATCMD_TGTABRT_RCV : [xpci_1.h](#)
- XPCI_STATCMD_TGTABRT_SIG : [xpci_1.h](#)
- XPCI_STATCMD_VGA_SNOOP_EN : [xpci_1.h](#)
- XPCI_STATV3_DATA_XFER : [xpci_1.h](#)
- XPCI_STATV3_DISC_WDATA : [xpci_1.h](#)
- XPCI_STATV3_DISC_WODATA : [xpci_1.h](#)
- XPCI_STATV3_MASK : [xpci_1.h](#)
- XPCI_STATV3_MASTER_ABRT : [xpci_1.h](#)
- XPCI_STATV3_NORM_TERM : [xpci_1.h](#)
- XPCI_STATV3_OFFSET : [xpci_1.h](#)
- XPCI_STATV3_PCI_RETRY_R : [xpci_1.h](#)
- XPCI_STATV3_PCI_RETRY_W : [xpci_1.h](#)
- XPCI_STATV3_TGT_ABRT : [xpci_1.h](#)
- XPCI_STATV3_TGT_TERM : [xpci_1.h](#)
- XPCI_STATV3_TRANS_END : [xpci_1.h](#)
- XPCI_STATV3_WRITE_BUSY : [xpci_1.h](#)
- XPCI_V3StatusCommandGet() : [xpci_v3.c](#), [xpci.h](#)
- XPCI_V3TransactionStatusClear() : [xpci_v3.c](#), [xpci.h](#)
- XPCI_V3TransactionStatusGet() : [xpci_v3.c](#), [xpci.h](#)
- XPlb2Opb_ClearErrors() : [xplb2opb.h](#), [xplb2opb.c](#)

- XPlb2Opb_ConfigTable : [xplb2opb_i.h](#), [xplb2opb_g.c](#)
- XPlb2Opb_DisableInterrupt() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_EnableInterrupt() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_GetErrorAddress() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_GetErrorByteEnables() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_GetErrorStatus() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_GetMasterDrivingError() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_GetNumMasters() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_Initialize() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_IsError() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_LookupConfig() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_mGetByteEnableReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetControlReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetErrorAddressReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetErrorDetectReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetErrorTypeReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetLockBitReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetMasterDrivingReg : [xplb2opb_1.h](#)
- XPlb2Opb_mGetReadWriteReg : [xplb2opb_1.h](#)
- XPlb2Opb_mSetControlReg : [xplb2opb_1.h](#)
- XPlb2Opb_mSetErrorDetectReg : [xplb2opb_1.h](#)
- XPlb2Opb_Reset() : [xplb2opb.h](#), [xplb2opb.c](#)
- XPlb2Opb_SelfTest() : [xplb2opb_selftest.c](#), [xplb2opb.h](#)
- XPlbArb_ClearErrors() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_ConfigTable : [xplbarb_i.h](#), [xplbarb_g.c](#)
- XPlbArb_DisableInterrupt() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_EnableInterrupt() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_GetErrorAddress() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_GetErrorStatus() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_GetNumMasters() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_Initialize() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_IsError() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_LookupConfig() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_mDisableInterrupt : [xplbarb_1.h](#)
- XPlbArb_mEnableInterrupt : [xplbarb_1.h](#)
- XPlbArb_mGetControlReg : [xplbarb_1.h](#)
- XPlbArb_mGetPearAddrReg : [xplbarb_1.h](#)
- XPlbArb_mGetPearByteEnReg : [xplbarb_1.h](#)

- XPlbArb_mGetPesrLockReg : [xplbarb_1.h](#)
- XPlbArb_mGetPesrMDriveReg : [xplbarb_1.h](#)
- XPlbArb_mGetPesrMerrReg : [xplbarb_1.h](#)
- XPlbArb_mGetPesrRnwReg : [xplbarb_1.h](#)
- XPlbArb_mReset : [xplbarb_1.h](#)
- XPlbArb_mSetPesrMerrReg : [xplbarb_1.h](#)
- XPlbArb_Reset() : [xplbarb.h](#), [xplbarb.c](#)
- XPlbArb_SelfTest() : [xplbarb_selftest.c](#), [xplbarb.h](#)
- XPs2_ClearStats() : [xps2_stats.c](#)
- XPs2_DisableInterrupt() : [xps2_intr.c](#), [xps2.h](#)
- XPs2_EnableInterrupt() : [xps2_intr.c](#), [xps2.h](#)
- XPs2_GetLastErrors() : [xps2_options.c](#), [xps2.h](#)
- XPs2_GetStats() : [xps2_stats.c](#)
- XPs2_Initialize() : [xps2.h](#), [xps2.c](#)
- XPs2_InterruptHandler() : [xps2_intr.c](#), [xps2.h](#)
- XPs2_IsSending() : [xps2_options.c](#), [xps2.h](#)
- XPs2_LookupConfig() : [xps2.h](#), [xps2.c](#)
- XPs2_mClearIntr : [xps2_1.h](#)
- XPs2_mDisableIntr : [xps2_1.h](#)
- XPs2_mEnableIntr : [xps2_1.h](#)
- XPs2_mGetIntrStatus : [xps2_1.h](#)
- XPs2_mGetStatus : [xps2_1.h](#)
- XPs2_mIsIntrEnabled : [xps2_1.h](#)
- XPs2_mIsReceiveEmpty : [xps2_1.h](#)
- XPs2_mIsTransmitFull : [xps2_1.h](#)
- XPs2_mReset : [xps2_1.h](#)
- XPs2_ReceiveBuffer() : [xps2_i.h](#), [xps2.c](#)
- XPs2_Recv() : [xps2.h](#), [xps2.c](#)
- XPs2_RecvByte() : [xps2_1.h](#), [xps2_1.c](#)
- XPs2_Send() : [xps2.h](#), [xps2.c](#)
- XPs2_SendBuffer() : [xps2_i.h](#), [xps2.c](#)
- XPs2_SendByte() : [xps2_1.h](#), [xps2_1.c](#)
- XPs2_SetHandler() : [xps2_intr.c](#), [xps2.h](#)
- XRRapidIo_mGetLinkStatus : [xrapidio_1.h](#)
- XRRapidIo_mReadReg : [xrapidio_1.h](#)
- XRRapidIo_mReset : [xrapidio_1.h](#)
- XRRapidIo_mWriteReg : [xrapidio_1.h](#)
- XRRapidIo_RecvPkt() : [xrapidio_1.h](#), [xrapidio_1.c](#)

- X RapidIo_SendPkt() : [xrapidio_1.h](#), [xrapidio_1.c](#)
- XSA_BMR_16BIT_MASK : [xsysace_1.h](#)
- XSA_BMR_OFFSET : [xsysace_1.h](#)
- XSA_CF_SECTOR_SIZE : [xsysace_1.h](#)
- XSA_CLR_LBA_MASK : [xsysace_1.h](#)
- XSA_CLR_OFFSET : [xsysace_1.h](#)
- XSA_CR_CFGADDR_MASK : [xsysace_1.h](#)
- XSA_CR_CFGADDR_SHIFT : [xsysace_1.h](#)
- XSA_CR_CFGDONEIRQ_MASK : [xsysace_1.h](#)
- XSA_CR_CFGMODE_MASK : [xsysace_1.h](#)
- XSA_CR_CFGPROG_MASK : [xsysace_1.h](#)
- XSA_CR_CFGRESET_MASK : [xsysace_1.h](#)
- XSA_CR_CFGSEL_MASK : [xsysace_1.h](#)
- XSA_CR_CFGSTART_MASK : [xsysace_1.h](#)
- XSA_CR_DATARDYIRQ_MASK : [xsysace_1.h](#)
- XSA_CR_ERRORIRQ_MASK : [xsysace_1.h](#)
- XSA_CR_FORCECFGADDR_MASK : [xsysace_1.h](#)
- XSA_CR_FORCECFGMODE_MASK : [xsysace_1.h](#)
- XSA_CR_FORCELOCK_MASK : [xsysace_1.h](#)
- XSA_CR_LOCKREQ_MASK : [xsysace_1.h](#)
- XSA_CR_OFFSET : [xsysace_1.h](#)
- XSA_CR_RESETIRQ_MASK : [xsysace_1.h](#)
- XSA_DATA_BUFFER_SIZE : [xsysace_1.h](#)
- XSA_DBR_OFFSET : [xsysace_1.h](#)
- XSA_ER_ABORT : [xsysace_1.h](#)
- XSA_ER_BAD_BLOCK : [xsysace_1.h](#)
- XSA_ER_CARD_READ : [xsysace_1.h](#)
- XSA_ER_CARD_READY : [xsysace_1.h](#)
- XSA_ER_CARD_RESET : [xsysace_1.h](#)
- XSA_ER_CARD_WRITE : [xsysace_1.h](#)
- XSA_ER_CFG_ADDR : [xsysace_1.h](#)
- XSA_ER_CFG_FAIL : [xsysace_1.h](#)
- XSA_ER_CFG_INIT : [xsysace_1.h](#)
- XSA_ER_CFG_INSTR : [xsysace_1.h](#)
- XSA_ER_CFG_READ : [xsysace_1.h](#)
- XSA_ER_GENERAL : [xsysace_1.h](#)
- XSA_ER_OFFSET : [xsysace_1.h](#)
- XSA_ER_RESERVED : [xsysace_1.h](#)

- XSA_ER_SECTOR_ID : [xsysace_1.h](#)
- XSA_ER_SECTOR_READY : [xsysace_1.h](#)
- XSA_ER_UNCORRECTABLE : [xsysace_1.h](#)
- XSA_EVENT_CFG_DONE : [xsysace.h](#)
- XSA_EVENT_DATA_DONE : [xsysace.h](#)
- XSA_EVENT_ERROR : [xsysace.h](#)
- XSA_FAT_12_BOOT_REC : [xsysace_1.h](#)
- XSA_FAT_12_CALC : [xsysace_1.h](#)
- XSA_FAT_12_PART_REC : [xsysace_1.h](#)
- XSA_FAT_16_BOOT_REC : [xsysace_1.h](#)
- XSA_FAT_16_CALC : [xsysace_1.h](#)
- XSA_FAT_16_PART_REC : [xsysace_1.h](#)
- XSA_FAT_VALID_BOOT_REC : [xsysace_1.h](#)
- XSA_FAT_VALID_PART_REC : [xsysace_1.h](#)
- XSA_FSR_OFFSET : [xsysace_1.h](#)
- XSA_MLR_LBA_MASK : [xsysace_1.h](#)
- XSA_MLR_OFFSET : [xsysace_1.h](#)
- XSA_SCCR_ABORT_MASK : [xsysace_1.h](#)
- XSA_SCCR_CMD_MASK : [xsysace_1.h](#)
- XSA_SCCR_COUNT_MASK : [xsysace_1.h](#)
- XSA_SCCR_IDENTIFY_MASK : [xsysace_1.h](#)
- XSA_SCCR_OFFSET : [xsysace_1.h](#)
- XSA_SCCR_READDATA_MASK : [xsysace_1.h](#)
- XSA_SCCR_RESET_MASK : [xsysace_1.h](#)
- XSA_SCCR_WRITEDATA_MASK : [xsysace_1.h](#)
- XSA_SR_CFBSY_MASK : [xsysace_1.h](#)
- XSA_SR_CFCERROR_MASK : [xsysace_1.h](#)
- XSA_SR_CFCORR_MASK : [xsysace_1.h](#)
- XSA_SR_CFDTECT_MASK : [xsysace_1.h](#)
- XSA_SR_CFDREQ_MASK : [xsysace_1.h](#)
- XSA_SR_CFDSC_MASK : [xsysace_1.h](#)
- XSA_SR_CFDWF_MASK : [xsysace_1.h](#)
- XSA_SR_CFERR_MASK : [xsysace_1.h](#)
- XSA_SR_CFGADDR_MASK : [xsysace_1.h](#)
- XSA_SR_CFGDONE_MASK : [xsysace_1.h](#)
- XSA_SR_CFGERROR_MASK : [xsysace_1.h](#)
- XSA_SR_CFGLOCK_MASK : [xsysace_1.h](#)
- XSA_SR_CFGMODE_MASK : [xsysace_1.h](#)

- XSA_SR_CFRDY_MASK : [xsysace_1.h](#)
- XSA_SR_DATABUFMODE_MASK : [xsysace_1.h](#)
- XSA_SR_DATABUFRDY_MASK : [xsysace_1.h](#)
- XSA_SR_MPULOCK_MASK : [xsysace_1.h](#)
- XSA_SR_OFFSET : [xsysace_1.h](#)
- XSA_SR_RDYFORCMD_MASK : [xsysace_1.h](#)
- XSA_VR_OFFSET : [xsysace_1.h](#)
- XSP_CLK_ACTIVE_LOW_OPTION : [xspi.h](#)
- XSP_CLK_PHASE_1_OPTION : [xspi.h](#)
- XSP_LOOPBACK_OPTION : [xspi.h](#)
- XSP_MANUAL_SSELECT_OPTION : [xspi.h](#)
- XSP_MASTER_OPTION : [xspi.h](#)
- XSpi_Abort() : [xspi_i.h](#), [xspi.c](#)
- XSpi_ClearStats() : [xspi_stats.c](#), [xspi.h](#)
- XSpi_ConfigTable : [xspi_i.h](#), [xspi_g.c](#)
- XSpi_GetOptions() : [xspi_options.c](#), [xspi.h](#)
- XSpi_GetSlaveSelect() : [xspi.h](#), [xspi.c](#)
- XSpi_GetStats() : [xspi_stats.c](#), [xspi.h](#)
- XSpi_Initialize() : [xspi.h](#), [xspi.c](#)
- XSpi_InterruptHandler() : [xspi.h](#), [xspi.c](#)
- XSpi_LookupConfig() : [xspi.h](#), [xspi.c](#)
- XSpi_mDisable : [xspi_1.h](#)
- XSpi_mEnable : [xspi_1.h](#)
- XSpi_mGetControlReg : [xspi_1.h](#)
- XSpi_mGetSlaveSelectReg : [xspi_1.h](#)
- XSpi_mGetStatusReg : [xspi_1.h](#)
- XSpi_mRecvByte : [xspi_1.h](#)
- XSpi_mSendByte : [xspi_1.h](#)
- XSpi_mSetControlReg : [xspi_1.h](#)
- XSpi_mSetSlaveSelectReg : [xspi_1.h](#)
- XSpi_Reset() : [xspi.h](#), [xspi.c](#)
- XSpi_SelfTest() : [xspi_selftest.c](#), [xspi.h](#)
- XSpi_SetOptions() : [xspi_options.c](#), [xspi.h](#)
- XSpi_SetSlaveSelect() : [xspi.h](#), [xspi.c](#)
- XSpi_SetStatusHandler() : [xspi.h](#), [xspi.c](#)
- XSpi_Start() : [xspi.h](#), [xspi.c](#)
- XSpi_StatusHandler : [xspi.h](#)
- XSpi_Stop() : [xspi.h](#), [xspi.c](#)

- XSpi_Transfer() : [xspi.h](#), [xspi.c](#)
- XStatus : [xstatus.h](#)
- XSysAce_AbortCF() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_ConfigTable : [xsysace_g.c](#)
- XSysAce_DisableInterrupt() : [xsysace_intr.c](#), [xsysace.h](#)
- XSysAce_EnableInterrupt() : [xsysace_intr.c](#), [xsysace.h](#)
- XSysAce_EventHandler : [xsysace.h](#)
- XSysAce_GetCfgSector() : [xsysace_jtagcfg.c](#), [xsysace.h](#)
- XSysAce_GetErrors() : [xsysace.h](#), [xsysace.c](#)
- XSysAce_GetFatStatus() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_GetVersion() : [xsysace_selftest.c](#), [xsysace.h](#)
- XSysAce_IdentifyCF() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_Initialize() : [xsysace.h](#), [xsysace.c](#)
- XSysAce_InterruptHandler() : [xsysace_intr.c](#), [xsysace.h](#)
- XSysAce_IsCfgDone() : [xsysace_jtagcfg.c](#), [xsysace.h](#)
- XSysAce_IsCFReady() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_Lock() : [xsysace.h](#), [xsysace.c](#)
- XSysAce_LookupConfig() : [xsysace.h](#), [xsysace.c](#)
- XSysAce_mAndControlReg : [xsysace_1.h](#)
- XSysAce_mDisableIntr : [xsysace_1.h](#)
- XSysAce_mEnableIntr : [xsysace_1.h](#)
- XSysAce_mGetControlReg : [xsysace_1.h](#)
- XSysAce_mGetErrorReg : [xsysace_1.h](#)
- XSysAce_mGetStatusReg : [xsysace_1.h](#)
- XSysAce_mIsCfgDone : [xsysace_1.h](#)
- XSysAce_mIsIntrEnabled : [xsysace_1.h](#)
- XSysAce_mIsMpuLocked : [xsysace_1.h](#)
- XSysAce_mIsReadyForCmd : [xsysace_1.h](#)
- XSysAce_mOrControlReg : [xsysace_1.h](#)
- XSysAce_mSetCfgAddr : [xsysace_1.h](#)
- XSysAce_mSetControlReg : [xsysace_1.h](#)
- XSysAce_mWaitForLock : [xsysace_1.h](#)
- XSysAce_ProgramChain() : [xsysace_jtagcfg.c](#), [xsysace.h](#)
- XSysAce_ReadDataBuffer() : [xsysace_1.h](#), [xsysace_1.c](#)
- XSysAce_ReadSector() : [xsysace_1.h](#), [xsysace_1.c](#)
- XSysAce_RegRead16() : [xsysace_1.h](#), [xsysace_1.c](#)
- XSysAce_RegRead32() : [xsysace_1.h](#), [xsysace_1.c](#)
- XSysAce_RegWrite16() : [xsysace_1.h](#), [xsysace_1.c](#)

- XSysAce_RegWrite32() : [xsysace_l.h](#), [xsysace_l.c](#)
- XSysAce_ResetCF() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_ResetCfg() : [xsysace_jtagcfg.c](#), [xsysace.h](#)
- XSysAce_SectorRead() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_SectorWrite() : [xsysace_compactflash.c](#), [xsysace.h](#)
- XSysAce_SelfTest() : [xsysace_selftest.c](#), [xsysace.h](#)
- XSysAce_SetCfgAddr() : [xsysace_jtagcfg.c](#), [xsysace.h](#)
- XSysAce_SetEventHandler() : [xsysace_intr.c](#), [xsysace.h](#)
- XSysAce_SetStartMode() : [xsysace_jtagcfg.c](#), [xsysace.h](#)
- XSysAce_Unlock() : [xsysace.h](#), [xsysace.c](#)
- XSysAce_WriteDataBuffer() : [xsysace_l.h](#), [xsysace_l.c](#)
- XSysAce_WriteSector() : [xsysace_l.h](#), [xsysace_l.c](#)
- XTC_AUTO_RELOAD_OPTION : [xtmrctr.h](#)
- XTC_CAPTURE_MODE_OPTION : [xtmrctr.h](#)
- XTC_DEVICE_TIMER_COUNT : [xtmrctr_l.h](#)
- XTC_DOWN_COUNT_OPTION : [xtmrctr.h](#)
- XTC_ENABLE_ALL_OPTION : [xtmrctr.h](#)
- XTC_EXT_COMPARE_OPTION : [xtmrctr.h](#)
- XTC_INT_MODE_OPTION : [xtmrctr.h](#)
- XTimerCtr_mReadReg : [xtmrctr_l.h](#)
- XTmrCtr_ClearStats() : [xtmrctr_stats.c](#), [xtmrctr.h](#)
- XTmrCtr_ConfigTable : [xtmrctr_i.h](#), [xtmrctr_g.c](#)
- XTmrCtr_GetCaptureValue() : [xtmrctr.h](#), [xtmrctr.c](#)
- XTmrCtr_GetOptions() : [xtmrctr_options.c](#), [xtmrctr.h](#)
- XTmrCtr_GetStats() : [xtmrctr_stats.c](#), [xtmrctr.h](#)
- XTmrCtr_GetValue() : [xtmrctr.h](#), [xtmrctr.c](#)
- XTmrCtr_Handler : [xtmrctr.h](#)
- XTmrCtr_Initialize() : [xtmrctr.h](#), [xtmrctr.c](#)
- XTmrCtr_InterruptHandler() : [xtmrctr_intr.c](#), [xtmrctr.h](#)
- XTmrCtr_IsExpired() : [xtmrctr.h](#), [xtmrctr.c](#)
- XTmrCtr_mDisable : [xtmrctr_l.h](#)
- XTmrCtr_mDisableIntr : [xtmrctr_l.h](#)
- XTmrCtr_mEnable : [xtmrctr_l.h](#)
- XTmrCtr_mEnableIntr : [xtmrctr_l.h](#)
- XTmrCtr_mGetControlStatusReg : [xtmrctr_l.h](#)
- XTmrCtr_mGetLoadReg : [xtmrctr_l.h](#)
- XTmrCtr_mGetTimerCounterReg : [xtmrctr_l.h](#)
- XTmrCtr_mHasEventOccurred : [xtmrctr_l.h](#)

- XTmrCtr_mLoadTimerCounterReg : [xtmrctr_1.h](#)
- XTmrCtr_mSetControlStatusReg : [xtmrctr_1.h](#)
- XTmrCtr_mSetLoadReg : [xtmrctr_1.h](#)
- XTmrCtr_mWriteReg : [xtmrctr_1.h](#)
- XTmrCtr_Reset() : [xtmrctr.h](#), [xtmrctr.c](#)
- XTmrCtr_SelfTest() : [xtmrctr_selftest.c](#), [xtmrctr.h](#)
- XTmrCtr_SetHandler() : [xtmrctr_intr.c](#), [xtmrctr.h](#)
- XTmrCtr_SetOptions() : [xtmrctr_options.c](#), [xtmrctr.h](#)
- XTmrCtr_SetResetValue() : [xtmrctr.h](#), [xtmrctr.c](#)
- XTmrCtr_Start() : [xtmrctr.h](#), [xtmrctr.c](#)
- XTmrCtr_Stop() : [xtmrctr.h](#), [xtmrctr.c](#)
- XTouchscreen_GetPosition_2D() : [xtouchscreen.h](#), [xtouchscreen.c](#)
- XTouchscreen_GetPosition_3D() : [xtouchscreen.h](#), [xtouchscreen.c](#)
- XTouchscreen_GetValue() : [xtouchscreen_1.h](#), [xtouchscreen_1.c](#)
- XTouchscreen_Initialize() : [xtouchscreen.h](#), [xtouchscreen.c](#)
- XTouchscreen_InterruptHandler() : [xtouchscreen_intr.c](#), [xtouchscreen.h](#)
- XTouchscreen_LookupConfig() : [xtouchscreen.h](#), [xtouchscreen.c](#)
- XTouchscreen_mClearIntr : [xtouchscreen_1.h](#)
- XTouchscreen_mGetIntrStatus : [xtouchscreen_1.h](#)
- XTouchscreen_mReadCtrlReg : [xtouchscreen_1.h](#)
- XTouchscreen_mWriteCtrlReg : [xtouchscreen_1.h](#)
- XTouchscreen_SetHandler() : [xtouchscreen_intr.c](#), [xtouchscreen.h](#)
- XTRUE : [xbasic_types.h](#)
- XUartLite_ClearStats() : [xuartlite_stats.c](#), [xuartlite.h](#)
- XUartLite_ConfigTable : [xuartlite_i.h](#), [xuartlite_g.c](#)
- XUartLite_DisableInterrupt() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartLite_EnableInterrupt() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartLite_GetStats() : [xuartlite_stats.c](#), [xuartlite.h](#)
- XUartLite_Handler : [xuartlite.h](#)
- XUartLite_Initialize() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_InterruptHandler() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartLite_IsSending() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_mDisableIntr : [xuartlite_1.h](#)
- XUartLite_mEnableIntr : [xuartlite_1.h](#)
- XUartLite_mGetControlReg : [xuartlite_1.h](#)
- XUartLite_mGetStatusReg : [xuartlite_1.h](#)
- XUartLite_mIsIntrEnabled : [xuartlite_1.h](#)
- XUartLite_mIsReceiveEmpty : [xuartlite_1.h](#)

- XUartLite_mIsTransmitFull : [xuartlite_1.h](#)
- XUartLite_mSetControlReg : [xuartlite_1.h](#)
- XUartLite_ReceiveBuffer() : [xuartlite_i.h](#), [xuartlite.c](#)
- XUartLite_Recv() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_RecvByte() : [xuartlite_1.h](#), [xuartlite_1.c](#)
- XUartLite_ResetFifos() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_SelfTest() : [xuartlite_selftest.c](#), [xuartlite.h](#)
- XUartLite_Send() : [xuartlite.h](#), [xuartlite.c](#)
- XUartLite_SendBuffer() : [xuartlite_i.h](#), [xuartlite.c](#)
- XUartLite_SendByte() : [xuartlite_1.h](#), [xuartlite_1.c](#)
- XUartLite_SetRecvHandler() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartLite_SetSendHandler() : [xuartlite_intr.c](#), [xuartlite.h](#)
- XUartNs550_ClearStats() : [xuartns550_stats.c](#), [xuartns550.h](#)
- XUartNs550_ConfigTable : [xuartns550_i.h](#), [xuartns550_g.c](#)
- XUartNs550_GetDataFormat() : [xuartns550_format.c](#), [xuartns550.h](#)
- XUartNs550_GetFifoThreshold() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_GetLastErrors() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_GetModemStatus() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_GetOptions() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_GetStats() : [xuartns550_stats.c](#), [xuartns550.h](#)
- XUartNs550_Handler : [xuartns550.h](#)
- XUartNs550_Initialize() : [xuartns550.h](#), [xuartns550.c](#)
- XUartNs550_InterruptHandler() : [xuartns550_intr.c](#), [xuartns550.h](#)
- XUartNs550_IsSending() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_LookupConfig() : [xuartns550.h](#), [xuartns550.c](#)
- XUartNs550_mDisableIntr : [xuartns550_1.h](#)
- XUartNs550_mEnableIntr : [xuartns550_1.h](#)
- XUartNs550_mGetLineControlReg : [xuartns550_1.h](#)
- XUartNs550_mGetLineStatusReg : [xuartns550_1.h](#)
- XUartNs550_mIsReceiveData : [xuartns550_1.h](#)
- XUartNs550_mIsTransmitEmpty : [xuartns550_1.h](#)
- XUartNs550_mReadReg : [xuartns550_1.h](#)
- XUartNs550_mSetLineControlReg : [xuartns550_1.h](#)
- XUartNs550_mWriteReg : [xuartns550_1.h](#)
- XUartNs550_ReceiveBuffer() : [xuartns550_i.h](#), [xuartns550.c](#)
- XUartNs550_Recv() : [xuartns550.h](#), [xuartns550.c](#)
- XUartNs550_RecvByte() : [xuartns550_1.h](#), [xuartns550_1.c](#)
- XUartNs550_SelfTest() : [xuartns550_selftest.c](#), [xuartns550.h](#)

- XUartNs550_Send() : [xuartns550.h](#), [xuartns550.c](#)
- XUartNs550_SendBuffer() : [xuartns550_i.h](#), [xuartns550.c](#)
- XUartNs550_SendByte() : [xuartns550_l.h](#), [xuartns550_l.c](#)
- XUartNs550_SetDataFormat() : [xuartns550_format.c](#), [xuartns550.h](#)
- XUartNs550_SetFifoThreshold() : [xuartns550_options.c](#), [xuartns550.h](#)
- XUartNs550_SetHandler() : [xuartns550_intr.c](#), [xuartns550.h](#)
- XUartNs550_SetOptions() : [xuartns550_options.c](#), [xuartns550.h](#)
- Xuint16 : [xbasic_types.h](#)
- Xuint32 : [xbasic_types.h](#)
- XUINT64_LSW : [xbasic_types.h](#)
- XUINT64_MSW : [xbasic_types.h](#)
- Xuint8 : [xbasic_types.h](#)
- XUN_ERROR_BREAK_MASK : [xuartns550.h](#)
- XUN_ERROR_FRAMING_MASK : [xuartns550.h](#)
- XUN_ERROR_NONE : [xuartns550.h](#)
- XUN_ERROR_OVERRUN_MASK : [xuartns550.h](#)
- XUN_ERROR_PARITY_MASK : [xuartns550.h](#)
- XUN_EVENT_MODEM : [xuartns550.h](#)
- XUN_EVENT_RECV_DATA : [xuartns550.h](#)
- XUN_EVENT_RECV_ERROR : [xuartns550.h](#)
- XUN_EVENT_RECV_TIMEOUT : [xuartns550.h](#)
- XUN_EVENT_SENT_DATA : [xuartns550.h](#)
- XUN_FORMAT_1_STOP_BIT : [xuartns550.h](#)
- XUN_FORMAT_2_STOP_BIT : [xuartns550.h](#)
- XUN_FORMAT_5_BITS : [xuartns550.h](#)
- XUN_FORMAT_6_BITS : [xuartns550.h](#)
- XUN_FORMAT_7_BITS : [xuartns550.h](#)
- XUN_FORMAT_8_BITS : [xuartns550.h](#)
- XUN_FORMAT_EVEN_PARITY : [xuartns550.h](#)
- XUN_FORMAT_NO_PARITY : [xuartns550.h](#)
- XUN_FORMAT_ODD_PARITY : [xuartns550.h](#)
- XUN_MODEM_CTS_DELTA_MASK : [xuartns550.h](#)
- XUN_MODEM_CTS_MASK : [xuartns550.h](#)
- XUN_MODEM_DCD_DELTA_MASK : [xuartns550.h](#)
- XUN_MODEM_DCD_MASK : [xuartns550.h](#)
- XUN_MODEM_DSR_DELTA_MASK : [xuartns550.h](#)
- XUN_MODEM_DSR_MASK : [xuartns550.h](#)
- XUN_MODEM_RING_STOP_MASK : [xuartns550.h](#)

- XUN_MODEM_RINGING_MASK : [xuartns550.h](#)
 - XUN_OPTION_ASSERT_DTR : [xuartns550.h](#)
 - XUN_OPTION_ASSERT_OUT1 : [xuartns550.h](#)
 - XUN_OPTION_ASSERT_OUT2 : [xuartns550.h](#)
 - XUN_OPTION_ASSERT_RTS : [xuartns550.h](#)
 - XUN_OPTION_DATA_INTR : [xuartns550.h](#)
 - XUN_OPTION_FIFOS_ENABLE : [xuartns550.h](#)
 - XUN_OPTION_LOOPBACK : [xuartns550.h](#)
 - XUN_OPTION_MODEM_INTR : [xuartns550.h](#)
 - XUN_OPTION_RESET_RX_FIFO : [xuartns550.h](#)
 - XUN_OPTION_RESET_TX_FIFO : [xuartns550.h](#)
 - XUN_OPTION_SET_BREAK : [xuartns550.h](#)
 - XUT_ALLMEMTESTS : [xutil.h](#)
 - XUT_FIXEDPATTERN : [xutil.h](#)
 - XUT_INCREMENT : [xutil.h](#)
 - XUT_INVERSEADDR : [xutil.h](#)
 - XUT_MAXTEST : [xutil.h](#)
 - XUT_WALKONES : [xutil.h](#)
 - XUT_WALKZEROS : [xutil.h](#)
 - XUtil_MemoryTest16() : [xutil_memtest.c](#), [xutil.h](#)
 - XUtil_MemoryTest32() : [xutil_memtest.c](#), [xutil.h](#)
 - XUtil_MemoryTest8() : [xutil_memtest.c](#), [xutil.h](#)
 - XWaitInAssert : [xbasic_types.c](#)
 - XWdtTb_ConfigTable : [xwdttb_i.h](#), [xwdttb_g.c](#)
 - XWdtTb_GetTbValue() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_Initialize() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_IsWdtExpired() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_mDisableWdt : [xwdttb_l.h](#)
 - XWdtTb_mEnableWdt : [xwdttb_l.h](#)
 - XWdtTb_mGetTimebaseReg : [xwdttb_l.h](#)
 - XWdtTb_mHasExpired : [xwdttb_l.h](#)
 - XWdtTb_mHasReset : [xwdttb_l.h](#)
 - XWdtTb_mRestartWdt : [xwdttb_l.h](#)
 - XWdtTb_RestartWdt() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_SelfTest() : [xwdttb_selftest.c](#), [xwdttb.h](#)
 - XWdtTb_Start() : [xwdttb.h](#), [xwdttb.c](#)
 - XWdtTb_Stop() : [xwdttb.h](#), [xwdttb.c](#)
-

hdlc/v1_00_a/src/xhdlc_selftest.c File Reference

Detailed Description

Self-test and diagnostic functions of the [XHdlc](#) driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	jhl	04/02/02	First release

```
#include "xbasic_types.h"  
#include "xhdlc_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"  
#include "xpacket_fifo_v1_00_b.h"
```

Defines

```
#define LOOPBACK_BYTE_COUNT
```

Functions

[XStatus XHdlc_SelfTest](#) ([XHdlc](#) *InstancePtr)

Define Documentation

`#define LOOPBACK_BYTE_COUNT`

Performs a loopback test on the HDLC device by sending and receiving a frame using loopback mode.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

XST_SUCCESS	Loopback was successful
XST_LOOPBACK_ERROR	Loopback was unsuccessful

Note:

None.

Function Documentation

`XStatus XHdlc_SelfTest(XHdlc * InstancePtr)`

Performs a self-test on the HDLC device. The test includes:

- Run self-test on the FIFOs, and IPIF components
- Reset the HDLC device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
-------------	--------------------------

XST_REGISTER_ERROR
XST_LOOPBACK_ERROR

HDL failed register reset test
Internal loopback failed

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

XHdlc Struct Reference

```
#include <xhdlc.h>
```

Detailed Description

The XHdlc driver instance data. The user is required to allocate a variable of this type for every HDLC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `hdlc/v1_00_a/src/xhdlc.h`
-

hdlc/v1_00_a/src/xhdlc.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xhdlc.h,v 1.5 2002/06/28 18:18:15 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file hdlc/v1_00_a/src/xhdlc.h
00026 *
00027 * The Xilinx HDLC driver component which supports the Xilinx HDLC device.
00028 *
00029 * <b>Driver Description</b>
00030 *
00031 * The device driver enables higher layer software (e.g., an application) to
00032 * communicate to HDLC devices. The driver handles transmission and reception of
00033 * HDLC frames, as well as configuration of the devices. A single device driver
00034 * can support multiple HDLC devices.
00035 *
00036 * The driver is designed for a zero-copy buffer scheme. That is, the driver
00037 * will
00038 * not copy buffers. This avoids potential throughput bottlenecks within the
00039 * driver.
00040 *
00041 * Since the driver is a simple pass-through mechanism between an application
00042 * and the HDLC devices, no assembly or disassembly of HDLC frames is done at
```

```

the
00042 * driver-level. This assumes that the application passes a correctly
00043 * formatted HDLC frame to the driver for transmission, and that the driver
00044 * does not validate the contents of an incoming frame.
00045 *
00046 * The driver supports polled mode and interrupt mode only with DMA
00047 * scatter-gather. FIFO interrupt mode without DMA scatter-gather is not yet
00048 * supported. The default mode of operation is polled.
00049 *
00050 * <b>Device Configuration</b>
00051 *
00052 * The device can be configured in various ways during the FPGA implementation
00053 * process. Configuration parameters are stored in the xhdlc_g.c file. A table
00054 * is defined where each entry contains configuration information for an HDLC
00055 * device. This information includes such things as the base address
00056 * of the memory-mapped device.
00057 *
00058 * <b>HDLC Frame Description</b>
00059 *
00060 * An HDLC frame contains a number of fields as illustrated below. The size of
00061 * several fields, the Address and FCS fields, are variable depending on the
00062 * the configuration of the device as set through the options.
00063 *
00064 *   <Opening Flag><Address><Control><Data><FCS><Closing Flag>
00065 *
00066 * <b>Asserts</b>
00067 *
00068 * Asserts are used within all Xilinx drivers to enforce constraints on argument
00069 * values. Asserts can be turned off on a system-wide basis by defining, at
compile
00070 * time, the NDEBUG identifier. By default, asserts are turned on and it is
00071 * recommended that application developers leave asserts on during development.
00072 *
00073 * @note
00074 *
00075 * This driver is intended to be RTOS and processor independent. It works
00076 * with physical addresses only. Any needs for dynamic memory management,
00077 * threads or thread mutual exclusion, virtual memory, or cache control must
00078 * be satisfied by the layer above this driver.
00079 *
00080 * <pre>
00081 * MODIFICATION HISTORY:
00082 *
00083 * Ver    Who    Date      Changes
00084 * -----
00085 * 1.00a  jhl    04/01/02  First release
00086 * </pre>
00087 *
00088 * *****/
00089
00090 #ifndef XHDLC_H /* prevent circular inclusions */
00091 #define XHDLC_H /* by using protection macros */
00092

```

```

00093 /***** Include Files *****/
00094
00095 #include "xbasic_types.h"
00096 #include "xstatus.h"
00097 #include "xpacket_fifo_v1_00_b.h" /* Uses v1.00b of Packet Fifo */
00098 #include "xdma_channel.h"
00099 #include "xhdlc_l.h"
00100 #include "xparameters.h"
00101
00102 /***** Constant Definitions *****/
00103
00104 /** @name Configuration options
00105  *
00106  * Device configuration options (see the XHdlc_SetOptions() and
00107  * XHdlc_GetOptions() for information on how to use these options). Note that
00108  * the option to remove addresses from the receive buffers is only valid when
00109  * address filtering is on.
00110  * @{
00111  */
00112 #define XHD_OPTION_POLLED 0x01 /**< polled mode, default */
00113 #define XHD_OPTION_LOOPBACK 0x02 /**< loopback transmit to receive */
00114 #define XHD_OPTION_CRC_32 0x04 /**< send/receive 32 bit CRCs */
00115 #define XHD_OPTION_CRC_DISABLE 0x08 /**< disable sending CRCs */
00116 #define XHD_OPTION_RX_FILTER_ADDR 0x10 /**< receive address filtering */
00117 #define XHD_OPTION_RX_REMOVE_ADDR 0x20 /**< don't buffer receive addresses */
00118 #define XHD_OPTION_RX_BROADCAST 0x40 /**< receive broadcast addresses */
00119 #define XHD_OPTION_RX_16_ADDR 0x80 /**< receive 16 bit addresses, 8 bit
00120 } is the default */
00121
00122 /* The following constants determine the configuration of the hardware device
00123  * and are used to allow the driver to verify it can operate with the hardware
00124  * configuration.
00125  */
00126 #define XHD_CFG_NO_IPIF 0 /* Not supported by the driver */
00127 #define XHD_CFG_NO_DMA 1 /* No DMA */
00128 #define XHD_CFG_SIMPLE_DMA 2 /* Not supported by the driver */
00129 #define XHD_CFG_DMA_SG 3 /* DMA scatter gather */
00130
00131 /*
00132  * Some default values for interrupt coalescing within the scatter-gather
00133  * DMA engine.
00134  */
00135 #define XHD_SGDMA_DFT_THRESHOLD 1 /* Default pkt threshold */
00136 #define XHD_SGDMA_MAX_THRESHOLD 255 /* Maximum pkt threshold */
00137 #define XHD_SGDMA_DFT_WAITBOUND 5 /* Default pkt wait bound (msec) */
00138 #define XHD_SGDMA_MAX_WAITBOUND 1023 /* Maximum pkt wait bound (msec) */
00139
00140 /* The following constants control the number of buffers for DMA scatter
00141  * gather processing
00142  */
00143 #define XHD_MIN_BUFFERS 16 /* minimum number of receive buffers */

```

```

00144 #define XHD_DFT_BUFFERS      64      /* default number of receive buffers */
00145
00146 #define XHD_MIN_RECV_DESC     8        /* minimum # of recv descriptors */
00147 #define XHD_DFT_RECV_DESC     32      /* default # of recv descriptors */
00148
00149 #define XHD_MIN_SEND_DESC     8        /* minimum # of send descriptors */
00150 #define XHD_DFT_SEND_DESC     32      /* default # of send descriptors */
00151
00152 /***** Type Definitions *****/
00153
00154 /**
00155  * HDLC statistics (see XHdlc_GetStats() and XHdlc_ClearStats())
00156  */
00157 typedef struct
00158 {
00159     Xuint16 XmitFrames;           /**< Number of frames transmitted */
00160     Xuint16 XmitBytes;           /**< Number of bytes transmitted */
00161
00162     Xuint16 RecvFrames;          /**< Number of frames received */
00163     Xuint16 RecvBytes;           /**< Number of bytes received */
00164     Xuint16 RecvFcsErrors;       /**< Number of frames discarded due
00165                                     to FCS errors */
00166     Xuint16 RecvAlignmentErrors; /**< Number of frames received with
00167                                     alignment errors */
00168     Xuint16 RecvOverrunErrors;   /**< Number of frames discarded due
00169                                     to overrun errors */
00170     Xuint16 RecvAbortedFrames;   /**< Number of transmit aborted frames */
00171
00172     Xuint16 FifoErrors;           /**< Number of FIFO errors since init */
00173     Xuint16 DmaErrors;           /**< Number of DMA errors */
00174     Xuint16 RecvInterrupts;      /**< Number of receive interrupts */
00175     Xuint16 XmitInterrupts;      /**< Number of transmit interrupts */
00176     Xuint16 HdlcInterrupts;     /**< Number of HDLC (device) interrupts */
00177 } XHdlc_Stats;
00178
00179 /**
00180  * This typedef contains configuration information for a device.
00181  */
00182 typedef struct
00183 {
00184     Xuint16 DeviceId;             /**< Unique ID of device */
00185     Xuint32 BaseAddress;          /**< Device base address */
00186     Xuint32 TransmitFifoSize;     /**< Transmit FIFO size in bytes */
00187     Xuint32 ReceiveFifoSize;      /**< Receive FIFO size in bytes */
00188     Xuint8 IpIfDmaConfig;         /**< IPIF/DMA hardware configuration */
00189 } XHdlc_Config;
00190
00191 /** @name Typedefs for callbacks
00192  * Callback functions.
00193  * @{

```

```

00194 */
00195 /**
00196 * Callback when data is sent or received with scatter-gather DMA.
00197 * @param CallbackRef is a callback reference passed in by the upper layer
00198 *         when setting the callback functions, and passed back to the upper
00199 *         layer when the callback is invoked.
00200 * @param FrameCount is the number of frames sent or received.
00201 */
00202 typedef void (*XHdlc_SgHandler)(void *CallbackRef, unsigned FrameCount);
00203
00204 /**
00205 * Callback when errors occur in interrupt mode.
00206 * @param CallbackRef is a callback reference passed in by the upper layer
00207 *         when setting the callback functions, and passed back to the upper
00208 *         layer when the callback is invoked.
00209 * @param ErrorCode indicates the error that occurred.
00210 */
00211 typedef void (*XHdlc_ErrorHandler)(void *CallbackRef, XStatus ErrorCode);
00212 /*@}*/
00213
00214 /**
00215 * The XHdlc driver instance data. The user is required to allocate a
00216 * variable of this type for every HDLC device in the system. A pointer
00217 * to a variable of this type is then passed to the driver API functions.
00218 */
00219 typedef struct
00220 {
00221     Xuint32 BaseAddress;           /* Base address for device (IPIF) */
00222     Xuint32 IsStarted;            /* Device is currently started */
00223     Xuint32 IsReady;             /* Device is initialized and ready */
00224     Xboolean IsPolled;           /* Polled/interrupt driven configuration */
00225     XHdlc_Config *ConfigPtr;     /* A pointer to the device configuration */
00226
00227     XHdlc_Stats Stats;
00228     XPacketFifoV100b RecvFifo;   /* FIFO used by receive channel */
00229     XPacketFifoV100b SendFifo;   /* FIFO used by send channel */
00230
00231     XDmaChannel RecvChannel;     /* DMA receive channel */
00232     XDmaChannel SendChannel;     /* DMA send channel */
00233
00234     /*
00235      * Callbacks
00236      */
00237     XHdlc_SgHandler SgRecvHandler;
00238     void *SgRecvRef;
00239     XHdlc_SgHandler SgSendHandler;
00240     void *SgSendRef;
00241     XHdlc_ErrorHandler ErrorHandler;
00242     void *ErrorRef;
00243 } XHdlc;
00244

```

```
00245 /***** Macros (Inline Functions) Definitions *****/
00246
00247
00248 /***** Function Prototypes *****/
00249
00250 /*
00251  * Initialization functions in xhdlc.c
00252  */
00253 XStatus XHdlc_Initialize(XHdlc *InstancePtr, Xuint16 DeviceId);
00254 XStatus XHdlc_Start(XHdlc *InstancePtr);
00255 XStatus XHdlc_Stop(XHdlc *InstancePtr);
00256 void XHdlc_Reset(XHdlc *InstancePtr);
00257 XHdlc_Config *XHdlc_LookupConfig(Xuint16 DeviceId);
00258
00259 XStatus XHdlc_Send(XHdlc *InstancePtr, Xuint8 *FramePtr, unsigned ByteCount);
00260 XStatus XHdlc_Recv(XHdlc *InstancePtr, Xuint8 *FramePtr,
00261                  unsigned *ByteCountPtr, Xuint8 *FrameStatusPtr);
00262
00263 /*
00264  * Diagnostic functions in xhdlc_selftest.c
00265  */
00266 XStatus XHdlc_SelfTest(XHdlc *InstancePtr);
00267
00268 /*
00269  * Options functions in xhdlc_options.c
00270  */
00271 XStatus XHdlc_SetOptions(XHdlc *InstancePtr, Xuint8 Options);
00272 Xuint8 XHdlc_GetOptions(XHdlc *InstancePtr);
00273 void XHdlc_SetAddress(XHdlc *InstancePtr, Xuint16 Address);
00274 Xuint16 XHdlc_GetAddress(XHdlc *InstancePtr);
00275
00276 /*
00277  * Statistics in xhdlc_stats.c
00278  */
00279 void XHdlc_GetStats(XHdlc *InstancePtr, XHdlc_Stats *StatsPtr);
00280 void XHdlc_ClearStats(XHdlc *InstancePtr);
00281
00282 /*
00283  * DMA scatter gather functions in xhdlc_dmasg.c
00284  */
00285 XStatus XHdlc_SgSend(XHdlc *InstancePtr, XBufDescriptor *BdPtr);
00286 XStatus XHdlc_SgRecv(XHdlc *InstancePtr, XBufDescriptor *BdPtr);
00287 XStatus XHdlc_SgGetSendFrame(XHdlc* InstancePtr, XBufDescriptor **PtrToBdPtr,
00288                             unsigned *BdCountPtr);
00289 XStatus XHdlc_SgGetRecvFrame(XHdlc* InstancePtr, XBufDescriptor **PtrToBdPtr,
00290                              unsigned *BdCountPtr);
00291
00292 XStatus XHdlc_SetSgRecvSpace(XHdlc *InstancePtr, Xuint32 *MemoryPtr,
00293                             unsigned ByteCount);
00294 XStatus XHdlc_SetSgSendSpace(XHdlc *InstancePtr, Xuint32 *MemoryPtr,
```

```
00295         unsigned ByteCount);
00296
00297 /*
00298  * DMA scatter gather interrupt handler and callbacks in xhdlc_dmasg.c
00299  */
00300 void XHdlc_InterruptHandler(void *InstancePtr);
00301 void XHdlc_SetSgRecvHandler(XHdlc *InstancePtr, void *CallBackRef,
00302                             XHdlc_SgHandler FuncPtr);
00303 void XHdlc_SetSgSendHandler(XHdlc *InstancePtr, void *CallBackRef,
00304                             XHdlc_SgHandler FuncPtr);
00305 void XHdlc_SetErrorHandler(XHdlc *InstancePtr, void *CallBackRef,
00306                             XHdlc_ErrorHandler FuncPtr);
00307
00308 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

hdlc/v1_00_a/src/xhdlc.h File Reference

Detailed Description

The Xilinx HDLC driver component which supports the Xilinx HDLC device.

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to HDLC devices. The driver handles transmission and reception of HDLC frames, as well as configuration of the devices. A single device driver can support multiple HDLC devices.

The driver is designed for a zero-copy buffer scheme. That is, the driver will not copy buffers. This avoids potential throughput bottlenecks within the driver.

Since the driver is a simple pass-through mechanism between an application and the HDLC devices, no assembly or disassembly of HDLC frames is done at the driver-level. This assumes that the application passes a correctly formatted HDLC frame to the driver for transmission, and that the driver does not validate the contents of an incoming frame.

The driver supports polled mode and interrupt mode only with DMA scatter-gather. FIFO interrupt mode without DMA scatter-gather is not yet supported. The default mode of operation is polled.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the [xhdlc_g.c](#) file. A table is defined where each entry contains configuration information for an HDLC device. This information includes such things as the base address of the memory-mapped device.

HDLC Frame Description

An HDLC frame contains a number of fields as illustrated below. The size of several fields, the Address

and FCS fields, are variable depending on the the configuration of the device as set through the options.

<Opening Flag><Address><Control><Data><FCS><Closing Flag>

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the NDEBUG identifier. By default, asserts are turned on and it is recommended that application developers leave asserts on during development.

Note:

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	jhl	04/01/02	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xpacket_fifo_v1_00_b.h"  
#include "xdma_channel.h"  
#include "xhdlc_1.h"  
#include "xparameters.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XHdlc  
struct XHdlc\_Config  
struct XHdlc\_Stats
```

Typedefs for callbacks

Callback functions.

```
typedef void(* XHdlc_SgHandler )(void *CallBackRef, unsigned FrameCount)
typedef void(* XHdlc_ErrorHandler )(void *CallBackRef, XStatus ErrorCode)
```

Defines

```
#define XHD_OPTION_POLLED
#define XHD_OPTION_LOOPBACK
#define XHD_OPTION_CRC_32
#define XHD_OPTION_CRC_DISABLE
#define XHD_OPTION_RX_FILTER_ADDR
#define XHD_OPTION_RX_REMOVE_ADDR
#define XHD_OPTION_RX_BROADCAST
#define XHD_OPTION_RX_16_ADDR
```

Functions

```
XStatus XHdlc_Initialize (XHdlc *InstancePtr, Xuint16 DeviceId)
XStatus XHdlc_Start (XHdlc *InstancePtr)
XStatus XHdlc_Stop (XHdlc *InstancePtr)
void XHdlc_Reset (XHdlc *InstancePtr)
XHdlc_Config * XHdlc_LookupConfig (Xuint16 DeviceId)
XStatus XHdlc_Send (XHdlc *InstancePtr, Xuint8 *FramePtr, unsigned ByteCount)
XStatus XHdlc_Recv (XHdlc *InstancePtr, Xuint8 *FramePtr, unsigned *ByteCountPtr, Xuint8
*FrameStatusPtr)
XStatus XHdlc_SelfTest (XHdlc *InstancePtr)
XStatus XHdlc_SetOptions (XHdlc *InstancePtr, Xuint8 Options)
Xuint8 XHdlc_GetOptions (XHdlc *InstancePtr)
void XHdlc_SetAddress (XHdlc *InstancePtr, Xuint16 Address)
Xuint16 XHdlc_GetAddress (XHdlc *InstancePtr)
void XHdlc_GetStats (XHdlc *InstancePtr, XHdlc_Stats *StatsPtr)
void XHdlc_ClearStats (XHdlc *InstancePtr)
```

XStatus XHdlc_SgSend (**XHdlc** *InstancePtr, XBufDescriptor *BdPtr)
XStatus XHdlc_SgRecv (**XHdlc** *InstancePtr, XBufDescriptor *BdPtr)
XStatus XHdlc_SgGetSendFrame (**XHdlc** *InstancePtr, XBufDescriptor **PtrToBdPtr, unsigned *BdCountPtr)
XStatus XHdlc_SgGetRecvFrame (**XHdlc** *InstancePtr, XBufDescriptor **PtrToBdPtr, unsigned *BdCountPtr)
XStatus XHdlc_SetSgRecvSpace (**XHdlc** *InstancePtr, **Xuint32** *MemoryPtr, unsigned ByteCount)
XStatus XHdlc_SetSgSendSpace (**XHdlc** *InstancePtr, **Xuint32** *MemoryPtr, unsigned ByteCount)
void **XHdlc_InterruptHandler** (void *InstancePtr)
void **XHdlc_SetSgRecvHandler** (**XHdlc** *InstancePtr, void *CallBackRef, **XHdlc_SgHandler** FuncPtr)
void **XHdlc_SetSgSendHandler** (**XHdlc** *InstancePtr, void *CallBackRef, **XHdlc_SgHandler** FuncPtr)
void **XHdlc_SetErrorHandler** (**XHdlc** *InstancePtr, void *CallBackRef, **XHdlc_ErrorHandler** FuncPtr)

Define Documentation

#define XHD_OPTION_CRC_32

send/receive 32 bit CRCs

#define XHD_OPTION_CRC_DISABLE

disable sending CRCs

#define XHD_OPTION_LOOPBACK

loopback transmit to receive

#define XHD_OPTION_POLLED

polled mode, default

#define XHD_OPTION_RX_16_ADDR

receive 16 bit addresses, 8 bit } is the default

```
#define XHD_OPTION_RX_BROADCAST
```

receive broadcast addresses

```
#define XHD_OPTION_RX_FILTER_ADDR
```

receive address filtering

```
#define XHD_OPTION_RX_REMOVE_ADDR
```

don't buffer receive addresses

Typedef Documentation

```
typedef void(* XHdlc_ErrorHandler)(void *CallbackRef, XStatus ErrorCode)
```

Callback when errors occur in interrupt mode.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

ErrorCode indicates the error that occurred.

```
typedef void(* XHdlc_SgHandler)(void *CallbackRef, unsigned FrameCount)
```

Callback when data is sent or received with scatter-gather DMA.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

FrameCount is the number of frames sent or received.

Function Documentation

```
void XHdlc_ClearStats( XHdlc * InstancePtr)
```

Clear the statistics for the specified HDLC driver instance.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

None.

Note:

None.

Xuint16 XHdlc_GetAddress(**XHdlc** * *InstancePtr*)

Get the receive address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

The receive address of the HDLC device.

Note:

None.

Xuint8 XHdlc_GetOptions(**XHdlc** * *InstancePtr*)

Get HDLC driver/device options. A value is returned which is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

The value of the HDLC options. The value is a bit-mask representing all options that are currently enabled. See **xhdlc.h** for a description of the available options.

Note:

None.

```
void XHdlc_GetStats( XHdlc * InstancePtr,  
                    XHdlc_Stats * StatsPtr  
                    )
```

Get a copy of the statistics structure, which contains the current statistics for this driver. The statistics are only cleared at initialization or on demand using the [XHdlc_ClearStats\(\)](#) function.

The FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the caller.

Parameters:

InstancePtr is a pointer to the [XHdlc](#) instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

```
XStatus XHdlc_Initialize( XHdlc * InstancePtr,  
                         Xuint16 DeviceId  
                         )
```

Initialize a specific [XHdlc](#) instance/driver. The initialization entails:

- Initialize fields of the [XHdlc](#) structure
- Clear the HDLC statistics for this device
- Configure the FIFO components and DMA channels
- Reset the HDLC device

The driver defaults to polled mode operation. Interrupt mode can be selected using the [SetOptions\(\)](#) function and turning off polled mode.

Parameters:

InstancePtr is a pointer to the [XHdlc](#) instance to be worked on.

DeviceId is the unique id of the device controlled by this **XHdlc** instance. Passing in a device id associates the generic **XHdlc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.
- XST_NO_FEATURE if the device configuration information indicates a feature that is not supported by this driver (no IPIF or simple DMA).

Note:

None.

void XHdlc_InterruptHandler(void * *InstancePtr*)

Interrupt handler for the HDLC driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: the HDLC device, the send packet FIFO, the receive packet FIFO, the send DMA channel, or the receive DMA channel. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the HDLC device.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the HDLC device instance for the interrupt.

Returns:

None.

Note:

None.

XHdlc_Config* XHdlc_LookupConfig(Xuint16 *DeviceId*)

Lookup the device configuration based on the unique device ID. The table `XHdlc_ConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or `XNULL` if no match is found.

Note:

None.

```
XStatus XHdlc_Recv( XHdlc * InstancePtr,  
                  Xuint8 * FramePtr,  
                  unsigned * ByteCountPtr,  
                  Xuint8 * FrameStatusPtr  
                  )
```

Receive an HDLC frame in polled mode. The driver receives the frame directly from the devices packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The buffer into which the frame will be received must be word-aligned.

The frames which are received by the device are stripped of the Opening Flag and Closing Flag fields such that buffers which receive data will not contain these fields. The frames do contain the FCS field. The Address field may or may not be contained in a receive buffer depending on the options set.

Parameters:

InstancePtr is a pointer to the `XHdlc` instance to be worked on.

FramePtr is a pointer to a 32 bit word aligned buffer into which the received HDLC frame will be copied.

ByteCountPtr is both an input and output parameter. It is a pointer to a 32-bit word that contains the number of bytes in the specified frame buffer on entry and the number of bytes in the received frame on return from the function.

FrameStatusPtr is an output which is changed by the driver to contain the status of the frame. It indicates any status that occurred for the received frame.

Returns:

- `XST_SUCCESS` if the frame was received successfully
- `XST_DEVICE_IS_STOPPED` if the device has not yet been started
- `XST_NO_DATA` if there is no frame to be received from the FIFO

- XST_BUFFER_TOO_SMALL if the specified receive buffer is smaller than the the received frame. The received frame is not retrieved from the receive FIFO such that a reset of the device is necessary to resynchronize the internal length and data FIFOs.
- XST_FIFO_ERROR if a non-recoverable FIFO error has occurred. A reset of the device is necessary to clear this error.

Note:

Receive buffer must also be 32-bit aligned. The user must ensure that the size of the receive buffer is large enough to hold the frames received.

void XHdlc_Reset(XHdlc * InstancePtr)

Reset the HDLC instance. This is a graceful reset in that the device is stopped first then it resets the FIFOs and DMA channels if present. Reset must only be called after the driver has been initialized. The reset does not remove any of the buffer descriptors from the scatter-gather list for DMA.

The configuration after this reset is as follows:

- Disabled transmitter and receiver
- Device interrupts are disabled
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation

The upper layer software is responsible for re-configuring (if necessary) and restarting the HDLC device after the reset. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

None.

Note:

None.

XStatus XHdlc_SelfTest(XHdlc * InstancePtr)

Performs a self-test on the HDLC device. The test includes:

- Run self-test on the FIFOs, and IPIF components
- Reset the HDLC device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_REGISTER_ERROR	HDLC failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed

Note:

None.

```
XStatus XHdlc_Send( XHdlc * InstancePtr,  
                   Xuint8 * FramePtr,  
                   unsigned ByteCount  
                   )
```

Send an HDLC frame in polled mode. The driver writes the frame directly to the HDLC packet FIFO. Statistics are updated if an error previously occurred. The buffer to be sent must be word-aligned. This function is a non-blocking function in that it does not wait for the frame to be sent before returning.

The hardware device adds the Opening Flag, FCS, and Closing Flag fields to the frame such that these should not be put in the buffers which are to be sent.

The hardware device uses a length FIFO to keep track of each frame that has been put into the data FIFO. When sending a lot of short frames it is possible to fill this FIFO so that another frame cannot be sent until a frame has been sent by the device.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

FramePtr is a pointer to a 32 bit word aligned buffer containing the HDLC frame to be sent.
ByteCount is the size of the HDLC frame as an input. This size must be smaller than the size of the transmit FIFO of the device.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_FIFO_NO_ROOM if there is no room in the devices FIFOs for this frame.
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.

Note:

None.

```
void XHdlc_SetAddress( XHdlc * InstancePtr,  
                     Xuint16 Address  
                     )
```

Set the receive address for this driver/device. The address is a 8 or 16 bit value within a HDLC frame.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Address is the address to be set.

Returns:

None.

Note:

None.

```
void XHdlc_SetErrorHandler( XHdlc * InstancePtr,  
                           void * CallbackRef,  
                           XHdlc_ErrorHandler FuncPtr  
                           )
```

Sets the callback function for handling errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback which should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- `XST_DMA_ERROR` indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- `XST_FIFO_ERROR` indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- `XST_RESET_ERROR` indicates an unrecoverable HDLC device error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- `XST_ERROR_COUNT_MAX` indicates the counters of the HDLC device have reached the maximum value and that the statistics of the HDLC device should be cleared.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XHdlc_SetOptions( XHdlc * InstancePtr,  
                          Xuint8 Options  
                          )
```

Set HDLC driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Options is a bit-mask representing the HDLC options to turn on or off. See **xhdlc.h** for a description of the available options.

Returns:

- XST_SUCCESS if the options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped
- XST_NO_FEATURE if the polled option is being turned off and the device configuration information indicates DMA scatter gather is not supported. This driver does not support interrupt driven without DMA scatter-gather (FIFO only) yet.

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
void XHdlc_SetSgRecvHandler( XHdlc *           InstancePtr,  
                             void *           CallbackRef,  
                             XHdlc_SgHandler FuncPtr  
                             )
```

Sets the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of frames, determined by the DMA scatter-gather packet threshold, are received. The number of received frames is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each received frame from the list and should attach a new buffer to each descriptor. It is important that the specified number of frames passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XHdlc_SetSgRecvSpace( XHdlc * InstancePtr,  
                               Xuint32 * MemoryPtr,  
                               unsigned ByteCount  
                               )
```

Gives the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the HDLC driver. The memory space must be word-aligned.

This function must be called prior to calling **XHdlc_Start()**.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the HDLC device is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XHdlc_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
void XHdlc_SetSgSendHandler( XHdlc *      InstancePtr,
                             void *      CallbackRef,
                             XHdlc_SgHandler FuncPtr
                             )
```

Sets the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of frames, determined by the DMA scatter-gather packet threshold, are sent. The number of sent frames is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each sent frame from the list and should also free the buffers attached to the descriptors if necessary. It is important that the specified number of frames passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XHdlc_SetSgSendSpace( XHdlc *      InstancePtr,
                              Xuint32 *      MemoryPtr,
                              unsigned      ByteCount
                              )
```

Gives the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the HDLC driver. The memory space must be word-aligned.

This function must be called prior to calling [XHdlc_Start\(\)](#).

Parameters:

InstancePtr is a pointer to the [XHdlc](#) instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the HDLC device is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the [XHdlc_Initialize\(\)](#) function because the DMA channel components must be initialized before the memory space is set.

```
XStatus XHdlc_SgGetRecvFrame( XHdlc *           InstancePtr,  
                             XBufDescriptor ** PtrToBdPtr,  
                             unsigned *      BdCountPtr  
                             )
```

Gets the first buffer descriptor of the oldest frame which was received by the scatter-gather DMA channel of the HDLC device. This function is provided to be called from a callback function such that the buffer descriptors for received frames can be processed. The function should be called by the application repetitively for the number of frames indicated as an argument in the callback function.

Parameters:

InstancePtr is a pointer to the [XHdlc](#) instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the frame. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the frame. This input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor

pointed to by `PtrToBdPtr` and a count of the number of buffer descriptors for the frame pointed to by `BdCountPtr` are both modified if the return status indicates success. The status values are:

- `XST_SUCCESS` if a descriptor was successfully returned to the driver.
- `XST_NOT_SGDMA` if the device is not in scatter-gather DMA mode.
- `XST_DMA_SG_NO_LIST` if the scatter gather list has not been created.
- `XST_DMA_SG_LIST_EMPTY` if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus XHdlc_SgGetSendFrame( XHdlc *           InstancePtr,  
                               XBufDescriptor ** PtrToBdPtr,  
                               unsigned *       BdCountPtr  
                               )
```

Gets the first buffer descriptor of the oldest frame which was sent by the scatter-gather DMA channel of the HDLC device. This function is provided to be called from a callback function such that the buffer descriptors for sent frames can be processed. The function should be called by the application repetitively for the number of frames indicated as an argument in the callback function.

Parameters:

InstancePtr is a pointer to the `XHdlc` instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the frame. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the frame. this input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by `PtrToBdPtr` and a count of the number of buffer descriptors for the frame pointed to by `BdCountPtr` are both modified if the return status indicates success. The status values are:

- `XST_SUCCESS` if a descriptor was successfully returned to the driver.
- `XST_NOT_SGDMA` if the device is not in scatter-gather DMA mode.
- `XST_DMA_SG_NO_LIST` if the scatter gather list has not been created.
- `XST_DMA_SG_LIST_EMPTY` if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus XHdlc_SgRecv( XHdlc * InstancePtr,  
                    XBufDescriptor * BdPtr  
                    )
```

Adds this descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. The contents of the specified buffer descriptor are copied into the scatter-gather transmit list. This function can be called when the device is started or stopped.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

Note:

None.

```
XStatus XHdlc_SgSend( XHdlc * InstancePtr,  
                    XBufDescriptor * BdPtr  
                    )
```

Sends a HDLC frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire frame may or may not be contained within one descriptor. The contents of the buffer descriptor are copied into the scatter-gather transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the DMA control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted HDLC frame based upon the

configuration of the HDLC device. The HDLC device must be started before calling this function.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Returns:

- XST_SUCCESS if the buffer was successfully sent
- XST_DEVICE_IS_STOPPED if the HDLC device has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

XStatus XHdlc_Start(XHdlc * InstancePtr)

Start the HDLC device and driver by enabling the transmitter and receiver. This function must be called before other functions to send or receive data. It supports either polled or DMA scatter gather interrupt driven modes of operation. It does not yet support interrupt driven with FIFOs (non-DMA) or simple DMA without scatter gather.

If the driver is configured for interrupt driven operation, the interrupts of the device are enabled. The user should have connected the interrupt handler of the driver to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

Prior to calling this function, the functions **XHdlc_SetSgRecvSpace()** and **XHdlc_SetSgSendSpace()** should be called to setup the memory buffers and descriptor lists for the DMA scatter-gather.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_DEVICE_IS_STARTED if the device is already started

- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.
- XST_DMA_SG_LIST_EMPTY if configured for scatter-gather DMA and a descriptor list has been created for the receive channel but no buffers inserted into it.

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus XHdlc_Stop(XHdlc * InstancePtr)

Stop the HDLC device as follows:

- If the device is configured with DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode (the higher layer software is responsible for disabling interrupts at the interrupt controller)

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the HDLC frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

None.

hdlc/v1_00_a/src/xhdlc_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of HDLC devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	jhl	04/08/02	First release

```
#include "xhdlc.h"
```

Variables

[XHdlc_Config](#) [XHdlc_ConfigTable](#) [XPAR_XHDLC_NUM_INSTANCES]

Variable Documentation

[XHdlc_Config](#) [XHdlc_ConfigTable](#)[XPAR_XHDLC_NUM_INSTANCES]

This table contains configuration information for each HDLC device in the system.

XHdlc_Config Struct Reference

```
#include <xhdlc.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xuint32 TransmitFifoSize

Xuint32 ReceiveFifoSize

Xuint8 IpIfDmaConfig

Field Documentation

Xuint32 XHdlc_Config::BaseAddress

Device base address

Xuint16 XHdlc_Config::DeviceId

Unique ID of device

Xuint8 XHdlc_Config::IpIfDmaConfig

IPIF/DMA hardware configuration

Xuint32 XHdlc_Config::ReceiveFifoSize

Receive FIFO size in bytes

Xuint32 XHdlc_Config::TransmitFifoSize

Transmit FIFO size in bytes

The documentation for this struct was generated from the following file:

- [hdlc/v1_00_a/src/xhdlc.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

common/v1_00_a/src/xbasic_types.h File Reference

Detailed Description

This file contains basic types for Xilinx software IP. These types do not follow the standard naming convention with respect to using the component name in front of each name because they are considered to be primitives.

Note:

This file contains items which are architecture dependent.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	12/14/01	First release
	rmm	05/09/03	Added "xassert always" macros to rid ourselves of diab compiler warnings

[Go to the source code of this file.](#)

Data Structures

struct **Xuint64**

Primitive types

These primitive types are created for transportability. They are dependent upon the target architecture.

```
typedef unsigned char Xuint8
    typedef char Xint8
typedef unsigned short Xuint16
    typedef short Xint16
typedef unsigned long Xuint32
    typedef long Xint32
    typedef float Xfloat32
    typedef double Xfloat64
typedef unsigned long Xboolean
```

Defines

```
#define XTRUE
#define XFALSE
#define XNULL
#define XUINT64_MSW(x)
#define XUINT64_LSW(x)
#define XASSERT_VOID(expression)
#define XASSERT_NONVOID(expression)
#define XASSERT_VOID_ALWAYS()
#define XASSERT_NONVOID_ALWAYS()
```

Typedefs

```
typedef void(* XInterruptHandler )(void *InstancePtr)
typedef void(* XAssertCallback )(char *FilenamePtr, int LineNumber)
```

Functions

```
void XAssert (char *, int)
void XAssertSetCallback (XAssertCallback Routine)
```

Variables

```
unsigned int XAssertStatus
```

Define Documentation

#define XASSERT_NONVOID(expression)

This assert macro is to be used for functions that do return a value. This in conjunction with the XWaitInAssert boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters:

expression is the expression to evaluate. If it evaluates to false, the assert occurs.

Returns:

Returns 0 unless the XWaitInAssert variable is true, in which case no return is made and an infinite loop is entered.

Note:

None.

#define XASSERT_NONVOID_ALWAYS()

Always assert. This assert macro is to be used for functions that do return a value. Use for instances where an assert should always occur.

Returns:

Returns void unless the XWaitInAssert variable is true, in which case no return is made and an infinite loop is entered.

Note:

None.

#define XASSERT_VOID(expression)

This assert macro is to be used for functions that do not return anything (void). This in conjunction with the XWaitInAssert boolean can be used to accomodate tests so that asserts which fail allow execution to continue.

Parameters:

expression is the expression to evaluate. If it evaluates to false, the assert occurs.

Returns:

Returns void unless the XWaitInAssert variable is true, in which case no return is made and an infinite loop is entered.

Note:

None.

#define XASSERT_VOID_ALWAYS()

Always assert. This assert macro is to be used for functions that do not return anything (void). Use for instances where an assert should always occur.

Returns:

Returns void unless the XWaitInAssert variable is true, in which case no return is made and an infinite loop is entered.

Note:

None.

#define XFALSE

Xboolean false

#define XNULL

Null

#define XTRUE

Xboolean true

#define XUINT64_LSW(x)

Return the least significant half of the 64 bit data type.

Parameters:

x is the 64 bit word.

Returns:

The lower 32 bits of the 64 bit word.

Note:

None.

```
#define XUINT64_MSW( x )
```

Return the most significant half of the 64 bit data type.

Parameters:

x is the 64 bit word.

Returns:

The upper 32 bits of the 64 bit word.

Note:

None.

Typedef Documentation

```
typedef void(* XAssertCallback)(char* FilenamePtr, int LineNumber)
```

This data type defines a callback to be invoked when an assert occurs. The callback is invoked only when asserts are enabled

```
typedef unsigned long Xboolean
```

boolean (XTRUE or XFALSE)

```
typedef float Xfloat32
```

32-bit floating point

```
typedef double Xfloat64
```

64-bit double precision floating point

typedef short Xint16

signed 16-bit

typedef long Xint32

signed 32-bit

typedef char Xint8

signed 8-bit

typedef void(* XInterruptHandler)(void *InstancePtr)

This data type defines an interrupt handler for a device. The argument points to the instance of the component

typedef unsigned short Xuint16

unsigned 16-bit

typedef unsigned long Xuint32

unsigned 32-bit

typedef unsigned char Xuint8

unsigned 8-bit

Function Documentation

```
void XAssert( char * File,  
             int   Line  
            )
```

Implements assert. Currently, it calls a user-defined callback function if one has been set. Then, it potentially enters an infinite loop depending on the value of the XWaitInAssert variable.

Parameters:

File is the name of the filename of the source

Line is the linenumber within File

Returns:

None.

Note:

None.

void XAssertSetCallback(XAssertCallback Routine)

Sets up a callback function to be invoked when an assert occurs. If there was already a callback installed, then it is replaced.

Parameters:

Routine is the callback to be invoked when an assert is taken

Returns:

None.

Note:

This function has no effect if NDEBUG is set

Variable Documentation

unsigned int XAssertStatus()

This variable allows testing to be done easier with asserts. An assert sets this variable such that a driver can evaluate this variable to determine if an assert occurred.

common/v1_00_a/src/xbasic_types.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xbasic_types.h,v 1.8 2003/05/09 14:07:02 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file common/v1_00_a/src/xbasic_types.h
00026 *
00027 * This file contains basic types for Xilinx software IP.  These types do not
00028 * follow the standard naming convention with respect to using the component
00029 * name in front of each name because they are considered to be primitives.
00030 *
00031 * @note
00032 *
00033 * This file contains items which are architecture dependent.
00034 *
00035 * <pre>
00036 * MODIFICATION HISTORY:
00037 *
00038 * Ver      Who      Date      Changes
00039 * -----  -
00040 * 1.00a    rmm      12/14/01  First release
00041 *          rmm      05/09/03  Added "xassert always" macros to rid ourselves of diab
00042 *                               compiler warnings
```



```

00043 * </pre>
00044 *
00045 *****/
00046
00047 #ifndef XBASIC_TYPES_H      /* prevent circular inclusions */
00048 #define XBASIC_TYPES_H      /* by using protection macros */
00049
00050 /***** Include Files *****/
00051
00052
00053 /***** Constant Definitions *****/
00054
00055 /** Xboolean true */
00056 #define XTRUE      1
00057 /** Xboolean false */
00058 #define XFALSE     0
00059
00060 #ifndef NULL
00061 #define NULL 0
00062 #endif
00063 /** Null */
00064 #define XNULL NULL
00065
00066 #define XCOMPONENT_IS_READY      0x11111111 /* component has been initialized
*/
00067 #define XCOMPONENT_IS_STARTED  0x22222222 /* component has been started */
00068
00069 /* the following constants and declarations are for unit test purposes and are
00070 * designed to be used in test applications.
00071 */
00072 #define XTEST_PASSED      0
00073 #define XTEST_FAILED     1
00074
00075 #define XASSERT_NONE     0
00076 #define XASSERT_OCCURRED 1
00077
00078 extern unsigned int XAssertStatus;
00079 extern void XAssert(char *, int);
00080
00081 /***** Type Definitions *****/
00082
00083 /** @name Primitive types
00084 * These primitive types are created for transportability.
00085 * They are dependent upon the target architecture.
00086 * @{
00087 */
00088 typedef unsigned char   Xuint8;      /**< unsigned 8-bit */
00089 typedef char            Xint8;       /**< signed 8-bit */
00090 typedef unsigned short  Xuint16;     /**< unsigned 16-bit */
00091 typedef short          Xint16;      /**< signed 16-bit */
00092 typedef unsigned long   Xuint32;     /**< unsigned 32-bit */

```

```

00093 typedef long          Xint32;      /**< signed 32-bit */
00094 typedef float          Xfloat32;     /**< 32-bit floating point */
00095 typedef double         Xfloat64;     /**< 64-bit double precision floating point
*/
00096 typedef unsigned long  Xboolean;     /**< boolean (XTRUE or XFALSE) */
00097
00098 typedef struct
00099 {
00100     Xuint32 Upper;
00101     Xuint32 Lower;
00102 } Xuint64;
00103
00104 /*@}*/
00105
00106 /**
00107  * This data type defines an interrupt handler for a device.
00108  * The argument points to the instance of the component
00109  */
00110 typedef void (*XInterruptHandler)(void *InstancePtr);
00111
00112 /**
00113  * This data type defines a callback to be invoked when an
00114  * assert occurs. The callback is invoked only when asserts are enabled
00115  */
00116 typedef void (*XAssertCallback)(char* FilenamePtr, int LineNumber);
00117
00118 /***** Macros (Inline Functions) Definitions *****/
00119
00120 /*****
00121 /**
00122  * Return the most significant half of the 64 bit data type.
00123  *
00124  * @param x is the 64 bit word.
00125  *
00126  * @return
00127  *
00128  * The upper 32 bits of the 64 bit word.
00129  *
00130  * @note
00131  *
00132  * None.
00133  *
00134  *****/
00135 #define XUINT64_MSW(x) ((x).Upper)
00136
00137 /*****
00138 /**
00139  * Return the least significant half of the 64 bit data type.
00140  *
00141  * @param x is the 64 bit word.
00142  *

```

```

00143 * @return
00144 *
00145 * The lower 32 bits of the 64 bit word.
00146 *
00147 * @note
00148 *
00149 * None.
00150 *
00151 *****/
00152 #define XUINT64_LSW(x) ((x).Lower)
00153
00154
00155 #ifndef NDEBUG
00156
00157 *****/
00158 /**
00159 * This assert macro is to be used for functions that do not return anything
00160 * (void). This in conjunction with the XWaitInAssert boolean can be used to
00161 * accomodate tests so that asserts which fail allow execution to continue.
00162 *
00163 * @param expression is the expression to evaluate. If it evaluates to false,
00164 * the assert occurs.
00165 *
00166 * @return
00167 *
00168 * Returns void unless the XWaitInAssert variable is true, in which case
00169 * no return is made and an infinite loop is entered.
00170 *
00171 * @note
00172 *
00173 * None.
00174 *
00175 *****/
00176 #define XASSERT_VOID(expression) \
00177 { \
00178     if (expression) \
00179     { \
00180         XAssertStatus = XASSERT_NONE; \
00181     } \
00182     else \
00183     { \
00184         XAssert(__FILE__, __LINE__); \
00185         XAssertStatus = XASSERT_OCCURRED; \
00186         return; \
00187     } \
00188 }
00189
00190 *****/
00191 /**
00192 * This assert macro is to be used for functions that do return a value. This in
00193 * conjunction with the XWaitInAssert boolean can be used to accomodate tests so
00194 * that asserts which fail allow execution to continue.

```



```

00247 /**
00248 * Always assert. This assert macro is to be used for functions that do return
00249 * a value. Use for instances where an assert should always occur.
00250 *
00251 * @return
00252 *
00253 * Returns void unless the XWaitInAssert variable is true, in which case
00254 * no return is made and an infinite loop is entered.
00255 *
00256 * @note
00257 *
00258 * None.
00259 *
00260 *****/
00261 #define XASSERT_NONVOID_ALWAYS() \
00262 { \
00263     XAssert(__FILE__, __LINE__); \
00264     XAssertStatus = XASSERT_OCCURRED; \
00265     return 0; \
00266 }
00267
00268
00269 #else
00270
00271 #define XASSERT_VOID(expression)
00272 #define XASSERT_VOID_ALWAYS()
00273 #define XASSERT_NONVOID(expression)
00274 #define XASSERT_NONVOID_ALWAYS()
00275 #endif
00276
00277 /***** Function Prototypes *****/
00278
00279 void XAssertSetCallback(XAssertCallback Routine);
00280
00281 #endif /* end of protection macro */

```

common/v1_00_a/src/xbasic_types.c File Reference

Detailed Description

This file contains basic functions for Xilinx software IP.

```
#include "xbasic_types.h"
```

Functions

void [XAssert](#) (char *File, int Line)

void [XAssertSetCallback](#) ([XAssertCallback](#) Routine)

Variables

unsigned int [XAssertStatus](#)

[Xboolean](#) [XWaitInAssert](#)

Function Documentation

```
void XAssert( char * File,  
             int   Line  
             )
```

Implements assert. Currently, it calls a user-defined callback function if one has been set. Then, it potentially enters an infinite loop depending on the value of the XWaitInAssert variable.

Parameters:

File is the name of the filename of the source

Line is the linenumber within File

Returns:

None.

Note:

None.

void XAssertSetCallback([XAssertCallback](#) *Routine*)

Sets up a callback function to be invoked when an assert occurs. If there was already a callback installed, then it is replaced.

Parameters:

Routine is the callback to be invoked when an assert is taken

Returns:

None.

Note:

This function has no effect if NDEBUG is set

Variable Documentation

unsigned int XAssertStatus

This variable allows testing to be done easier with asserts. An assert sets this variable such that a driver can evaluate this variable to determine if an assert occurred.

[Xboolean](#) XWaitInAssert

This variable allows the assert functionality to be changed for testing such that it does not wait infinitely. Use the debugger to disable the waiting during testing of asserts.

Generated on 29 May 2003 for Xilinx Device Drivers

common/v1_00_a/src/xstatus.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xstatus.h,v 1.19 2002/07/16 14:18:42 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file common/v1_00_a/src/xstatus.h
00026 *
00027 * This file contains Xilinx software status codes.  Status codes have their
00028 * own data type called XStatus.  These codes are used throughout the Xilinx
00029 * device drivers.
00030 *
00031 *****/
00032
00033 #ifndef XSTATUS_H      /* prevent circular inclusions */
00034 #define XSTATUS_H     /* by using protection macros */
00035
00036 /***** Include Files *****/
00037
00038 #include "xbasic_types.h"
00039
00040 /***** Constant Definitions *****/
00041
00042 /***** Common statuses 0 - 500 *****/
```

```

00043
00044 #define XST_SUCCESS 0L
00045 #define XST_FAILURE 1L
00046 #define XST_DEVICE_NOT_FOUND 2L
00047 #define XST_DEVICE_BLOCK_NOT_FOUND 3L
00048 #define XST_INVALID_VERSION 4L
00049 #define XST_DEVICE_IS_STARTED 5L
00050 #define XST_DEVICE_IS_STOPPED 6L
00051 #define XST_FIFO_ERROR 7L /* an error occurred during an
00052 operation with a FIFO such as
00053 an underrun or overrun, this
00054 error requires the device to
00055 be reset */
00056 #define XST_RESET_ERROR 8L /* an error occurred which requires
00057 the device to be reset */
00058 #define XST_DMA_ERROR 9L /* a DMA error occurred, this error
00059 typically requires the device
00060 using the DMA to be reset */
00061 #define XST_NOT_POLLED 10L /* the device is not configured
for
00062 polled mode operation */
00063 #define XST_FIFO_NO_ROOM 11L /* a FIFO did not have room to put
00064 the specified data into */
00065 #define XST_BUFFER_TOO_SMALL 12L /* the buffer is not large enough
00066 to hold the expected data */
00067 #define XST_NO_DATA 13L /* there was no data available */
00068 #define XST_REGISTER_ERROR 14L /* a register did not contain the
00069 expected value */
00070 #define XST_INVALID_PARAM 15L /* an invalid parameter was passed
00071 into the function */
00072 #define XST_NOT_SGDMA 16L /* the device is not configured for
00073 scatter-gather DMA operation */
00074 #define XST_LOOPBACK_ERROR 17L /* a loopback test failed */
00075 #define XST_NO_CALLBACK 18L /* a callback has not yet been
00076 * registered */
00077 #define XST_NO_FEATURE 19L /* device is not configured with
00078 * the requested feature */
00079 #define XST_NOT_INTERRUPT 20L /* device is not configured for
00080 * interrupt mode operation */
00081 #define XST_DEVICE_BUSY 21L /* device is busy */
00082 #define XST_ERROR_COUNT_MAX 22L /* the error counters of a device
00083 * have maxed out */
00084
00085 /***** Utility Component statuses 401 - 500 *****/
00086
00087 #define XST_MEMTEST_FAILED 401L /* memory test failed */
00088
00089
00090 /***** Common Components statuses 501 - 1000 *****/
00091
00092 /***** Packet Fifo statuses 501 - 510 *****/
00093

```

```

00094 #define XST_PFIFO_LACK_OF_DATA          501L /* not enough data in FIFO */
00095 #define XST_PFIFO_NO_ROOM                502L /* not enough room in FIFO */
00096 #define XST_PFIFO_BAD_REG_VALUE          503L /* self test, a register value
00097                                     was invalid after reset */
00098
00099 /***** DMA statuses 511 - 530 *****/
00100
00101 #define XST_DMA_TRANSFER_ERROR            511L /* self test, DMA transfer
00102                                     failed */
00103 #define XST_DMA_RESET_REGISTER_ERROR      512L /* self test, a register value
00104                                     was invalid after reset */
00105 #define XST_DMA_SG_LIST_EMPTY             513L /* scatter gather list contains
00106                                     no buffer descriptors ready
00107                                     to be processed */
00108 #define XST_DMA_SG_IS_STARTED             514L /* scatter gather not stopped */
00109 #define XST_DMA_SG_IS_STOPPED            515L /* scatter gather not running */
00110 #define XST_DMA_SG_LIST_FULL              517L /* all the buffer descriptors of
00111                                     the scatter gather list are
00112                                     being used */
00113 #define XST_DMA_SG_BD_LOCKED              518L /* the scatter gather buffer
00114                                     descriptor which is to be
00115                                     copied over in the scatter
00116                                     list is locked */
00117 #define XST_DMA_SG_NOTHING_TO_COMMIT      519L /* no buffer descriptors have been
00118                                     put into the scatter gather
00119                                     list to be committed */
00120 #define XST_DMA_SG_COUNT_EXCEEDED         521L /* the packet count threshold
00121                                     specified was larger than the
00122                                     total # of buffer descriptors
00123                                     in the scatter gather list */
00124 #define XST_DMA_SG_LIST_EXISTS            522L /* the scatter gather list has
00125                                     already been created */
00126 #define XST_DMA_SG_NO_LIST                523L /* no scatter gather list has
00127                                     been created */
00128 #define XST_DMA_SG_BD_NOT_COMMITTED       524L /* the buffer descriptor which was
00129                                     being started was not committed
00130                                     to the list */
00131 #define XST_DMA_SG_NO_DATA                525L /* the buffer descriptor to start
00132                                     has already been used by the
00133                                     hardware so it can't be reused
00134                                     */
00135
00136 /***** IPIF statuses 531 - 550 *****/
00137
00138 #define XST_IPIF_REG_WIDTH_ERROR          531L /* an invalid register width
00139                                     was passed into the function */
00140 #define XST_IPIF_RESET_REGISTER_ERROR      532L /* the value of a register at
00141                                     reset was not valid */
00142 #define XST_IPIF_DEVICE_STATUS_ERROR      533L /* a write to the device interrupt
00143                                     status register did not read
00144                                     back correctly */
00145 #define XST_IPIF_DEVICE_ACK_ERROR         534L /* the device interrupt status

```

```

00146                                     register did not reset when
00147                                     acked */
00148 #define XST_IPIF_DEVICE_ENABLE_ERROR    535L /* the device interrupt enable
00149                                     register was not updated when
00150                                     other registers changed */
00151 #define XST_IPIF_IP_STATUS_ERROR        536L /* a write to the IP interrupt
00152                                     status register did not read
00153                                     back correctly */
00154 #define XST_IPIF_IP_ACK_ERROR           537L /* the IP interrupt status
register
00155                                     did not reset when acked */
00156 #define XST_IPIF_IP_ENABLE_ERROR        538L /* IP interrupt enable register
was
00157                                     not updated correctly when
other
00158                                     registers changed */
00159 #define XST_IPIF_DEVICE_PENDING_ERROR    539L /* The device interrupt pending
00160                                     register did not indicate the
00161                                     expected value */
00162 #define XST_IPIF_DEVICE_ID_ERROR        540L /* The device interrupt ID
register
00163                                     did not indicate the expected
00164                                     value */
00165
00166 /***** Device specific statuses 1001 - 4095 *****/
00167
00168 /***** Ethernet statuses 1001 - 1050 *****/
00169
00170 #define XST_EMAC_MEMORY_SIZE_ERROR      1001L /* Memory space is not big enough
00171                                     * to hold the minimum number of
00172                                     * buffers or descriptors */
00173 #define XST_EMAC_MEMORY_ALLOC_ERROR     1002L /* Memory allocation failed */
00174 #define XST_EMAC_MII_READ_ERROR         1003L /* MII read error */
00175 #define XST_EMAC_MII_BUSY               1004L /* An MII operation is in progress */
00176 #define XST_EMAC_OUT_OF_BUFFERS        1005L /* Adapter is out of buffers */
00177 #define XST_EMAC_PARSE_ERROR            1006L /* Invalid adapter init string */
00178 #define XST_EMAC_COLLISION_ERROR        1007L /* Excess deferral or late
00179                                     * collision on polled send */
00180
00181 /***** UART statuses 1051 - 1075 *****/
00182 #define XST_UART
00183
00184 #define XST_UART_INIT_ERROR             1051L
00185 #define XST_UART_START_ERROR            1052L
00186 #define XST_UART_CONFIG_ERROR           1053L
00187 #define XST_UART_TEST_FAIL              1054L
00188 #define XST_UART_BAUD_ERROR             1055L
00189 #define XST_UART_BAUD_RANGE             1056L
00190
00191
00192 /***** IIC statuses 1076 - 1100 *****/
00193

```

```

00194 #define XST_IIC_SELFTEST_FAILED          1076 /* self test failed */
00195 #define XST_IIC_BUS_BUSY                 1077 /* bus found busy */
00196 #define XST_IIC_GENERAL_CALL_ADDRESS     1078 /* mastersend attempted with */
00197                                           /* general call address */
00198 #define XST_IIC_STAND_REG_RESET_ERROR     1079 /* A non parameterizable reg */
00199                                           /* value after reset not valid */
00200 #define XST_IIC_TX_FIFO_REG_RESET_ERROR   1080 /* Tx fifo included in design */
00201                                           /* value after reset not valid */
00202 #define XST_IIC_RX_FIFO_REG_RESET_ERROR   1081 /* Rx fifo included in design */
00203                                           /* value after reset not valid */
00204 #define XST_IIC_TBA_REG_RESET_ERROR       1082 /* 10 bit addr incl in design */
00205                                           /* value after reset not valid */
00206 #define XST_IIC_CR_READBACK_ERROR        1083 /* Read of the control register*/
00207                                           /* didn't return value written */
00208 #define XST_IIC_DTR_READBACK_ERROR       1084 /* Read of the data Tx reg */
00209                                           /* didn't return value written */
00210 #define XST_IIC_DRR_READBACK_ERROR       1085 /* Read of the data Receive reg*/
00211                                           /* didn't return value written */
00212 #define XST_IIC_ADR_READBACK_ERROR       1086 /* Read of the data Tx reg */
00213                                           /* didn't return value written */
00214 #define XST_IIC_TBA_READBACK_ERROR       1087 /* Read of the 10 bit addr reg */
00215                                           /* didn't return written value */
00216 #define XST_IIC_NOT_SLAVE                 1088 /* The device isn't a slave */
00217
00218 /***** ATMC statuses 1101 - 1125 *****/
00219
00220 #define XST_ATMC_ERROR_COUNT_MAX          1101L /* the error counters in the ATM
00221                                           controller hit the max value
00222                                           which requires the statistics
00223                                           to be cleared */
00224
00225 /***** Flash statuses 1126 - 1150 *****/
00226
00227 #define XST_FLASH_BUSY                    1126L /* Flash is erasing or programming
*/
00228 #define XST_FLASH_READY                   1127L /* Flash is ready for commands */
00229 #define XST_FLASH_ERROR                   1128L /* Flash had detected an internal
00230                                           error. Use XFlash_DeviceControl
00231                                           to retrieve device specific
codes */
00232 #define XST_FLASH_ERASE_SUSPENDED         1129L /* Flash is in suspended erase
state */
00233 #define XST_FLASH_WRITE_SUSPENDED        1130L /* Flash is in suspended write
state */
00234 #define XST_FLASH_PART_NOT_SUPPORTED     1131L /* Flash type not supported by
00235                                           driver */
00236 #define XST_FLASH_NOT_SUPPORTED          1132L /* Operation not supported */
00237 #define XST_FLASH_TOO_MANY_REGIONS      1133L /* Too many erase regions */
00238 #define XST_FLASH_TIMEOUT_ERROR          1134L /* Programming or erase operation
00239                                           aborted due to a timeout */
00240 #define XST_FLASH_ADDRESS_ERROR          1135L /* Accessed flash outside its
00241                                           addressable range */
00242 #define XST_FLASH_ALIGNMENT_ERROR        1136L /* Write alignment error */

```

```

00243 #define XST_FLASH_BLOCKING_CALL_ERROR 1137L /* Couldn't return immediately from
00244 write/erase function with
00245 XFL_NON_BLOCKING_WRITE/ERASE
00246 option cleared */
00247 #define XST_FLASH_CFI_QUERY_ERROR 1138L /* Failed to query the device */
00248
00249 /***** SPI statuses 1151 - 1175 *****/
00250
00251 #define XST_SPI_MODE_FAULT 1151 /* master was selected as slave */
00252 #define XST_SPI_TRANSFER_DONE 1152 /* data transfer is complete */
00253 #define XST_SPI_TRANSMIT_UNDERRUN 1153 /* slave underruns transmit register
*/
00254 #define XST_SPI_RECEIVE_OVERRUN 1154 /* device overruns receive register */
00255 #define XST_SPI_NO_SLAVE 1155 /* no slave has been selected yet */
00256 #define XST_SPI_TOO_MANY_SLAVES 1156 /* more than one slave is being
00257 * selected */
00258 #define XST_SPI_NOT_MASTER 1157 /* operation is valid only as master
*/
00259 #define XST_SPI_SLAVE_ONLY 1158 /* device is configured as slave-only
*/
00260 #define XST_SPI_SLAVE_MODE_FAULT 1159 /* slave was selected while disabled
*/
00261
00262 /***** OPB Arbiter statuses 1176 - 1200 *****/
00263
00264 #define XST_OPBARB_INVALID_PRIORITY 1176 /* the priority registers have either
00265 * one master assigned to two or more
00266 * priorities, or one master not
00267 * assigned to any priority
00268 */
00269 #define XST_OPBARB_NOT_SUSPENDED 1177 /* an attempt was made to modify the
00270 * priority levels without first
00271 * suspending the use of priority
00272 * levels
00273 */
00274 #define XST_OPBARB_PARK_NOT_ENABLED 1178 /* bus parking by id was enabled but
00275 * bus parking was not enabled
00276 */
00277 #define XST_OPBARB_NOT_FIXED_PRIORITY 1179 /* the arbiter must be in fixed
00278 * priority mode to allow the
00279 * priorities to be changed
00280 */
00281
00282 /***** Intc statuses 1201 - 1225 *****/
00283
00284 #define XST_INTC_FAIL_SELFTEST 1201 /* self test failed */
00285 #define XST_INTC_CONNECT_ERROR 1202 /* interrupt already in use */
00286
00287 /***** TmrCtr statuses 1226 - 1250 *****/
00288
00289 #define XST_TMRCTR_TIMER_FAILED 1226 /* self test failed */
00290

```

```

00291 /***** WdtTb statuses 1251 - 1275 *****/
00292
00293 #define XST_WDTTB_TIMER_FAILED      1251L
00294
00295 /***** PlbArb statuses 1276 - 1300 *****/
00296
00297 #define XST_PLBARB_FAIL_SELFTEST    1276L
00298
00299 /***** Plb2Opb statuses 1301 - 1325 *****/
00300
00301 #define XST_PLB2OPB_FAIL_SELFTEST   1301L
00302
00303 /***** Opb2Plb statuses 1326 - 1350 *****/
00304
00305 #define XST_OPB2PLB_FAIL_SELFTEST   1326L
00306
00307 /***** SysAce statuses 1351 - 1360 *****/
00308
00309 #define XST_SYSACE_NO_LOCK           1351L    /* No MPU lock has been granted */
00310
00311 /***** PCI Bridge statuses 1361 - 1375 *****/
00312
00313 #define XST_PCI_INVALID_ADDRESS      1361L
00314
00315 /***** Type Definitions *****/
00316
00317 /**
00318  * The status typedef.
00319  */
00320 typedef Xuint32 XStatus;
00321
00322 /***** Macros (Inline Functions) Definitions *****/
00323
00324
00325 /***** Function Prototypes *****/
00326
00327
00328 #endif                /* end of protection macro */

```

common/v1_00_a/src/xstatus.h File Reference

Detailed Description

This file contains Xilinx software status codes. Status codes have their own data type called XStatus. These codes are used throughout the Xilinx device drivers.

```
#include "xbasic_types.h"
```

[Go to the source code of this file.](#)

Typedefs

```
typedef Xuint32 XStatus
```

Typedef Documentation

```
typedef Xuint32 XStatus
```

The status typedef.

hdlc/v1_00_a/src/xhdlc_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xhdlc_l.h,v 1.4 2002/06/28 18:18:15 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file hdlc/v1_00_a/src/xhdlc_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  High-level driver functions
00029 * are defined in xhdlc.h.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver   Who   Date       Changes
00035 * ----  ---  -
00036 * 1.00b jhl   05/20/02 First release
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XHDLC_L_H /* prevent circular inclusions */
00042 #define XHDLC_L_H /* by using protection macros */
```

```

00043
00044 /***** Include Files *****/
00045
00046 #include "xbasic_types.h"
00047 #include "xio.h"
00048
00049 /***** Constant Definitions *****/
00050
00051 /* Offsets from the base address of the device (IPIF) for the registers of the
00052  * device, the FIFOs and the DMA channels
00053  */
00054 #define XHD_REG_OFFSET          0x1100
00055 #define XHD_PFIFO_TXREG_OFFSET  0x2000
00056 #define XHD_PFIFO_TXDATA_OFFSET 0x2100
00057 #define XHD_PFIFO_RXREG_OFFSET  0x2010
00058 #define XHD_PFIFO_RXDATA_OFFSET 0x2200
00059 #define XHD_DMA_OFFSET          0x2300
00060 #define XHD_DMA_SEND_OFFSET     (XHD_DMA_OFFSET + 0x0) /* DMA send channel */
00061 #define XHD_DMA_RECV_OFFSET     (XHD_DMA_OFFSET + 0x40) /* DMA recv channel */
00062
00063
00064 /* HDLC Register offsets
00065  */
00066 #define XHD_RXCR_OFFSET  XHD_REG_OFFSET + 0x00 /* receive control */
00067 #define XHD_RXSR_OFFSET  XHD_REG_OFFSET + 0x04 /* receive status */
00068 #define XHD_RPLR_OFFSET  XHD_REG_OFFSET + 0x08 /* receive length */
00069 #define XHD_TXCR_OFFSET  XHD_REG_OFFSET + 0x0C /* transmit control */
00070 #define XHD_TXSR_OFFSET  XHD_REG_OFFSET + 0x10 /* transmit status */
00071 #define XHD_TPLR_OFFSET  XHD_REG_OFFSET + 0x14 /* transmit length */
00072 #define XHD_ADDR_OFFSET  XHD_REG_OFFSET + 0x18 /* receive address */
00073 #define XHD_CRCE_OFFSET  XHD_REG_OFFSET + 0x1C /* receive CRC error count */
00074 #define XHD_ABRT_OFFSET  XHD_REG_OFFSET + 0x20 /* receive abort error count */
00075
00076 /*
00077  * Register offsets for the IPIF components
00078  */
00079 #define XHD_IISR_OFFSET          0x20UL          /* Interrupt status */
00080
00081 /*
00082  * Receive (Rx) Control Register bit masks
00083  */
00084 #define XHD_RXCR_LOOP_MASK      0x40 /* local loopback enable, default off */
00085 #define XHD_RXCR_CRC32_MASK     0x20 /* CRC-32, default is 16 CRC-CCITT */
00086 #define XHD_RXCR_ADSIZE16_MASK 0x10 /* 16 bit address recognition, default 8
00087 */
00087 #define XHD_RXCR_ADREC_MASK     0x08 /* address recognition, default off */
00088 #define XHD_RXCR_RMADR_MASK     0x04 /* remove address field, default off */
00089 #define XHD_RXCR_BAMEN_MASK     0x02 /* broadcast address match, default off */
00090 #define XHD_RXCR_RXEN_MASK      0x01 /* receiver enable, default off */
00091
00092 /*
00093  * Receive (Rx) Status Register bit masks, a value of 0 indicates a frame

```

```

00094 * without any errors
00095 */
00096 #define XHD_RXSR_RXFA_MASK      0x08 /* frame aborted */
00097 #define XHD_RXSR_RXFCS_MASK    0x04 /* frame FCS error */
00098 #define XHD_RXSR_RXAE_MASK    0x02 /* frame alignment error */
00099 #define XHD_RXSR_RXBA_MASK    0x01 /* frame broadcast address received */
00100 #define XHD_RXSR_NO_STATUS     0    /* no frame status */
00101
00102 /*
00103 * Transmit (Tx) Control Register bit masks
00104 */
00105 #define XHD_TXCR_DISCRC_MASK    0x08 /* CRC generation disable */
00106 #define XHD_TXCR_TXCRC32_MASK  0x04 /* CRC-32, default is 16 CRC-CCITT */
00107 #define XHD_TXCR_TXFA_MASK     0x02 /* send frame abort sequence */
00108 #define XHD_TXCR_TXEN_MASK     0x01 /* transmitter enable */
00109
00110 /*
00111 * Transmit (Tx) Status Register bit masks
00112 */
00113 #define XHD_TXSR_TXUR_MASK     0x01 /* transmit data underrun */
00114
00115 /*
00116 * IPIF Device Interrupt Enable/Status Register bit masks
00117 */
00118 #define XHD_DIXR_HDLC_MASK     0x04UL /* HDLC device interrupt */
00119 #define XHD_DIXR_SEND_DMA_MASK 0x08UL /* Send DMA interrupt */
00120 #define XHD_DIXR_RECV_DMA_MASK 0x10UL /* Receive DMA interrupt */
00121 #define XHD_DIXR_RECV_FIFO_MASK 0x20UL /* Receive FIFO interrupt */
00122 #define XHD_DIXR_SEND_FIFO_MASK 0x40UL /* Send FIFO interrupt */
00123
00124 /*
00125 * IPIF IP Interrupt Enable/Status Register bit masks.
00126 */
00127 #define XHD_IIXR_DUALPORT_OUR_MASK 0x0400 /* dual port over/underrun */
00128 #define XHD_IIXR_ROLLOVER_MASK   0x0200 /* CRC/aborted counter rollover */
00129 #define XHD_IIXR_RLSFIFO_UNDER_MASK 0x0100 /* rx length/status fifo underrun */
00130 #define XHD_IIXR_XLSFIFO_UNDER_MASK 0x0080 /* tx length/status fifo underrun */
00131 #define XHD_IIXR_XLSFIFO_OVER_MASK 0x0040 /* tx length/status fifo overrun */
00132 #define XHD_IIXR_RLSFIFO_OVER_MASK 0x0020 /* rx length/status fifo overrun */
00133 #define XHD_IIXR_ALIGN_ERROR_MASK 0x0010 /* rx alignment error */
00134 #define XHD_IIXR_FCS_ERROR_MASK  0x0008 /* rx FCS error */
00135 #define XHD_IIXR_RDFIFO_OVER_MASK 0x0004 /* rx data fifo overrun */
00136 #define XHD_IIXR_RECV_DONE_MASK  0x0002 /* rx complete, (or aborted) */
00137 #define XHD_IIXR_XMIT_DONE_MASK   0x0001 /* tx complete */
00138
00139 /***** Type Definitions *****/
00140
00141
00142 /***** Macros (Inline Functions) Definitions *****/
00143
00144 /*****
00145 *
00146 * Low-level driver macros. The list below provides signatures

```

```

00147 * to help the user use the macros.
00148 *
00149 * Xuint32 XHdlc_mReadReg(Xuint32 BaseAddress, int RegOffset)
00150 * void XHdlc_mWriteReg(Xuint32 BaseAddress, int RegOffset, Xuint32 Data)
00151 *
00152 * void XHdlc_mSetRxControlReg(Xuint32 BaseAddress, Xuint32 Data)
00153 * void XHdlc_mSetTxControlReg(Xuint32 BaseAddress, Xuint32 Data)
00154 * void XHdlc_mSetAddressReg(Xuint32 BaseAddress, Xuint32 Address)
00155 *
00156 * void XHdlc_mEnable(Xuint32 BaseAddress)
00157 * void XHdlc_mDisable(Xuint32 BaseAddress)
00158 *
00159 *****/
00160
00161 /*****/
00162 /**
00163 *
00164 * Read the specified HDLC register. A 32 bit read of the register is performed.
00165 *
00166 * @param BaseAddress is the base address of the device
00167 * @param RegOffset is the register offset to be read
00168 *
00169 * @return The 32 bit value of the register
00170 *
00171 * @note None.
00172 *
00173 *****/
00174 #define XHdlc_mReadReg(BaseAddress, RegOffset) \
00175     XIo_In32((BaseAddress) + (RegOffset))
00176
00177 /*****/
00178 /**
00179 *
00180 * Write the specified data to the specified register.
00181 *
00182 * @param BaseAddress is the base address of the device
00183 * @param RegOffset is the register offset to be written
00184 * @param Data is the 32-bit value to write to the register
00185 *
00186 * @return None.
00187 *
00188 * @note None.
00189 *
00190 *****/
00191 #define XHdlc_mWriteReg(BaseAddress, RegOffset, Data) \
00192     XIo_Out32((BaseAddress) + (RegOffset), (Data))
00193
00194 /*****/
00195 /**
00196 *
00197 * Set the contents of the receive control register. Use the XHD_RXCR_*
00198 * constants defined above to create the bit-mask to be written to the register.

```

```

00199 *
00200 * @param BaseAddress is the base address of the device
00201 * @param Mask is the 32 bit value to write to the control register
00202 *
00203 * @return None.
00204 *
00205 * @note None.
00206 *
00207 *****/
00208 #define XHdlc_mSetRxControlReg(BaseAddress, Mask) \
00209     XIo_Out32((BaseAddress) + XHD_RXCR_OFFSET, (Mask))
00210
00211 *****/
00212 /**
00213 *
00214 * Set the contents of the transmit control register. Use the XHD_TXCR_*
00215 * constants defined above to create the bit-mask to be written to the register.
00216 *
00217 * @param BaseAddress is the base address of the device
00218 * @param Mask is the 32 bit value to write to the control register
00219 *
00220 * @return None.
00221 *
00222 * @note None.
00223 *
00224 *****/
00225 #define XHdlc_mSetTxControlReg(BaseAddress, Mask) \
00226     XIo_Out32((BaseAddress) + XHD_TXCR_OFFSET, (Mask))
00227
00228 *****/
00229 /**
00230 *
00231 * Set the address register to be used for address recognition.
00232 *
00233 * @param BaseAddress is the base address of the device
00234 * @param Address is a 16 bit address
00235 *
00236 * @return None.
00237 *
00238 * @note None.
00239 *
00240 *****/
00241 #define XHdlc_mSetAddressReg(BaseAddress, Address) \
00242     XIo_Out32((BaseAddress) + XHD_ADDR_OFFSET, Address)
00243
00244 *****/
00245 /**
00246 *
00247 * Enable the transmitter and receiver. Preserve the contents of the control
00248 * registers.
00249 *
00250 * @param BaseAddress is the base address of the device

```

```

00251 *
00252 * @return    None.
00253 *
00254 * @note      None.
00255 *
00256 *****/
00257 #define XHdlc_mEnable(BaseAddress) \
00258 { \
00259     XIo_Out32((BaseAddress) + XHD_RXCR_OFFSET, \
00260         (XIo_In32((BaseAddress) + XHD_RXCR_OFFSET) | XHD_RXCR_RXEN_MASK)); \
00261     XIo_Out32((BaseAddress) + XHD_TXCR_OFFSET, \
00262         (XIo_In32((BaseAddress) + XHD_TXCR_OFFSET) | XHD_TXCR_TXEN_MASK)); \
00263 }
00264
00265 *****/
00266 /**
00267 *
00268 * Disable the transmitter and receiver. Preserve the contents of the control
00269 * registers.
00270 *
00271 * @param      BaseAddress is the base address of the device
00272 *
00273 * @return     None.
00274 *
00275 * @note      None.
00276 *
00277 *****/
00278 #define XHdlc_mDisable(BaseAddress) \
00279 { \
00280     XIo_Out32((BaseAddress) + XHD_RXCR_OFFSET, \
00281         XIo_In32((BaseAddress) + XHD_RXCR_OFFSET) & ~XHD_RXCR_RXEN_MASK); \
00282     XIo_Out32((BaseAddress) + XHD_TXCR_OFFSET, \
00283         XIo_In32((BaseAddress) + XHD_TXCR_OFFSET) & ~XHD_TXCR_RXEN_MASK); \
00284 }
00285
00286 *****/ Function Prototypes *****/
00287
00288 void XHdlc_SendFrame(Xuint32 BaseAddress, Xuint8 *FramePtr, unsigned
ByteCount);
00289 unsigned XHdlc_RecvFrame(Xuint32 BaseAddress, Xuint8 *FramePtr,
Xuint32 *FrameStatusPtr);
00290
00291
00292 #endif          /* end of protection macro */

```

hdlc/v1_00_a/src/xhdlc_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xhdlc.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	05/20/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XHdlc_mReadReg(BaseAddress, RegOffset)  
#define XHdlc_mWriteReg(BaseAddress, RegOffset, Data)  
#define XHdlc_mSetRxControlReg(BaseAddress, Mask)  
#define XHdlc_mSetTxControlReg(BaseAddress, Mask)  
#define XHdlc_mSetAddressReg(BaseAddress, Address)  
#define XHdlc_mEnable(BaseAddress)  
#define XHdlc_mDisable(BaseAddress)
```

Functions

void [XHdlc_SendFrame](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *FramePtr, unsigned ByteCount)

unsigned [XHdlc_RecvFrame](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *FramePtr, [Xuint32](#) *FrameStatusPtr)

Define Documentation

```
#define XHdlc_mDisable( BaseAddress )
```

Disable the transmitter and receiver. Preserve the contents of the control registers.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XHdlc_mEnable( BaseAddress )
```

Enable the transmitter and receiver. Preserve the contents of the control registers.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.


```
#define XHdlc_mReadReg( BaseAddress,  
                        RegOffset  )
```

Read the specified HDLC register. A 32 bit read of the register is performed.

Parameters:

BaseAddress is the base address of the device

RegOffset is the register offset to be read

Returns:

The 32 bit value of the register

Note:

None.

```
#define XHdlc_mSetAddressReg( BaseAddress,  
                              Address  )
```

Set the address register to be used for address recognition.

Parameters:

BaseAddress is the base address of the device

Address is a 16 bit address

Returns:

None.

Note:

None.

```
#define XHdlc_mSetRxControlReg( BaseAddress,  
                                Mask  )
```

Set the contents of the receive control register. Use the XHD_RXCR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device

Mask is the 32 bit value to write to the control register

Returns:

None.

Note:

None.

```
#define XHdlc_mSetTxControlReg( BaseAddress,  
                               Mask      )
```

Set the contents of the transmit control register. Use the XHD_TXCR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device

Mask is the 32 bit value to write to the control register

Returns:

None.

Note:

None.

```
#define XHdlc_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data      )
```

Write the specified data to the specified register.

Parameters:

BaseAddress is the base address of the device
RegOffset is the register offset to be written
Data is the 32-bit value to write to the register

Returns:

None.

Note:

None.

Function Documentation

```
unsigned XHdlc_RecvFrame( Xuint32 BaseAddress,  
                        Xuint8 * FramePtr,  
                        Xuint32 * FrameStatusPtr  
                        )
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device
FramePtr is a pointer to a 32 bit word-aligned buffer where the frame will be stored.
FrameStatusPtr is a pointer to a frame status that will be valid after this function returns.

Returns:

The size, in bytes, of the frame received.

Note:

None.

```
void XHdlc_SendFrame( Xuint32 BaseAddress,  
                     Xuint8 * FramePtr,  
                     unsigned ByteCount  
                     )
```

Send a HDLC frame. This size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to 32 bit word-aligned frame

ByteCount is the number of bytes in the frame

Returns:

None.

Note:

None.

cpu_ppc405/v1_00_a/src/xio.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xio.h,v 1.8 2003/05/05 21:08:31 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file cpu_ppc405/v1_00_a/src/xio.h
00026 *
00027 * This file contains the interface for the XIo component, which encapsulates
00028 * the Input/Output functions for the PowerPC architecture.
00029 *
00030 * @note
00031 *
00032 * This file contains architecture-dependent items (memory mapped or non memory
00033 * mapped I/O).
00034 *
00035 *****/
00036
00037 #ifndef XIO_H                /* prevent circular inclusions */
00038 #define XIO_H                /* by using protection macros */
00039
00040 /***** Include Files *****/
00041
00042 #include "xbasic_types.h"
```



```

00094
00095 __asm void XIo_EndianSwap16(Xuint16 Source, Xuint16 *DestPtr)
00096 {
00097 % reg Source; reg DestPtr;
00098
00099 sthbrx Source,0,DestPtr
00100 }
00101
00102 __asm void XIo_EndianSwap32(Xuint32 Source, Xuint32 *DestPtr)
00103 {
00104 % reg Source; reg DestPtr;
00105
00106 stwbrx Source,0,DestPtr
00107 }
00108
00109 #else
00110
00111 #define XIo_EndianSwap16(Source, DestPtr) \
00112 {\
00113     Xuint16 src = (Source); \
00114     Xuint16 *destptr = (DestPtr); \
00115     *destptr = src >> 8; \
00116     *destptr |= (src << 8); \
00117 }
00118
00119 #define XIo_EndianSwap32(Source, DestPtr) \
00120 {\
00121     Xuint32 src = (Source); \
00122     Xuint32 *destptr = (DestPtr); \
00123     *destptr = src >> 24; \
00124     *destptr |= ((src >> 8) & 0x0000FF00); \
00125     *destptr |= ((src << 8) & 0x00FF0000); \
00126     *destptr |= ((src << 24) & 0xFF000000); \
00127 }
00128
00129 #endif
00130
00131 #ifdef XLITTLE_ENDIAN
00132 /* little-endian processor */
00133
00134 #define XIo_ToLittleEndian16                XIo_EndianNoop
00135 #define XIo_ToLittleEndian32                XIo_EndianNoop
00136 #define XIo_FromLittleEndian16              XIo_EndianNoop
00137 #define XIo_FromLittleEndian32              XIo_EndianNoop
00138
00139 #define XIo_ToBigEndian16(Source, DestPtr)  XIo_EndianSwap16(Source, DestPtr)
00140 #define XIo_ToBigEndian32(Source, DestPtr)  XIo_EndianSwap32(Source, DestPtr)
00141 #define XIo_FromBigEndian16                 XIo_ToBigEndian16
00142 #define XIo_FromBigEndian32                 XIo_ToBigEndian32
00143
00144 #else
00145 /* big-endian processor */

```

```

00146
00147 #define XIo_ToLittleEndian16(Source, DestPtr) XIo_EndianSwap16(Source, DestPtr)
00148 #define XIo_ToLittleEndian32(Source, DestPtr) XIo_EndianSwap32(Source, DestPtr)
00149 #define XIo_FromLittleEndian16                XIo_ToLittleEndian16
00150 #define XIo_FromLittleEndian32                XIo_ToLittleEndian32
00151
00152 #define XIo_ToBigEndian16                    XIo_EndianNoop
00153 #define XIo_ToBigEndian32                    XIo_EndianNoop
00154 #define XIo_FromBigEndian16                  XIo_EndianNoop
00155 #define XIo_FromBigEndian32                  XIo_EndianNoop
00156
00157 #endif
00158
00159
00160 /***** Function Prototypes *****/
00161
00162 /* The following functions allow the software to be transportable across
00163  * processors which may use memory mapped I/O or I/O which is mapped into a
00164  * separate address space such as X86. The functions are better suited for
00165  * debugging and are therefore the default implementation. Macros can instead
00166  * be used if USE_IO_MACROS is defined.
00167  */
00168 #ifndef USE_IO_MACROS
00169
00170 /* Functions */
00171 Xuint8 XIo_In8(XIo_Address InAddress);
00172 Xuint16 XIo_In16(XIo_Address InAddress);
00173 Xuint32 XIo_In32(XIo_Address InAddress);
00174
00175 void XIo_Out8(XIo_Address OutAddress, Xuint8 Value);
00176 void XIo_Out16(XIo_Address OutAddress, Xuint16 Value);
00177 void XIo_Out32(XIo_Address OutAddress, Xuint32 Value);
00178
00179 #else
00180
00181 /* The following macros allow optimized I/O operations for memory mapped I/O
00182  * Note that the SYNCHRONIZE_IO may be moved by the compiler during
00183  * optimization.
00184  */
00185
00186 #define XIo_In8(InputPtr) (*(volatile Xuint8 *) (InputPtr)); SYNCHRONIZE_IO;
00187 #define XIo_In16(InputPtr) (*(volatile Xuint16 *) (InputPtr)); SYNCHRONIZE_IO;
00188 #define XIo_In32(InputPtr) (*(volatile Xuint32 *) (InputPtr)); SYNCHRONIZE_IO;
00189
00190 #define XIo_Out8(OutputPtr, Value) \
00191     { (*(volatile Xuint8 *) (OutputPtr) = Value); SYNCHRONIZE_IO; }
00192 #define XIo_Out16(OutputPtr, Value) \
00193     { (*(volatile Xuint16 *) (OutputPtr) = Value); SYNCHRONIZE_IO; }
00194 #define XIo_Out32(OutputPtr, Value) \
00195     { (*(volatile Xuint32 *) (OutputPtr) = Value); SYNCHRONIZE_IO; }
00196
00197 #endif

```



```
00198
00199 /* The following functions handle IO addresses where data must be swapped
00200  * They cannot be implemented as macros
00201  */
00202 Xuint16 XIo_InSwap16(XIo_Address InAddress);
00203 Xuint32 XIo_InSwap32(XIo_Address InAddress);
00204 void XIo_OutSwap16(XIo_Address OutAddress, Xuint16 Value);
00205 void XIo_OutSwap32(XIo_Address OutAddress, Xuint32 Value);
00206
00207 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

cpu_ppc405/v1_00_a/src/xio.h File Reference

Detailed Description

This file contains the interface for the XIo component, which encapsulates the Input/Output functions for the PowerPC architecture.

Note:

This file contains architecture-dependent items (memory mapped or non memory mapped I/O).

```
#include "xbasic_types.h"
```

[Go to the source code of this file.](#)

Typedefs

```
typedef Xuint32 XIo\_Address
```

Functions

```
Xuint8 XIo\_In8 (XIo\_Address InAddress)  
Xuint16 XIo\_In16 (XIo\_Address InAddress)  
Xuint32 XIo\_In32 (XIo\_Address InAddress)  
void XIo\_Out8 (XIo\_Address OutAddress, Xuint8 Value)  
void XIo\_Out16 (XIo\_Address OutAddress, Xuint16 Value)
```

void **XIo_Out32** (**XIo_Address** OutAddress, **Xuint32** Value)

Xuint16 **XIo_InSwap16** (**XIo_Address** InAddress)

Xuint32 **XIo_InSwap32** (**XIo_Address** InAddress)

void **XIo_OutSwap16** (**XIo_Address** OutAddress, **Xuint16** Value)

void **XIo_OutSwap32** (**XIo_Address** OutAddress, **Xuint32** Value)

Typedef Documentation

typedef **Xuint32** **XIo_Address**

Typedef for an I/O address. Typically correlates to the width of the address bus.

Function Documentation

Xuint16 **XIo_In16**(**XIo_Address** *InAddress*)

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint32 **XIo_In32**(**XIo_Address** *InAddress*)

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint8 XIo_In8(XIo_Address *InAddress*)

Performs an input operation for an 8-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint16 XIo_InSwap16(XIo_Address *InAddress*)

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

Xuint32 XIo_InSwap32(XIo_Address InAddress)

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

```
void XIo_Out16( XIo_Address OutAddress,  
               Xuint16      Value  
               )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void XIo_Out32( XIo_Address OutAddress,  
               Xuint32      Value  
               )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void XIo_Out8( XIo_Address OutAddress,  
              Xuint8      Value  
              )
```

Performs an output operation for an 8-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void XIo_OutSwap16( XIo_Address  OutAddress,  
                   Xuint16      Value  
                   )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void XIo_OutSwap32( XIo_Address  OutAddress,  
                   Xuint32      Value  
                   )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

cpu/v1_00_a/src/xio.c File Reference

Detailed Description

Contains I/O functions for memory-mapped or non-memory-mapped I/O architectures. These functions encapsulate generic CPU I/O requirements.

Note:

This file may contain architecture-dependent code.

```
#include "xio.h"  
#include "xbasic_types.h"
```

Functions

```
void XIo_EndianSwap16( Xuint16 Source, Xuint16 *DestPtr)  
void XIo_EndianSwap32( Xuint32 Source, Xuint32 *DestPtr)
```

Function Documentation

```
void XIo_EndianSwap16( Xuint16  Source,  
                      Xuint16 * DestPtr  
                      )
```

Performs a 16-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

```
void XIo_EndianSwap32( Xuint32 Source,  
                      Xuint32 * DestPtr  
                      )
```

Performs a 32-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

cpu/v1_00_a/src/xio.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xio.h,v 1.6 2002/06/28 17:57:05 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file cpu/v1_00_a/src/xio.h
00026 *
00027 * This file contains the interface for the XIo component, which encapsulates
00028 * the Input/Output functions for processors that do not require any special
00029 * I/O handling.
00030 *
00031 * @notes
00032 *
00033 * This file may contain architecture-dependent items (memory-mapped or
00034 * non-memory-mapped I/O).
00035 *
00036 *****/
00037
00038 #ifndef XIO_H                /* prevent circular inclusions */
00039 #define XIO_H                /* by using protection macros */
00040
00041 /***** Include Files *****/
00042
```

```

00043 #include "xbasic_types.h"
00044
00045 /***** Constant Definitions *****/
00046
00047
00048 /***** Type Definitions *****/
00049
00050 /**
00051  * Typedef for an I/O address. Typically correlates to the width of the
00052  * address bus.
00053  */
00054 typedef Xuint32 XIo_Address;
00055
00056 /***** Macros (Inline Functions) Definitions *****/
00057
00058 /*
00059  * The following macros allow optimized I/O operations for memory mapped I/O.
00060  * It should be noted that macros cannot be used if synchronization of the I/O
00061  * operation is needed as it will likely break some code.
00062  */
00063
00064 /*****
00065  /**
00066  *
00067  * Performs an input operation for an 8-bit memory location by reading from the
00068  * specified address and returning the value read from that address.
00069  *
00070  * @param InputPtr contains the address to perform the input operation at.
00071  *
00072  * @return The value read from the specified input address.
00073  *
00074  *****/
00075 #define XIo_In8(InputPtr) (*(volatile Xuint8 *) (InputPtr))
00076
00077 /*****
00078  /**
00079  *
00080  * Performs an input operation for a 16-bit memory location by reading from the
00081  * specified address and returning the value read from that address.
00082  *
00083  * @param InputPtr contains the address to perform the input operation at.
00084  *
00085  * @return The value read from the specified input address.
00086  *
00087  *****/
00088 #define XIo_In16(InputPtr) (*(volatile Xuint16 *) (InputPtr))
00089
00090 /*****
00091  /**
00092  *
00093  * Performs an input operation for a 32-bit memory location by reading from the
00094  * specified address and returning the value read from that address.

```

```

00095 *
00096 * @param InputPtr contains the address to perform the input operation at.
00097 *
00098 * @return The value read from the specified input address.
00099 *
00100 *****/
00101 #define XIo_In32(InputPtr) (*(volatile Xuint32*)(InputPtr))
00102
00103
00104 *****/
00105 /**
00106 *
00107 * Performs an output operation for an 8-bit memory location by writing the
00108 * specified value to the the specified address.
00109 *
00110 * @param OutputPtr contains the address to perform the output operation at.
00111 * @param Value contains the value to be output at the specified address.
00112 *
00113 * @return None.
00114 *
00115 *****/
00116 #define XIo_Out8(OutputPtr, Value) \
00117     { (*(volatile Xuint8*)(OutputPtr) = Value); }
00118
00119 *****/
00120 /**
00121 *
00122 * Performs an output operation for a 16-bit memory location by writing the
00123 * specified value to the the specified address.
00124 *
00125 * @param OutputPtr contains the address to perform the output operation at.
00126 * @param Value contains the value to be output at the specified address.
00127 *
00128 * @return None.
00129 *
00130 *****/
00131 #define XIo_Out16(OutputPtr, Value) \
00132     { (*(volatile Xuint16*)(OutputPtr) = Value); }
00133
00134 *****/
00135 /**
00136 *
00137 * Performs an output operation for a 32-bit memory location by writing the
00138 * specified value to the the specified address.
00139 *
00140 * @param OutputPtr contains the address to perform the output operation at.
00141 * @param Value contains the value to be output at the specified address.
00142 *
00143 * @return None.
00144 *
00145 *****/
00146 #define XIo_Out32(OutputPtr, Value) \

```

```

00147     { (*(volatile Xuint32 *)(OutputPtr) = Value); }
00148
00149
00150 /* The following macros allow the software to be transportable across
00151 * processors which use big or little endian memory models.
00152 *
00153 * Defined first is a no-op endian conversion macro. This macro is not to
00154 * be used directly by software. Instead, the XIo_To/FromLittleEndianXX and
00155 * XIo_To/FromBigEndianXX macros below are to be used to allow the endian
00156 * conversion to only be performed when necessary
00157 */
00158 #define XIo_EndianNoop(Source, Destination)    (*DestPtr = Source)
00159
00160 #ifdef XLITTLE_ENDIAN
00161
00162 #define XIo_ToLittleEndian16                    XIo_EndianNoop
00163 #define XIo_ToLittleEndian32                    XIo_EndianNoop
00164 #define XIo_FromLittleEndian16                  XIo_EndianNoop
00165 #define XIo_FromLittleEndian32                  XIo_EndianNoop
00166
00167 #define XIo_ToBigEndian16(Source, DestPtr)     XIo_EndianSwap16(Source, DestPtr)
00168 #define XIo_ToBigEndian32(Source, DestPtr)     XIo_EndianSwap32(Source, DestPtr)
00169 #define XIo_FromBigEndian16                    XIo_ToBigEndian16
00170 #define XIo_FromBigEndian32                    XIo_ToBigEndian32
00171
00172 #else
00173
00174 #define XIo_ToLittleEndian16(Source, DestPtr) XIo_EndianSwap16(Source, DestPtr)
00175 #define XIo_ToLittleEndian32(Source, DestPtr) XIo_EndianSwap32(Source, DestPtr)
00176 #define XIo_FromLittleEndian16                XIo_ToLittleEndian16
00177 #define XIo_FromLittleEndian32                XIo_ToLittleEndian32
00178
00179 #define XIo_ToBigEndian16                      XIo_EndianNoop
00180 #define XIo_ToBigEndian32                      XIo_EndianNoop
00181 #define XIo_FromBigEndian16                    XIo_EndianNoop
00182 #define XIo_FromBigEndian32                    XIo_EndianNoop
00183
00184 #endif
00185
00186 /***** Function Prototypes *****/
00187
00188 /* The following functions allow the software to be transportable across
00189 * processors which use big or little endian memory models. These functions
00190 * should not be directly called, but the macros XIo_To/FromLittleEndianXX and
00191 * XIo_To/FromBigEndianXX should be used to allow the endian conversion to only
00192 * be performed when necessary.
00193 */
00194 void XIo_EndianSwap16(Xuint16 Source, Xuint16* DestPtr);
00195 void XIo_EndianSwap32(Xuint32 Source, Xuint32* DestPtr);
00196
00197
00198 #endif          /* end of protection macro */

```


cpu/v1_00_a/src/xio.h File Reference

Detailed Description

This file contains the interface for the XIo component, which encapsulates the Input/Output functions for processors that do not require any special I/O handling.

Note:

This file may contain architecture-dependent items (memory-mapped or non-memory-mapped I/O).

```
#include "xbasic_types.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XIo_In8(InputPtr)  
#define XIo_In16(InputPtr)  
#define XIo_In32(InputPtr)  
#define XIo_Out8(OutputPtr, Value)  
#define XIo_Out16(OutputPtr, Value)  
#define XIo_Out32(OutputPtr, Value)
```

Typedefs

```
typedef Xuint32 XIo_Address
```


Functions

void **XIo_EndianSwap16** (**Xuint16** Source, **Xuint16** *DestPtr)

void **XIo_EndianSwap32** (**Xuint32** Source, **Xuint32** *DestPtr)

Define Documentation

#define XIo_In16(InputPtr)

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InputPtr contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

#define XIo_In32(InputPtr)

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InputPtr contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

#define XIo_In8(InputPtr)

Performs an input operation for an 8-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InputPtr contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

```
#define XIo_Out16( OutputPtr,  
                Value      )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address.

Parameters:

OutputPtr contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

```
#define XIo_Out32( OutputPtr,  
                Value      )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address.

Parameters:

OutputPtr contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

```
#define XIo_Out8( OutputPtr,  
                Value      )
```

Performs an output operation for an 8-bit memory location by writing the specified value to the the specified address.

Parameters:

OutputPtr contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Typedef Documentation

```
typedef Xuint32 XIo_Address
```

Typedef for an I/O address. Typically correlates to the width of the address bus.

Function Documentation

```
void XIo_EndianSwap16( Xuint16 Source,  
                     Xuint16 * DestPtr  
                     )
```

Performs a 16-bit endian converion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

```
void XIo_EndianSwap32( Xuint32 Source,  
                      Xuint32 * DestPtr  
                      )
```

Performs a 32-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

cpu_ppc405/v1_00_a/src/xio.c File Reference

Detailed Description

Contains I/O functions for memory-mapped or non-memory-mapped I/O architectures. These functions encapsulate PowerPC architecture-specific I/O requirements.

Note:

This file contains architecture-dependent code.

The order of the SYNCHRONIZE_IO and the read or write operation is important. For the Read operation, all I/O needs to complete prior to the desired read to insure valid data from the address. The PPC is a weakly ordered I/O model and reads can and will occur prior to writes and the SYNCHRONIZE_IO ensures that any writes occur prior to the read. For the Write operation the SYNCHRONIZE_IO occurs after the desired write to ensure that the address is updated with the new value prior to any subsequent read.

```
#include "xio.h"  
#include "xbasic_types.h"
```

Functions

[Xuint8 XIo_In8 \(XIo_Address InAddress\)](#)
[Xuint16 XIo_In16 \(XIo_Address InAddress\)](#)
[Xuint32 XIo_In32 \(XIo_Address InAddress\)](#)
[Xuint16 XIo_InSwap16 \(XIo_Address InAddress\)](#)
[Xuint32 XIo_InSwap32 \(XIo_Address InAddress\)](#)

```
void XIo_Out8 (XIo_Address OutAddress, Xuint8 Value)
void XIo_Out16 (XIo_Address OutAddress, Xuint16 Value)
void XIo_Out32 (XIo_Address OutAddress, Xuint32 Value)
void XIo_EndianSwap16OLD (Xuint16 Source, Xuint16 *DestPtr)
void XIo_EndianSwap32OLD (Xuint32 Source, Xuint32 *DestPtr)
void XIo_OutSwap16 (XIo_Address OutAddress, Xuint16 Value)
void XIo_OutSwap32 (XIo_Address OutAddress, Xuint32 Value)
```

Function Documentation

```
void XIo_EndianSwap16OLD( Xuint16  Source,
                          Xuint16 * DestPtr
                          )
```

Performs a 16-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

```
void XIo_EndianSwap32OLD( Xuint32  Source,
                          Xuint32 * DestPtr
                          )
```

Performs a 32-bit endian conversion.

Parameters:

Source contains the value to be converted.

DestPtr contains a pointer to the location to put the converted value.

Returns:

None.

Note:

None.

Xuint16 XIo_In16(XIo_Address InAddress)

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint32 XIo_In32(XIo_Address InAddress)

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint8 XIo_In8(XIo_Address *InAddress*)

Performs an input operation for an 8-bit memory location by reading from the specified address and returning the value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The value read from the specified input address.

Note:

None.

Xuint16 XIo_InSwap16(XIo_Address *InAddress*)

Performs an input operation for a 16-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

Xuint32 XIo_InSwap32(XIo_Address *InAddress*)

Performs an input operation for a 32-bit memory location by reading from the specified address and returning the byte-swapped value read from that address.

Parameters:

InAddress contains the address to perform the input operation at.

Returns:

The byte-swapped value read from the specified input address.

Note:

None.

```
void XIo_Out16( XIo_Address OutAddress,  
               Xuint16     Value  
               )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void XIo_Out32( XIo_Address OutAddress,  
               Xuint32    Value  
               )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void XIo_Out8( XIo_Address OutAddress,  
              Xuint8    Value  
              )
```

Performs an output operation for an 8-bit memory location by writing the specified value to the the specified address.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void XIo_OutSwap16( XIo_Address  OutAddress,  
                   Xuint16      Value  
                   )
```

Performs an output operation for a 16-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

```
void XIo_OutSwap32( XIo_Address  OutAddress,  
                   Xuint32      Value  
                   )
```

Performs an output operation for a 32-bit memory location by writing the specified value to the the specified address. The value is byte-swapped before being written.

Parameters:

OutAddress contains the address to perform the output operation at.

Value contains the value to be output at the specified address.

Returns:

None.

Note:

None.

hdlc/v1_00_a/src/xhdlc_l.c File Reference

Detailed Description

This file contains low-level polled functions to send and receive HDLC frames.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	jhl	05/20/02	First release

```
#include "xhdlc_1.h"  
#include "xio.h"
```

Functions

```
void XHdlc_SendFrame (Xuint32 BaseAddress, Xuint8 *FramePtr, unsigned ByteCount)  
unsigned XHdlc_RecvFrame (Xuint32 BaseAddress, Xuint8 *FramePtr, Xuint32 *FrameStatusPtr)
```

Function Documentation

```
unsigned XHdlc_RecvFrame( Xuint32  BaseAddress,  
                          Xuint8 *  FramePtr,  
                          Xuint32 *  FrameStatusPtr  
                          )
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a 32 bit word-aligned buffer where the frame will be stored.

FrameStatusPtr is a pointer to a frame status that will be valid after this function returns.

Returns:

The size, in bytes, of the frame received.

Note:

None.

```
void XHdlc_SendFrame( Xuint32 BaseAddress,  
                     Xuint8 * FramePtr,  
                     unsigned ByteCount  
                     )
```

Send a HDLC frame. This size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to 32 bit word-aligned frame

ByteCount is the number of bytes in the frame

Returns:

None.

Note:

None.

common/v1_00_a/src/xparameters.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xparameters.h,v 1.57 2003/05/09 14:15:47 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file common/v1_00_a/src/xparameters.h
00026 *
00027 * This file contains system parameters for the Xilinx device driver
environment.
00028 * It is a representation of the system in that it contains the number of each
00029 * device in the system as well as the parameters and memory map for each
00030 * device.  The user can view this file to obtain a summary of the devices in
00031 * their system and the device parameters.
00032 *
00033 * This file may be automatically generated by a design tool such as System
00034 * Generator.
00035 *
00036 *****/
00037
00038 /***** Include Files *****/
00039
00040 #ifndef XPARAMETERS_H          /* prevent circular inclusions */
00041 #define XPARAMETERS_H         /* by using protection macros */
```

```

00042
00043 /* unifying driver changes
00044
00045 added XPAR_INTC_0_ACK_BEFORE, XPAR_INTC_1_ACK_BEFORE
00046 changed XPAR_INTC_MAX_ID to XPAR_INTC_MAX_NUM_INTR_INPUTS
00047 deleted XPAR_INTC_0_MAX_ID, XPAR_INTC_1_MAX_ID
00048
00049 */
00050
00051 /***** Constant Definitions *****/
00052
00053 /*
00054  * The following constants are for each device.
00055  *
00056  * An instance must exist for each physical device that exists in the system.
00057  * The device IDs in the following constants are unique between all devices to
00058  * allow device IDs to be searched in the future.
00059  */
00060
00061 /*****
00062  *
00063  * SystemLevel defines. These constants are for devices that do not require
00064  * a device driver. Examples of these types of devices include volatile RAM
00065  * devices.
00066  */
00067 #define XPAR_ZBT_NUM_INSTANCES      1
00068 #define XPAR_ZBT_0_BASE              0x00000000
00069 #define XPAR_ZBT_0_SIZE              0x00100000
00070
00071 #define XPAR_SRAM_NUM_INSTANCES     1
00072 #define XPAR_SRAM_0_BASE             0x00100000
00073 #define XPAR_SRAM_0_SIZE             0x00200000
00074
00075 #define XPAR_DDR_NUM_INSTANCES      1
00076 #define XPAR_DDR_0_BASE              0xF0000000
00077 #define XPAR_DDR_0_SIZE             0x01000000
00078
00079 #define XPAR_CORE_CLOCK_FREQ_HZ     12500000
00080
00081 #define XPAR_CPU_PPC405_CORE_CLOCK_FREQ_HZ  XPAR_CORE_CLOCK_FREQ_HZ
00082
00083 /*****
00084  *
00085  * Interrupt Controller (Intc) defines.
00086  * DeviceID starts at 0
00087  */
00088 #define XPAR_XINTC_NUM_INSTANCES     2          /* Number of instances */
00089 #define XPAR_INTC_MAX_NUM_INTR_INPUTS 31       /* max # inputs of all */
00090 #define XPAR_INTC_SINGLE_BASEADDR   0x70800000 /* low level driver base */
00091 #define XPAR_INTC_SINGLE_ACK_BEFORE  0xFFFF00FF /* low level driver */
00092
00093 #define XPAR_INTC_0_DEVICE_ID        1          /* Device ID for instance */
00094 #define XPAR_INTC_0_ACK_BEFORE       0xFFFF00FF /* Ack timing, before/after

```



```

*/
00095 #define XPAR_INTC_0_BASEADDR          0x70800000 /* Register base address */
00096
00097 #define XPAR_INTC_0_UARTLITE_0_VEC_ID  4           /* Interrupt source for
vector */
00098 #define XPAR_INTC_0_WDTTB_0_VEC_ID    5           /* Interrupt source for
vector */
00099 #define XPAR_INTC_0_WD_0_VEC_ID       6           /* Interrupt source for
vector */
00100 #define XPAR_INTC_0_TMRCTR_0_VEC_ID   7           /* Interrupt source for
vector */
00101 #define XPAR_INTC_0_SPI_0_VEC_ID      11          /* Interrupt source for
vector */
00102 #define XPAR_INTC_0_IIC_0_VEC_ID      12          /* Interrupt source for
vector */
00103 #define XPAR_INTC_0_UARTNS550_0_VEC_ID 13          /* Interrupt source for
vector */
00104 #define XPAR_INTC_0_UARTNS550_1_VEC_ID 14          /* Interrupt source for
vector */
00105 #define XPAR_INTC_0_EMAC_0_VEC_ID      15          /* Interrupt source for
vector */
00106
00107 #define XPAR_INTC_1_DEVICE_ID            2           /* Device ID for instance */
00108 #define XPAR_INTC_1_ACK_BEFORE          0xFFFF00FF /* Ack timing, before/after
*/
00109 #define XPAR_INTC_1_BASEADDR            0x70800020 /* Register base address */
00110
00111 #define XPAR_INTC_1_OPB_TO_PLB_ERR_VEC_ID 0           /* Interrupt source for
vector */
00112 #define XPAR_INTC_1_PLB_TO_OPB_ERR_VEC_ID 1          /* Interrupt source for
vector */
00113
00114 /*****
00115  *
00116  * Ethernet 10/100 MAC defines.
00117  * DeviceID starts at 10
00118  */
00119 #define XPAR_XEMAC_NUM_INSTANCES         1           /* Number of instances */
00120
00121 #define XPAR_EMAC_0_DEVICE_ID            10          /* Device ID for instance */
00122 #define XPAR_EMAC_0_BASEADDR             0x60000000 /* Device base address */
00123 #define XPAR_EMAC_0_DMA_PRESENT          XFALSE     /* Does device have DMA? */
00124 #define XPAR_EMAC_0_ERR_COUNT_EXIST      XTRUE      /* Does device have counters?
*/
00125 #define XPAR_EMAC_0_MII_EXIST            XTRUE      /* Does device support MII? */
00126
00127 /*****
00128  *
00129  * NS16550 UART defines.
00130  * DeviceID starts at 20
00131  */
00132 #define XPAR_XUARTNS550_NUM_INSTANCES    1           /* Number of instances */

```

```

00133
00134 #define XPAR_UARTNS550_0_DEVICE_ID      20          /* Device ID for instance */
00135 #define XPAR_UARTNS550_0_BASEADDR        0xA0010000 /* IPIF base address */
00136 #define XPAR_UARTNS550_0_CLOCK_HZ       (66000000L) /* 66 MHz clock */
00137
00138 #define XPAR_UARTNS550_1_DEVICE_ID      21          /* Device ID for instance */
00139 #define XPAR_UARTNS550_1_BASEADDR        0xA0000000 /* IPIF base address */
00140 #define XPAR_UARTNS550_1_CLOCK_HZ       (66000000L) /* 66 MHz clock */
00141
00142 /*****
00143  *
00144  * UartLite defines.
00145  * DeviceID starts at 30
00146  */
00147 #define XPAR_XUARTLITE_NUM_INSTANCES 1          /* Number of instances */
00148
00149 #define XPAR_UARTLITE_0_DEVICE_ID      30          /* Device ID for instance */
00150 #define XPAR_UARTLITE_0_BASEADDR        0xA0020000 /* Device base address */
00151 #define XPAR_UARTLITE_0_BAUDRATE        19200      /* Baud rate */
00152 #define XPAR_UARTLITE_0_USE_PARITY      XFALSE     /* Parity generator enabled */
00153 #define XPAR_UARTLITE_0_ODD_PARITY      XFALSE     /* Type of parity generated */
00154 #define XPAR_UARTLITE_0_DATA_BITS       8          /* Data bits */
00155
00156 /*****
00157  *
00158  * ATM controller defines.
00159  * DeviceID starts at 40
00160  */
00161 #define XPAR_XATMC_NUM_INSTANCES        1          /* Number of instances */
00162
00163 #define XPAR_ATMC_0_DEVICE_ID           40          /* Device ID for instance */
00164 #define XPAR_ATMC_0_BASEADDR            0x70000000 /* Device base address */
00165 #define XPAR_ATMC_0_DMA_PRESENT         XFALSE     /* Does device have DMA? */
00166
00167 /*****
00168  *
00169  * Serial Peripheral Interface (SPI) defines.
00170  * DeviceID starts at 50
00171  */
00172 #define XPAR_XSPI_NUM_INSTANCES         2          /* Number of instances */
00173
00174 #define XPAR_SPI_0_DEVICE_ID            50          /* Device ID for instance */
00175 #define XPAR_SPI_0_BASEADDR             0x50000000 /* Device base address */
00176 #define XPAR_SPI_0_FIFO_EXIST          XTRUE      /* Does device have FIFOs? */
00177 #define XPAR_SPI_0_SLAVE_ONLY          XFALSE     /* Is the device slave only? */
00178 #define XPAR_SPI_0_NUM_SS_BITS          32         /* Number of slave select bits
*/
00179
00180 #define XPAR_SPI_1_DEVICE_ID            51          /* Device ID for instance */
00181 #define XPAR_SPI_1_BASEADDR             0x50000100 /* IPIF base address */
00182 #define XPAR_SPI_1_FIFO_EXIST          XTRUE      /* Does device have FIFOs? */
00183 #define XPAR_SPI_1_SLAVE_ONLY          XFALSE     /* Is the device slave only? */
00184 #define XPAR_SPI_1_NUM_SS_BITS          32         /* Number of slave select bits

```

```
*/
00185
00186 /*****
00187  *
00188  * OPB Arbiter defines.
00189  * DeviceID starts at 60
00190  */
00191 #define XPAR_XOPBARB_NUM_INSTANCES 1          /* Number of instances */
00192
00193 #define XPAR_OPBARB_0_DEVICE_ID      60        /* Device ID for instance */
00194 #define XPAR_OPBARB_0_BASEADDR      0x80000000 /* Register base address */
00195 #define XPAR_OPBARB_0_NUM_MASTERS    8         /* Number of masters on bus */
00196
00197 /*****
00198  *
00199  * Watchdog timer/timebase (WdtTb) defines.
00200  * DeviceID starts at 70
00201  */
00202 #define XPAR_XWDTTB_NUM_INSTANCES    1         /* Number of instances */
00203
00204 #define XPAR_WDTTB_0_DEVICE_ID       70        /* Device ID for instance */
00205 #define XPAR_WDTTB_0_BASEADDR       0x70800040 /* Register base address */
00206
00207 /*****
00208  *
00209  * Timer Counter (TmrCtr) defines.
00210  * DeviceID starts at 80
00211  */
00212 #define XPAR_XTMRCTR_NUM_INSTANCES   2         /* Number of instances */
00213
00214 #define XPAR_TMRCTR_0_DEVICE_ID      80        /* Device ID for instance */
00215 #define XPAR_TMRCTR_0_BASEADDR      0x70800100 /* Register base address */
00216
00217 /*****
00218  *
00219  * IIC defines.
00220  * DeviceID starts at 90
00221  */
00222 #define XPAR_XIIC_NUM_INSTANCES      2         /* Number of instances */
00223
00224 #define XPAR_IIC_0_DEVICE_ID         90        /* Device ID for instance */
00225 #define XPAR_IIC_0_BASEADDR         0xA8000000 /* Device base address */
00226 #define XPAR_IIC_0_TEN_BIT_ADR      XTRUE     /* Supports 10 bit addresses */
00227
00228 #define XPAR_IIC_1_DEVICE_ID         91        /* Device ID for instance */
00229 #define XPAR_IIC_1_BASEADDR         0xA8000000 /* Device base address */
00230 #define XPAR_IIC_1_TEN_BIT_ADR      XTRUE     /* Supports 10 bit addresses */
00231
00232 /*****
00233  *
00234  * Flash defines.
00235  * DeviceID starts at 100
```

```

00236 */
00237 #define XPAR_XFLASH_NUM_INSTANCES      1          /* Number of instances */
00238 #define XPAR_FLASH_INTEL_SUPPORT        /* Include intel flash support
*/
00239
00240 #define XPAR_FLASH_0_DEVICE_ID          100         /* Device ID for first instance
*/
00241 #define XPAR_FLASH_0_BASEADDR           0xFF000000 /* Base address of parts */
00242 #define XPAR_FLASH_0_NUM_PARTS          2          /* Number of parts in array */
00243 #define XPAR_FLASH_0_PART_WIDTH         2          /* Width of each part in bytes
*/
00244 #define XPAR_FLASH_0_PART_MODE           2          /* Mode of each part in bytes */
00245
00246 /*****
00247 *
00248 * GPIO defines.
00249 * DeviceID starts at 110
00250 */
00251 #define XPAR_XGPIO_NUM_INSTANCES        1
00252
00253 #define XPAR_GPIO_0_DEVICE_ID            110         /* Device ID for instance */
00254 #define XPAR_GPIO_0_BASEADDR            0x90000000 /* Register base address */
00255
00256
00257 /*****
00258 *
00259 * EMC defines.
00260 * DeviceID starts at 120
00261 */
00262 #define XPAR_XEMC_NUM_INSTANCES          1
00263
00264 #define XPAR_EMCC_0_DEVICE_ID            120         /* Device ID for instance */
00265 #define XPAR_EMCC_0_BASEADDR            0xE0000000 /* Register base address */
00266 #define XPAR_EMCC_0_NUM_BANKS_MEM        3          /* Number of banks */
00267
00268 /*****
00269 *
00270 * PLB Arbiter defines.
00271 * DeviceID starts at 130
00272 */
00273 #define XPAR_XPLBARB_NUM_INSTANCES        1
00274
00275 #define XPAR_PLBARB_0_DEVICE_ID          130         /* Device ID for instance */
00276 #define XPAR_PLBARB_0_BASEADDR          0x300        /* Register base address */
00277 #define XPAR_PLBARB_0_NUM_MASTERS        1          /* Number of masters on bus
*/
00278
00279 /*****
00280 *
00281 * PLB To OPB Bridge defines.
00282 * DeviceID starts at 140
00283 */
00284 #define XPAR_XPLB2OPB_NUM_INSTANCES      1

```

```
00285
00286 #define XPAR_PLB2OPB_0_DEVICE_ID      140          /* Device ID for instance */
00287 #define XPAR_PLB2OPB_0_DCR_BASEADDR    0x0          /* DCR Register base address
*/
00288 #define XPAR_PLB2OPB_0_NUM_MASTERS     1            /* Number of masters on bus
*/
00289
00290
00291 /*****
00292  *
00293  * OPB To PLB Bridge defines.
00294  * DeviceID starts at 150
00295  */
00296 #define XPAR_XOPB2PLB_NUM_INSTANCES     1
00297 #define XPAR_XOPB2PLB_ANY_OPB_REG_INTF   /* Accessible from OPB, not DCR */
00298
00299 #define XPAR_OPB2PLB_0_DEVICE_ID        150         /* Device ID for instance */
00300 #define XPAR_OPB2PLB_0_OPB_BASEADDR     0x0         /* Register base address */
00301 #define XPAR_OPB2PLB_0_DCR_BASEADDR     0x0         /* DCR Register base address */
00302
00303
00304 /*****
00305  *
00306  * System ACE defines.
00307  * DeviceID starts at 160
00308  */
00309 #define XPAR_XSYSACE_NUM_INSTANCES      1
00310
00311 #define XPAR_SYSACE_0_DEVICE_ID          160         /* Device ID for instance */
00312 #define XPAR_SYSACE_0_BASEADDR          0xCF000000   /* Register base address */
00313
00314
00315 /*****
00316  *
00317  * HDLC defines.
00318  * DeviceID starts at 170
00319  */
00320 #define XPAR_XHDLN_NUM_INSTANCES        1
00321
00322 #define XPAR_HDLN_0_DEVICE_ID            170         /* Device ID for instance
*/
00323 #define XPAR_HDLN_0_BASEADDR             0x60010000   /* Register base address */
00324 #define XPAR_HDLN_0_TX_MEM_DEPTH        2048         /* Tx FIFO depth (bytes) */
00325 #define XPAR_HDLN_0_RX_MEM_DEPTH        2048         /* Rx FIFO depth (bytes) */
00326 #define XPAR_HDLN_0_DMA_PRESENT         3            /* DMA SG in hardware */
00327
00328
00329 /*****
00330  *
00331  * PS2 Reference driver defines.
00332  * DeviceID starts at 180
00333  */
```

```
00334 #define XPAR_XPS2_NUM_INSTANCES      2
00335
00336 #define XPAR_PS2_0_DEVICE_ID           180          /* Device ID for instance */
00337 #define XPAR_PS2_0_BASEADDR            0x40010000    /* Register base address */
00338
00339 #define XPAR_PS2_1_DEVICE_ID           181          /* Device ID for instance */
00340 #define XPAR_PS2_1_BASEADDR            0x40020000    /* Register base address */
00341
00342 /*****
00343  *
00344  * Rapid IO defines.
00345  * DeviceID starts at 190
00346  */
00347 #define XPAR_XRAPIDIO_NUM_INSTANCES     1
00348
00349 #define XPAR_RAPIDIO_0_DEVICE_ID        190          /* Device ID for
instance */
00350 #define XPAR_RAPIDIO_0_BASEADDR        0x60000000    /* Register base address
*/
00351
00352
00353 /*****
00354  *
00355  * PCI defines.
00356  * DeviceID starts at 200
00357  */
00358 #define XPAR_XPCI_NUM_INSTANCES         1
00359 #define XPAR_OPB_PCI_1_DEVICE_ID        200
00360 #define XPAR_OPB_PCI_1_BASEADDR        0x86000000
00361 #define XPAR_OPB_PCI_1_HIGHADDR        0x860001FF
00362 #define XPAR_OPB_PCI_1_PCIBAR_0        0x10000000
00363 #define XPAR_OPB_PCI_1_PCIBAR_LEN_0    27
00364 #define XPAR_OPB_PCI_1_PCIBAR2IPIF_0   0xF0000000
00365 #define XPAR_OPB_PCI_1_PCIBAR_ENDIAN_TRANSLATE_EN_0 0
00366 #define XPAR_OPB_PCI_1_PCI_PREFETCH_0  1
00367 #define XPAR_OPB_PCI_1_PCI_SPACETYPE_0 1
00368 #define XPAR_OPB_PCI_1_PCIBAR_1        0x3F000000
00369 #define XPAR_OPB_PCI_1_PCIBAR_LEN_1    15
00370 #define XPAR_OPB_PCI_1_PCIBAR2IPIF_1   0xC0FF8000
00371 #define XPAR_OPB_PCI_1_PCIBAR_ENDIAN_TRANSLATE_EN_1 0
00372 #define XPAR_OPB_PCI_1_PCI_PREFETCH_1  1
00373 #define XPAR_OPB_PCI_1_PCI_SPACETYPE_1 1
00374 #define XPAR_OPB_PCI_1_PCIBAR_2        0x5F000000
00375 #define XPAR_OPB_PCI_1_PCIBAR_LEN_2    16
00376 #define XPAR_OPB_PCI_1_PCIBAR2IPIF_2   0x00000000
00377 #define XPAR_OPB_PCI_1_PCIBAR_ENDIAN_TRANSLATE_EN_2 0
00378 #define XPAR_OPB_PCI_1_PCI_PREFETCH_2  1
00379 #define XPAR_OPB_PCI_1_PCI_SPACETYPE_2 1
00380 #define XPAR_OPB_PCI_1_IPIFBAR_0        0x80000000
00381 #define XPAR_OPB_PCI_1_IPIF_HIGHADDR_0 0x81FFFFFF
00382 #define XPAR_OPB_PCI_1_IPIFBAR2PCI_0   0xF0000000
00383 #define XPAR_OPB_PCI_1_IPIFBAR_ENDIAN_TRANSLATE_EN_0 0
00384 #define XPAR_OPB_PCI_1_IPIF_PREFETCH_0 1
```

```
00385 #define XPAR_OPB_PCI_1_IPIF_SPACETYPE_0 1
00386 #define XPAR_OPB_PCI_1_IPIFBAR_1 0x82000000
00387 #define XPAR_OPB_PCI_1_IPIF_HIGHADDR_1 0x820007FF
00388 #define XPAR_OPB_PCI_1_IPIFBAR2PCI_1 0xCE000000
00389 #define XPAR_OPB_PCI_1_IPIFBAR_ENDIAN_TRANSLATE_EN_1 0
00390 #define XPAR_OPB_PCI_1_IPIF_PREFETCH_1 1
00391 #define XPAR_OPB_PCI_1_IPIF_SPACETYPE_1 1
00392 #define XPAR_OPB_PCI_1_IPIFBAR_2 0x82320000
00393 #define XPAR_OPB_PCI_1_IPIF_HIGHADDR_2 0x8232FFFF
00394 #define XPAR_OPB_PCI_1_IPIFBAR2PCI_2 0x00010000
00395 #define XPAR_OPB_PCI_1_IPIFBAR_ENDIAN_TRANSLATE_EN_2 0
00396 #define XPAR_OPB_PCI_1_IPIF_PREFETCH_2 1
00397 #define XPAR_OPB_PCI_1_IPIF_SPACETYPE_2 1
00398 #define XPAR_OPB_PCI_1_IPIFBAR_3 0x82330000
00399 #define XPAR_OPB_PCI_1_IPIF_HIGHADDR_3 0x8233FFFF
00400 #define XPAR_OPB_PCI_1_IPIFBAR2PCI_3 0x00010000
00401 #define XPAR_OPB_PCI_1_IPIFBAR_ENDIAN_TRANSLATE_EN_3 0
00402 #define XPAR_OPB_PCI_1_IPIF_PREFETCH_3 1
00403 #define XPAR_OPB_PCI_1_IPIF_SPACETYPE_3 0
00404 #define XPAR_OPB_PCI_1_IPIFBAR_4 0x82340000
00405 #define XPAR_OPB_PCI_1_IPIF_HIGHADDR_4 0x8234FFFF
00406 #define XPAR_OPB_PCI_1_IPIFBAR2PCI_4 0x00010000
00407 #define XPAR_OPB_PCI_1_IPIFBAR_ENDIAN_TRANSLATE_EN_4 0
00408 #define XPAR_OPB_PCI_1_IPIF_PREFETCH_4 0
00409 #define XPAR_OPB_PCI_1_IPIF_SPACETYPE_4 0
00410 #define XPAR_OPB_PCI_1_IPIFBAR_5 0x82350000
00411 #define XPAR_OPB_PCI_1_IPIF_HIGHADDR_5 0x8235FFFF
00412 #define XPAR_OPB_PCI_1_IPIFBAR2PCI_5 0x00010000
00413 #define XPAR_OPB_PCI_1_IPIFBAR_ENDIAN_TRANSLATE_EN_5 0
00414 #define XPAR_OPB_PCI_1_IPIF_PREFETCH_5 1
00415 #define XPAR_OPB_PCI_1_IPIF_SPACETYPE_5 1
00416 #define XPAR_OPB_PCI_1_DMA_BASEADDR 0x87000000
00417 #define XPAR_OPB_PCI_1_DMA_HIGHADDR 0x8700007F
00418 #define XPAR_OPB_PCI_1_DMA_CHAN_TYPE 0
00419 #define XPAR_OPB_PCI_1_DMA_LENGTH_WIDTH 11
00420
00421 /*****
00422  *
00423  * GEmac defines.
00424  * DeviceID starts at 210
00425  */
00426 #define XPAR_XGEMAC_NUM_INSTANCES 1
00427 #define XPAR_GEMAC_0_DEVICE_ID 210
00428 #define XPAR_GEMAC_0_BASEADDR 0x61000000
00429 #define XPAR_GEMAC_0_DMA_TYPE 9
00430 #define XPAR_GEMAC_0_MIIM_EXIST XFALSE
00431
00432
00433 /*****
00434  *
00435  * Touchscreen defines .
00436  * DeviceID starts at 220
```

```
00437 */
00438 #define XPAR_XTOUCHSCREEN_NUM_INSTANCES 1
00439 #define XPAR_TOUCHSCREEN_0_DEVICE_ID 220
00440 #define XPAR_TOUCHSCREEN_0_BASEADDR 0x70000000
00441
00442 /***** Type Definitions *****/
00443
00444
00445 /***** Macros (Inline Functions) Definitions *****/
00446
00447
00448 #endif /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

common/v1_00_a/src/xparameters.h File Reference

Detailed Description

This file contains system parameters for the Xilinx device driver environment. It is a representation of the system in that it contains the number of each device in the system as well as the parameters and memory map for each device. The user can view this file to obtain a summary of the devices in their system and the device parameters.

This file may be automatically generated by a design tool such as System Generator.

[Go to the source code of this file.](#)

XHdlc_Stats Struct Reference

```
#include <xhdlc.h>
```

Detailed Description

HDLC statistics (see [XHdlc_GetStats\(\)](#) and [XHdlc_ClearStats\(\)](#))

Data Fields

[Xuint16 XmitFrames](#)

[Xuint16 XmitBytes](#)

[Xuint16 RecvFrames](#)

[Xuint16 RecvBytes](#)

[Xuint16 RecvFcsErrors](#)

[Xuint16 RecvAlignmentErrors](#)

[Xuint16 RecvOverrunErrors](#)

[Xuint16 RecvAbortedFrames](#)

[Xuint16 FifoErrors](#)

[Xuint16 DmaErrors](#)

[Xuint16 RecvInterrupts](#)

[Xuint16 XmitInterrupts](#)

[Xuint16 HdlcInterrupts](#)

Field Documentation

Xuint16 XHdlc_Stats::DmaErrors

Number of DMA errors

Xuint16 XHdlc_Stats::FifoErrors

Number of FIFO errors since init

Xuint16 XHdlc_Stats::HdlcInterrupts

Number of HDLC (device) interrupts

Xuint16 XHdlc_Stats::RecvAbortedFrames

Number of transmit aborted frames

Xuint16 XHdlc_Stats::RecvAlignmentErrors

Number of frames received with alignment errors

Xuint16 XHdlc_Stats::RecvBytes

Number of bytes received

Xuint16 XHdlc_Stats::RecvFcsErrors

Number of frames discarded due to FCS errors

Xuint16 XHdlc_Stats::RecvFrames

Number of frames received

Xuint16 XHdlc_Stats::RecvInterrupts

Number of receive interrupts

Xuint16 XHdlc_Stats::RecvOverrunErrors

Number of frames discarded due to overrun errors

Xuint16 XHdlc_Stats::XmitBytes

Number of bytes transmitted

Xuint16 XHdlc_Stats::XmitFrames

Number of frames transmitted

Xuint16 XHdlc_Stats::XmitInterrupts

Number of transmit interrupts

The documentation for this struct was generated from the following file:

- [hdlc/v1_00_a/src/xhdlc.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

hdlc/v1_00_a/src/xhdlc.c File Reference

Detailed Description

Functions in this file are the minimum required functions for the HDLC driver. See [xhdlc.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	jhl	04/01/02	First release
1.00a	rpm	01/03/03	Changed location of IsStarted assignment in XHdlc_Start to be sure the flag is set before the device and interrupts are enabled.
1.00a	rmm	05/14/03	Fixed diab compiler warnings relating to asserts.

```
#include "xbasic_types.h"  
#include "xhdlc_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"  
#include "xdma_channel.h"  
#include "xhdlc.h"
```

Functions

XStatus XHdlc_Initialize (XHdlc *InstancePtr, **Xuint16** DeviceId)
XStatus XHdlc_Start (XHdlc *InstancePtr)
XStatus XHdlc_Stop (XHdlc *InstancePtr)
void **XHdlc_Reset** (XHdlc *InstancePtr)
XStatus XHdlc_Send (XHdlc *InstancePtr, **Xuint8** *FramePtr, unsigned ByteCount)

XStatus XHdlc_Recv (**XHdlc** *InstancePtr, **Xuint8** *FramePtr, unsigned *ByteCountPtr, **Xuint8** *FrameStatusPtr)

XHdlc_Config * **XHdlc_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
XStatus XHdlc_Initialize( XHdlc * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initialize a specific **XHdlc** instance/driver. The initialization entails:

- Initialize fields of the **XHdlc** structure
- Clear the HDLC statistics for this device
- Configure the FIFO components and DMA channels
- Reset the HDLC device

The driver defaults to polled mode operation. Interrupt mode can be selected using the SetOptions() function and turning off polled mode.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XHdlc** instance. Passing in a device id associates the generic **XHdlc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.
- XST_NO_FEATURE if the device configuration information indicates a feature that is not supported by this driver (no IPIF or simple DMA).

Note:

None.

```
XHdlc_Config* XHdlc_LookupConfig( Xuint16 DeviceId)
```

Lookup the device configuration based on the unique device ID. The table XHdlc_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

```
XStatus XHdlc_Recv( XHdlc * InstancePtr,  
                  Xuint8 * FramePtr,  
                  unsigned * ByteCountPtr,  
                  Xuint8 * FrameStatusPtr  
                  )
```

Receive an HDLC frame in polled mode. The driver receives the frame directly from the devices packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The buffer into which the frame will be received must be word-aligned.

The frames which are received by the device are stripped of the Opening Flag and Closing Flag fields such that buffers which receive data will not contain these fields. The frames do contain the FCS field. The Address field may or may not be contained in a receive buffer depending on the options set.

Parameters:

InstancePtr is a pointer to the XHdlc instance to be worked on.

FramePtr is a pointer to a 32 bit word aligned buffer into which the received HDLC frame will be copied.

ByteCountPtr is both an input and output parameter. It is a pointer to a 32-bit word that contains the number of bytes in the specified frame buffer on entry and the number of bytes in the received frame on return from the function.

FrameStatusPtr is an output which is changed by the driver to contain the status of the frame. It indicates any status that occurred for the received frame.

Returns:

- XST_SUCCESS if the frame was received successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the specified receive buffer is smaller than the the received frame. The received frame is not retrieved from the receive FIFO such that a reset of the device is necessary to resynchronize the internal length and data FIFOs.

- XST_FIFO_ERROR if a non-recoverable FIFO error has occurred. A reset of the device is necessary to clear this error.

Note:

Receive buffer must also be 32-bit aligned. The user must ensure that the size of the receive buffer is large enough to hold the frames received.

```
void XHdlc_Reset( XHdlc * InstancePtr)
```

Reset the HDLC instance. This is a graceful reset in that the device is stopped first then it resets the FIFOs and DMA channels if present. Reset must only be called after the driver has been initialized. The reset does not remove any of the buffer descriptors from the scatter-gather list for DMA.

The configuration after this reset is as follows:

- Disabled transmitter and receiver
- Device interrupts are disabled
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation

The upper layer software is responsible for re-configuring (if necessary) and restarting the HDLC device after the reset. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus XHdlc_Send( XHdlc * InstancePtr,  
                  Xuint8 * FramePtr,  
                  unsigned ByteCount  
                  )
```


Send an HDLC frame in polled mode. The driver writes the frame directly to the HDLC packet FIFO. Statistics are updated if an error previously occurred. The buffer to be sent must be word-aligned. This function is a non-blocking function in that it does not wait for the frame to be sent before returning.

The hardware device adds the Opening Flag, FCS, and Closing Flag fields to the frame such that these should not be put in the buffers which are to be sent.

The hardware device uses a length FIFO to keep track of each frame that has been put into the data FIFO. When sending a lot of short frames it is possible to fill this FIFO so that another frame cannot be sent until a frame has been sent by the device.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

FramePtr is a pointer to a 32 bit word aligned buffer containing the HDLC frame to be sent.

ByteCount is the size of the HDLC frame as an input. This size must be smaller than the size of the transmit FIFO of the device.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_FIFO_NO_ROOM if there is no room in the devices FIFOs for this frame.
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.

Note:

None.

XStatus XHdlc_Start(XHdlc * InstancePtr)

Start the HDLC device and driver by enabling the transmitter and receiver. This function must be called before other functions to send or receive data. It supports either polled or DMA scatter gather interrupt driven modes of operation. It does not yet support interrupt driven with FIFOs (non-DMA) or simple DMA without scatter gather.

If the driver is configured for interrupt driven operation, the interrupts of the device are enabled. The user should have connected the interrupt handler of the driver to an interrupt source such as an interrupt controller or the processor interrupt prior to this function being called.

Prior to calling this function, the functions **XHdlc_SetSgRecvSpace()** and **XHdlc_SetSgSendSpace()** should be called to setup the memory buffers and descriptor lists for the DMA scatter-gather.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_DEVICE_IS_STARTED if the device is already started
- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.
- XST_DMA_SG_LIST_EMPTY if configured for scatter-gather DMA and a descriptor list has been created for the receive channel but no buffers inserted into it.

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus XHdlc_Stop(XHdlc * InstancePtr)

Stop the HDLC device as follows:

- If the device is configured with DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode (the higher layer software is responsible for disabling interrupts at the interrupt controller)

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the HDLC frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

None.

hdlc/v1_00_a/src/xhdlc_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xhdlc_i.h,v 1.4 2002/06/28 18:18:15 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file hdlc/v1_00_a/src/xhdlc_i.h
00026 *
00027 * This header file contains internal identifiers, which are those shared
00028 * between XHdlc components.  The identifiers in this file are not intended for
00029 * use external to the driver.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver   Who   Date       Changes
00035 * ----  ---  -
00036 * 1.00a jhl   04/04/02 First release
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XHDLC_I_H /* prevent circular inclusions */
00042 #define XHDLC_I_H /* by using protection macros */
```



```

00095
00096 /* a default interrupt mask for scatter-gather DMA operation */
00097
00098 /* #define XHD_IIXR_DFT_SG_MASK (XHD_IIXR_DUALPORT_OUR_MASK | */
00099 #define XHD_IIXR_DFT_SG_MASK      (XHD_IIXR_ROLLOVER_MASK      | \
00100                                   XHD_IIXR_RLSFIFO_UNDER_MASK   | \
00101                                   XHD_IIXR_XLSFIFO_UNDER_MASK   | \
00102                                   XHD_IIXR_XLSFIFO_OVER_MASK    | \
00103                                   XHD_IIXR_RLSFIFO_OVER_MASK    | \
00104                                   XHD_IIXR_ALIGN_ERROR_MASK    | \
00105                                   XHD_IIXR_FCS_ERROR_MASK       | \
00106                                   XHD_IIXR_RDFIFO_OVER_MASK)
00107
00108 /*
00109  * Mask for the DMA interrupt enable and status registers.
00110  */
00111 #define XHD_DMA_SG_INTR_MASK      (XDC_IIXR_DMA_ERROR_MASK     | \
00112                                   XDC_IIXR_PKT_THRESHOLD_MASK  | \
00113                                   XDC_IIXR_PKT_WAIT_BOUND_MASK  | \
00114                                   XDC_IIXR_SG_END_MASK)
00115
00116 /***** Type Definitions *****/
00117
00118
00119 /***** Macros (Inline Functions) Definitions *****/
00120
00121
00122 #define XHD_CLEAR_STATS(InstancePtr)
00123 {
00124     Xuint8 *ClearPtr = (Xuint8 *)&InstancePtr->Stats;
00125     int Index;
00126
00127     /* Clear the stats in the instance */
00128
00129     for (Index = 0; Index < sizeof(XHdlc_Stats); Index++)
00130     {
00131         *ClearPtr++ = 0;
00132     }
00133     XIo_Out32(InstancePtr->BaseAddress + XHD_CRCE_OFFSET, 0);
00134     XIo_Out32(InstancePtr->BaseAddress + XHD_ABRT_OFFSET, 0);
00135 }
00136
00137 extern XHdlc_Config XHdlc_ConfigTable[];
00138
00139 #endif /* end of protection macro */

```

hdlc/v1_00_a/src/xhdlc_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between **XHdlc** components. The identifiers in this file are not intended for use external to the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	jhl	04/04/02	First release

```
#include "xhdlc.h"
```

[Go to the source code of this file.](#)

Variables

XHdlc_Config **XHdlc_ConfigTable** []

Variable Documentation

XHdlc_Config **XHdlc_ConfigTable**[()]

This table contains configuration information for each HDLC device in the system.

hdlc/v1_00_a/src/xhdlc_options.c File Reference

Detailed Description

Functions in this file allows options for the **XHdlc** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	jhl	04/02/02	First release

```
#include "xbasic_types.h"
#include "xhdlc_i.h"
#include "xipif_v1_23_b.h"
#include "xio.h"
```

Data Structures

```
struct OptionMap
```

Functions

XStatus **XHdlc_SetOptions** (**XHdlc** *InstancePtr, **Xuint8** Options)

Xuint8 **XHdlc_GetOptions** (**XHdlc** *InstancePtr)


```
void XHdlc_SetAddress (XHdlc *InstancePtr, Xuint16 Address)
Xuint16 XHdlc_GetAddress (XHdlc *InstancePtr)
```

Function Documentation

```
Xuint16 XHdlc_GetAddress( XHdlc * InstancePtr)
```

Get the receive address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

The receive address of the HDLC device.

Note:

None.

```
Xuint8 XHdlc_GetOptions( XHdlc * InstancePtr)
```

Get HDLC driver/device options. A value is returned which is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

The value of the HDLC options. The value is a bit-mask representing all options that are currently enabled. See [xhdlc.h](#) for a description of the available options.

Note:

None.

```
void XHdlc_SetAddress( XHdlc * InstancePtr,
                      Xuint16 Address
                      )
```

Set the receive address for this driver/device. The address is a 8 or 16 bit value within a HDLC frame.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Address is the address to be set.

Returns:

None.

Note:

None.

```
XStatus XHdlc_SetOptions( XHdlc * InstancePtr,  
                          Xuint8 Options  
                          )
```

Set HDLC driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Options is a bit-mask representing the HDLC options to turn on or off. See [xhdlc.h](#) for a description of the available options.

Returns:

- XST_SUCCESS if the options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped
- XST_NO_FEATURE if the polled option is being turned off and the device configuration information indicates DMA scatter gather is not supported. This driver does not support interrupt driven without DMA scatter-gather (FIFO only) yet.

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

hdlc/v1_00_a/src/xhdlc_stats.c File Reference

Detailed Description

Contains functions to get and clear the [XHdlc](#) driver statistics.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	jhl	04/02/02	First release

```
#include "xbasic_types.h"  
#include "xhdlc_i.h"
```

Functions

```
void XHdlc\_GetStats (XHdlc *InstancePtr, XHdlc\_Stats *StatsPtr)  
void XHdlc\_ClearStats (XHdlc *InstancePtr)
```

Function Documentation

```
void XHdlc_ClearStats(XHdlc * InstancePtr)
```

Clear the statistics for the specified HDLC driver instance.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

Returns:

None.

Note:

None.

```
void XHdlc_GetStats( XHdlc * InstancePtr,  
                    XHdlc_Stats * StatsPtr  
                    )
```

Get a copy of the statistics structure, which contains the current statistics for this driver. The statistics are only cleared at initialization or on demand using the **XHdlc_ClearStats()** function.

The FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the caller.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

hdlc/v1_00_a/src/xhdlc_dmasg.c File Reference

Detailed Description

This file contains the HDLC DMA scatter gather processing. This file contains send and receive functions as well as interrupt service routines.

Note:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	JHL	06/03/02	First release

```
#include "xhdlc.h"  
#include "xhdlc_i.h"  
#include "xpacket_fifo_v1_00_b.h"  
#include "xipif_v1_23_b.h"  
#include "xio.h"
```

Functions

[XStatus XHdlc_SgSend](#) ([XHdlc](#) *InstancePtr, [XBufDescriptor](#) *BdPtr)

[XStatus XHdlc_SgRecv](#) ([XHdlc](#) *InstancePtr, [XBufDescriptor](#) *BdPtr)

XStatus XHdlc_SgGetSendFrame (**XHdlc** *InstancePtr, XBufDescriptor **PtrToBdPtr, unsigned *BdCountPtr)
XStatus XHdlc_SgGetRecvFrame (**XHdlc** *InstancePtr, XBufDescriptor **PtrToBdPtr, unsigned *BdCountPtr)
void **XHdlc_InterruptHandler** (void *InstancePtr)
XStatus XHdlc_SetSgRecvSpace (**XHdlc** *InstancePtr, **Xuint32** *MemoryPtr, unsigned ByteCount)
XStatus XHdlc_SetSgSendSpace (**XHdlc** *InstancePtr, **Xuint32** *MemoryPtr, unsigned ByteCount)
void **XHdlc_SetSgRecvHandler** (**XHdlc** *InstancePtr, void *CallBackRef, **XHdlc_SgHandler** FuncPtr)
void **XHdlc_SetSgSendHandler** (**XHdlc** *InstancePtr, void *CallBackRef, **XHdlc_SgHandler** FuncPtr)
void **XHdlc_SetErrorHandler** (**XHdlc** *InstancePtr, void *CallBackRef, **XHdlc_ErrorHandler** FuncPtr)

Function Documentation

void XHdlc_InterruptHandler(void * *InstancePtr*)

Interrupt handler for the HDLC driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: the HDLC device, the send packet FIFO, the receive packet FIFO, the send DMA channel, or the receive DMA channel. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the HDLC device.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the HDLC device instance for the interrupt.

Returns:

None.

Note:

None.

```

void XHdlc_SetErrorHandler( XHdlc *           InstancePtr,
                           void *           CallbackRef,
                           XHdlc_ErrorHandler FuncPtr
                           )

```

Sets the callback function for handling errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback which should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable HDLC device error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_ERROR_COUNT_MAX indicates the counters of the HDLC device have reached the maximum value and that the statistics of the HDLC device should be cleared.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```

void XHdlc_SetSgRecvHandler( XHdlc *           InstancePtr,
                             void *           CallbackRef,
                             XHdlc_SgHandler FuncPtr
                             )

```


Sets the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of frames, determined by the DMA scatter-gather packet threshold, are received. The number of received frames is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each received frame from the list and should attach a new buffer to each descriptor. It is important that the specified number of frames passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XHdlc_SetSgRecvSpace( XHdlc * InstancePtr,  
                               Xuint32 * MemoryPtr,  
                               unsigned ByteCount  
                               )
```

Gives the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the HDLC driver. The memory space must be word-aligned.

This function must be called prior to calling [XHdlc_Start\(\)](#).

Parameters:

InstancePtr is a pointer to the [XHdlc](#) instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the HDLC device is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the [XHdlc_Initialize\(\)](#) function because the DMA channel components must be initialized before the memory space is set.

```
void XHdlc_SetSgSendHandler( XHdlc *           InstancePtr,  
                             void *           CallbackRef,  
                             XHdlc\_SgHandler FuncPtr  
                             )
```

Sets the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of frames, determined by the DMA scatter-gather packet threshold, are sent. The number of sent frames is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each sent frame from the list and should also free the buffers attached to the descriptors if necessary. It is important that the specified number of frames passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XHdlc_SetSgSendSpace( XHdlc * InstancePtr,  
                             Xuint32 * MemoryPtr,  
                             unsigned ByteCount  
                             )
```

Gives the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the HDLC driver. The memory space must be word-aligned.

This function must be called prior to calling **XHdlc_Start()**.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the HDLC device is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XHdlc_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
XStatus XHdlc_SgGetRecvFrame( XHdlc * InstancePtr,  
                               XBufDescriptor ** PtrToBdPtr,  
                               unsigned * BdCountPtr  
                               )
```

Gets the first buffer descriptor of the oldest frame which was received by the scatter-gather DMA channel of the HDLC device. This function is provided to be called from a callback function such that the buffer descriptors for received frames can be processed. The function should be called by the application repetitively for the number of frames indicated as an argument in the callback function.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the frame. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the frame. This input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the frame pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus XHdlc_SgGetSendFrame( XHdlc * InstancePtr,  
                               XBufDescriptor ** PtrToBdPtr,  
                               unsigned * BdCountPtr  
                               )
```

Gets the first buffer descriptor of the oldest frame which was sent by the scatter-gather DMA channel of the HDLC device. This function is provided to be called from a callback function such that the buffer descriptors for sent frames can be processed. The function should be called by the application repetitively for the number of frames indicated as an argument in the callback function.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the frame. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the frame. this input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the frame pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus XHdlc_SgRecv( XHdlc * InstancePtr,  
                    XBufDescriptor * BdPtr  
                    )
```

Adds this descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. The contents of the specified buffer descriptor are copied into the scatter-gather transmit list. This function can be called when the device is started or stopped.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

Note:

None.

```
XStatus XHdlc_SgSend( XHdlc * InstancePtr,
                    XBufDescriptor * BdPtr
                    )
```

Sends a HDLC frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire frame may or may not be contained within one descriptor. The contents of the buffer descriptor are copied into the scatter-gather transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the DMA control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted HDLC frame based upon the configuration of the HDLC device. The HDLC device must be started before calling this function.

Parameters:

InstancePtr is a pointer to the **XHdlc** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Returns:

- XST_SUCCESS if the buffer was successfully sent
- XST_DEVICE_IS_STOPPED if the HDLC device has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the

list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

Generated on 29 May 2003 for Xilinx Device Drivers

XAtmc Struct Reference

```
#include <xatmc.h>
```

Detailed Description

The XAtmc driver instance data. The user is required to allocate a variable of this type for every ATMC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `atmc/v1_00_c/src/xatmc.h`
-

atmc/v1_00_c/src/xatmc.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xatmc.h,v 1.1 2003/01/16 21:02:46 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *****/
00019 /*****
00020 /**
00021 *
00022 * @file atmc/v1_00_c/src/xatmc.h
00023 *
00024 * The implementation of the XAtmc component, which is the driver for the
00025 * Xilinx ATM controller.
00026 *
00027 * The Xilinx ATM controller supports the following features:
00028 * - Simple and scatter-gather DMA operations, as well as simple memory
00029 *   mapped direct I/O interface (FIFOs).
00030 * - Independent internal transmit and receive FIFOs
00031 * - Internal loopback
00032 * - Header error check (HEC) generation and checking
00033 * - Cell buffering with or without header/User Defined
00034 * - Parity generation and checking
00035 * - Header generation for transmit cell payloads
00036 * - Physical interface (PHY) data path of 16 bits
00037 * - Basic statistics gathering such as long cells, short cells, parity
errors,
00038 *   and HEC errors
00039 *
00040 * The driver does not support all of the features listed above. Features not
00041 * currently supported by the driver are:
```

```
00042 *   - Simple DMA (in polled or interrupt mode)
00043 *   - Direct I/O (FIFO) operations in interrupt mode (polled mode does use
00044 *     the FIFO directly)
00045 *
00046 * It is the responsibility of the application get the interrupt handler of
00047 * the ATM controller and connect it to the interrupt source.
00048 *
00049 * The driver services interrupts and passes ATM cells to the upper layer
00050 * software through callback functions. The upper layer software must register
00051 * its callback functions during initialization. The driver requires callback
00052 * functions for received cells, for confirmation of transmitted cells, and
00053 * for asynchronous errors. The frequency of interrupts can be controlled with
00054 * the packet threshold and packet wait bound features of the scatter-gather DMA
00055 * engine.
00056 *
00057 * The callback function which performs processing for scatter-gather DMA is
00058 * executed in an interrupt context and is designed to allow the processing of
00059 * the scatter-gather list to be passed to a thread context. The scatter-gather
00060 * processing can require more processing than desired in an interrupt context.
00061 * Functions are provided to be called from the callback function or thread
00062 * context to get cells from the send and receive scatter-gather list.
00063 *
00064 * Some errors that can occur in the device require a device reset. These
00065 * errors are listed in the SetErrorHandler function header. The upper layer's
00066 * error handler is responsible for resetting the device and re-configuring it
00067 * based on its needs (the driver does not save the current configuration).
00068 *
00069 * <b>DMA Support</b>
00070 *
00071 * The Xilinx ATMC device is available for both the IBM On-Chip Peripheral Bus
00072 * (OPB) and Processor Local Bus (PLB). This driver works for both. However, a
00073 * current limitation of the ATMC device on the PLB is that it does not support
00074 * DMA. For this reason, the DMA scatter-gather functions (e.g.,
XAtmc_SgSend())
00075 * of this driver will not function for the PLB version of the ATMC device.
00076 *
00077 * @note
00078 *
00079 * Xilinx drivers are typically composed of two components, one is the driver
00080 * and the other is the adapter. The driver is independent of OS and processor
00081 * and is intended to be highly portable. The adapter is OS-specific and
00082 * facilitates communication between the driver and the OS.
00083 * <br><br>
00084 * This driver is intended to be RTOS and processor independent. It works with
00085 * physical addresses only. Any needs for dynamic memory management, threads
00086 * or thread mutual exclusion, virtual memory, or cache control must be
00087 * satisfied by the layer above this driver.
00088 *
00089 * <pre>
00090 * MODIFICATION HISTORY:
00091 *
00092 * Ver   Who   Date       Changes
00093 * -----
```

```

00094 * 1.00a JHL 07/31/01 First release
00095 * 1.00c rpm 01/08/03 New release supports v2.00a of packet fifo driver
00096 *          an v1.23b of the IPIF driver
00097 * </pre>
00098 *
00099 *****/
00100
00101 #ifndef XATMC_H /* prevent circular inclusions */
00102 #define XATMC_H /* by using protection macros */
00103
00104 /***** Include Files *****/
00105
00106 #include "xstatus.h"
00107 #include "xparameters.h"
00108 #include "xdma_channel.h"
00109 #include "xbasic_types.h"
00110 #include "xpacket_fifo_v2_00_a.h"
00111
00112 /***** Constant Definitions *****/
00113
00114 /** @name Configuration options
00115  * These options are used in XAtmc_SetOptions() to configure the device.
00116  * @{
00117  */
00118 /**
00119  * <pre>
00120  *   XAT_LOOPBACK_OPTION           Enable sent data to be received
00121  *   XAT_POLLED_OPTION             Enables polled mode (no interrupts)
00122  *   XAT_DISCARD_SHORT_OPTION     Discard runt/short cells
00123  *   XAT_DISCARD_PARITY_OPTION    Discard cells with parity errors
00124  *   XAT_DISCARD_LONG_OPTION      Discard long cells
00125  *   XAT_DISCARD_HEC_OPTION        Discard cells with HEC errors
00126  *   XAT_DISCARD_VXI_OPTION       Discard cells which don't match in the
00127  *                                 VCI/VPI fields
00128  *   XAT_PAYLOAD_ONLY_OPTION       Buffer payload only
00129  *   XAT_NO_SEND_PARITY_OPTION     Disable parity for sent cells
00130  * </pre>
00131  */
00132 #define XAT_LOOPBACK_OPTION           0x1
00133 #define XAT_POLLED_OPTION             0x2
00134 #define XAT_DISCARD_SHORT_OPTION     0x4
00135 #define XAT_DISCARD_PARITY_OPTION    0x8
00136 #define XAT_DISCARD_LONG_OPTION      0x10
00137 #define XAT_DISCARD_HEC_OPTION        0x20
00138 #define XAT_DISCARD_VXI_OPTION       0x80
00139 #define XAT_PAYLOAD_ONLY_OPTION       0x200
00140 #define XAT_NO_SEND_PARITY_OPTION     0x800
00141 /* @} */
00142
00143 /** @name Cell status
00144  * These constants define the status values for a received cell.

```

```

00145 * The status is available when polling to receive a cell or in the buffer
00146 * descriptor after a cell is received using DMA scatter-gather.
00147 * @{
00148 */
00149 /**
00150 * <pre>
00151 * XAT_CELL_STATUS_LONG          Cell was too long
00152 * XAT_CELL_STATUS_SHORT         Cell was too short
00153 * XAT_CELL_STATUS_BAD_PARITY    Cell parity was not correct
00154 * XAT_CELL_STATUS_BAD_HEC       Cell HEC was not correct
00155 * XAT_CELL_STATUS_VXI_MISMATCH Cell VPI/VCI fields didn't match the expected
00156 *                               header values
00157 * XAT_CELL_STATUS_NO_ERROR      Cell received without errors
00158 * </pre>
00159 */
00160 #define XAT_CELL_STATUS_LONG          0x40UL
00161 #define XAT_CELL_STATUS_SHORT         0x20UL
00162 #define XAT_CELL_STATUS_BAD_PARITY    0x10UL
00163 #define XAT_CELL_STATUS_BAD_HEC       0x08UL
00164 #define XAT_CELL_STATUS_VXI_MISMATCH 0x04UL
00165 #define XAT_CELL_STATUS_NO_ERROR      0x02UL
00166 /*@}*/
00167
00168 /*
00169 * Constants to determine the configuration of the hardware device. They are
00170 * aused to allow the driver to verify it can operate with the hardware.
00171 */
00172 #define XAT_CFG_NO_IPIF              0          /* Not supported by the driver */
00173 #define XAT_CFG_NO_DMA                1          /* No DMA */
00174 #define XAT_CFG_SIMPLE_DMA            2          /* Not supported by the driver */
00175 #define XAT_CFG_DMA_SG                3          /* DMA scatter gather */
00176
00177 /*
00178 * Some default values for interrupt coalescing within the scatter-gather
00179 * DMA engine.
00180 */
00181 #define XAT_SGDMA_DFT_THRESHOLD       4          /* Default pkt threshold */
00182 #define XAT_SGDMA_MAX_THRESHOLD       255       /* Maximum pkt theshold */
00183 #define XAT_SGDMA_DFT_WAITBOUND       100       /* Default pkt wait bound (msec) */
00184 #define XAT_SGDMA_MAX_WAITBOUND       1023      /* Maximum pkt wait bound (msec) */
00185
00186 /*
00187 * Direction identifiers. These are used for setting values like packet
00188 * thresholds and wait bound for specific channels
00189 */
00190 #define XAT_SEND            1
00191 #define XAT_RECV            2
00192
00193 /*
00194 * The next few constants help upper layers determine the size of memory
00195 * pools used for ATM cells and descriptor lists.
00196 */

```

```

00197 #define XAT_PAYLOAD_SIZE      48
00198 #define XAT_MAX_HEADER_SIZE  6
00199 #define XAT_MAX_CELL_SIZE    (XAT_PAYLOAD_SIZE + XAT_MAX_HEADER_SIZE)
00200
00201 #define XAT_MIN_BUFFERS      16      /* minimum number of receive buffers */
00202 #define XAT_DFT_BUFFERS     64      /* default number of receive buffers */
00203
00204 #define XAT_MIN_RECV_DESC    8       /* minimum # of recv descriptors */
00205 #define XAT_DFT_RECV_DESC   32      /* default # of recv descriptors */
00206
00207 #define XAT_MIN_SEND_DESC    8       /* minimum # of send descriptors */
00208 #define XAT_DFT_SEND_DESC   32      /* default # of send descriptors */
00209
00210 /***** Type Definitions *****/
00211
00212 /**
00213  * ATM controller statistics
00214  */
00215 typedef struct
00216 {
00217     Xuint32 XmitCells;           /**< Number of cells transmitted */
00218     Xuint32 RecvCells;          /**< Number of cells received */
00219     Xuint32 RecvUnexpectedHeaders; /**< Number of cells with unexpected headers
*/
00220     Xuint32 RecvShortCells;     /**< Number of short cells */
00221     Xuint32 RecvLongCells;      /**< Number of long cells */
00222     Xuint32 RecvHecErrors;      /**< Number of HEC errors */
00223     Xuint32 RecvParityErrors;   /**< Number of parity errors */
00224     Xuint32 DmaErrors;          /**< Number of DMA errors since init */
00225     Xuint32 FifoErrors;         /**< Number of FIFO errors since init */
00226     Xuint32 RecvInterrupts;     /**< Number of receive interrupts */
00227     Xuint32 XmitInterrupts;     /**< Number of transmit interrupts */
00228     Xuint32 AtmcInterrupts;     /**< Number of ATMC interrupts */
00229 } XAtmc_Stats;
00230
00231
00232 /**
00233  * This typedef contains configuration information for the device.
00234  */
00235 typedef struct
00236 {
00237     Xuint16 DeviceId;           /**< Unique ID of device */
00238     Xuint32 BaseAddress;        /**< Base address of device */
00239     Xuint8  IpIfDmaConfig;      /**< IPIF/DMA hardware configuration */
00240 } XAtmc_Config;
00241
00242
00243 /** @name Typedefs for callbacks
00244  * Callback functions.
00245  * @{

```

```

00246 */
00247 /**
00248 * Callback when data is sent or received with scatter-gather DMA.
00249 * @param CallbackRef is a callback reference passed in by the upper layer
00250 *         when setting the callback functions, and passed back to the upper
00251 *         layer when the callback is invoked.
00252 * @param CellCount is the number of cells sent or received.
00253 */
00254 typedef void (*XAtmc_SgHandler)(void *CallbackRef, Xuint32 CellCount);
00255
00256 /**
00257 * Callback when data is sent or received with scatter-gather DMA.
00258 * @param CallbackRef is a callback reference passed in by the upper layer
00259 *         when setting the callback functions, and passed back to the upper
00260 *         layer when the callback is invoked.
00261 * @param ErrorCode indicates the error that occurred.
00262 */
00263 typedef void (*XAtmc_ErrorHandler)(void *CallbackRef, XStatus ErrorCode);
00264 /*@}*/
00265
00266 /**
00267 * The XAtmc driver instance data. The user is required to allocate a
00268 * variable of this type for every ATMC device in the system. A pointer
00269 * to a variable of this type is then passed to the driver API functions.
00270 */
00271 typedef struct
00272 {
00273     XAtmc_Stats Stats;
00274     Xuint32 BaseAddress;           /* Base address of ATM controller */
00275     Xuint32 IsStarted;
00276     Xuint32 IsReady;              /* Device is initialized and ready */
00277     Xboolean IsPolled;            /* Device is in polled mode */
00278     Xuint8 IpIfDmaConfig;         /* IPIF/DMA hardware configuration */
00279
00280     XPacketFifoV200a RecvFifo;    /* FIFO used by receive DMA channel */
00281     XPacketFifoV200a SendFifo;    /* FIFO used by send DMA channel */
00282     XDmaChannel RecvChannel;
00283     XDmaChannel SendChannel;
00284
00285     /*
00286      * Callbacks
00287      */
00288     XAtmc_SgHandler SgRecvHandler;
00289     void *SgRecvRef;
00290     XAtmc_SgHandler SgSendHandler;
00291     void *SgSendRef;
00292     XAtmc_ErrorHandler ErrorHandler;
00293     void *ErrorRef;
00294 } XAtmc;
00295
00296

```

```
00297
00298 /***** Macros (Inline Functions) Definitions *****/
00299
00300
00301 /***** Function Prototypes *****/
00302
00303 /*
00304  * Initialization functions
00305  */
00306 XStatus XAtmc_Initialize(XAtmc *InstancePtr, Xuint16 DeviceId);
00307 XStatus XAtmc_Start(XAtmc *InstancePtr);
00308 XStatus XAtmc_Stop(XAtmc *InstancePtr);
00309 void XAtmc_Reset(XAtmc *InstancePtr);
00310 XStatus XAtmc_SelfTest(XAtmc *InstancePtr);
00311 XAtmc_Config *XAtmc_LookupConfig(Xuint16 DeviceId);
00312
00313 /*
00314  * Send and receive functions
00315  */
00316 XStatus XAtmc_SgSend(XAtmc *InstancePtr, XBufDescriptor *BdPtr);
00317 XStatus XAtmc_SgRecv(XAtmc *InstancePtr, XBufDescriptor *BdPtr);
00318 XStatus XAtmc_SgGetSendCell(XAtmc* InstancePtr, XBufDescriptor **PtrToBdPtr,
00319                             int *BdCountPtr);
00320 XStatus XAtmc_SgGetRecvCell(XAtmc* InstancePtr, XBufDescriptor **PtrToBdPtr,
00321                             int *BdCountPtr);
00322 XStatus XAtmc_PollSend(XAtmc *InstancePtr, Xuint8 *BufPtr,
00323                       Xuint32 ByteCount);
00324 XStatus XAtmc_PollRecv(XAtmc *InstancePtr, Xuint8 *BufPtr,
00325                       Xuint32 *ByteCountPtr, Xuint32 *CellStatusPtr);
00326
00327 /*
00328  * ATMC configuration
00329  */
00330 XStatus XAtmc_SetOptions(XAtmc *InstancePtr, Xuint32 Options);
00331 Xuint32 XAtmc_GetOptions(XAtmc *InstancePtr);
00332
00333 XStatus XAtmc_SetPhyAddress(XAtmc *InstancePtr, Xuint8 Address);
00334 Xuint8 XAtmc_GetPhyAddress(XAtmc *InstancePtr);
00335
00336 XStatus XAtmc_SetHeader(XAtmc *InstancePtr, Xuint32 Direction,
00337                       Xuint32 Header);
00338 Xuint32 XAtmc_GetHeader(XAtmc *InstancePtr, Xuint32 Direction);
00339
00340 XStatus XAtmc_SetUserDefined(XAtmc *InstancePtr, Xuint8 UserDefined);
00341 Xuint8 XAtmc_GetUserDefined(XAtmc *InstancePtr);
00342
00343 XStatus XAtmc_SetPktThreshold(XAtmc *InstancePtr, Xuint32 Direction,
00344                             Xuint8 Threshold);
00345 XStatus XAtmc_GetPktThreshold(XAtmc *InstancePtr, Xuint32 Direction,
```

```

00346         Xuint8 *ThreshPtr);
00347 XStatus XAtmc_SetPktWaitBound(XAtmc *InstancePtr, Xuint32 Direction,
00348         Xuint32 TimerValue);
00349 XStatus XAtmc_GetPktWaitBound(XAtmc *InstancePtr, Xuint32 Direction,
00350         Xuint32 *WaitPtr);
00351
00352 /*
00353  * Statistics
00354  */
00355 void XAtmc_GetStats(XAtmc *InstancePtr, XAtmc_Stats *StatsPtr);
00356 void XAtmc_ClearStats(XAtmc *InstancePtr);
00357
00358 /*
00359  * Descriptor memory space
00360  */
00361 XStatus XAtmc_SetSgRecvSpace(XAtmc *InstancePtr, Xuint32 *MemoryPtr,
00362         Xuint32 ByteCount);
00363 XStatus XAtmc_SetSgSendSpace(XAtmc *InstancePtr, Xuint32 *MemoryPtr,
00364         Xuint32 ByteCount);
00365
00366 /*
00367  * Interrupt handler and Callbacks
00368  */
00369 void XAtmc_InterruptHandler(void *InstancePtr);
00370 void XAtmc_SetSgRecvHandler(XAtmc *InstancePtr, void *CallBackRef,
00371         XAtmc_SgHandler FuncPtr);
00372 void XAtmc_SetSgSendHandler(XAtmc *InstancePtr, void *CallBackRef,
00373         XAtmc_SgHandler FuncPtr);
00374 void XAtmc_SetErrorHandler(XAtmc *InstancePtr, void *CallBackRef,
00375         XAtmc_ErrorHandler FuncPtr);
00376
00377 #endif          /* end of protection macro */

```

Generated on 29 May 2003 for Xilinx Device Drivers

XAtmc_Stats Struct Reference

```
#include <xatmc.h>
```

Detailed Description

ATM controller statistics

Data Fields

[Xuint32 XmitCells](#)

[Xuint32 RecvCells](#)

[Xuint32 RecvUnexpectedHeaders](#)

[Xuint32 RecvShortCells](#)

[Xuint32 RecvLongCells](#)

[Xuint32 RecvHecErrors](#)

[Xuint32 RecvParityErrors](#)

[Xuint32 DmaErrors](#)

[Xuint32 FifoErrors](#)

[Xuint32 RecvInterrupts](#)

[Xuint32 XmitInterrupts](#)

[Xuint32 AtmcInterrupts](#)

Field Documentation

[Xuint32 XAtmc_Stats::AtmcInterrupts](#)

Number of ATMC interrupts

Xuint32 XAtmc_Stats::DmaErrors

Number of DMA errors since init

Xuint32 XAtmc_Stats::FifoErrors

Number of FIFO errors since init

Xuint32 XAtmc_Stats::RecvCells

Number of cells received

Xuint32 XAtmc_Stats::RecvHecErrors

Number of HEC errors

Xuint32 XAtmc_Stats::RecvInterrupts

Number of receive interrupts

Xuint32 XAtmc_Stats::RecvLongCells

Number of long cells

Xuint32 XAtmc_Stats::RecvParityErrors

Number of parity errors

Xuint32 XAtmc_Stats::RecvShortCells

Number of short cells

Xuint32 XAtmc_Stats::RecvUnexpectedHeaders

Number of cells with unexpected headers

Xuint32 XAtmc_Stats::XmitCells

Number of cells transmitted

Xuint32 XAtmc_Stats::XmitInterrupts

Number of transmit interrupts

The documentation for this struct was generated from the following file:

- [atmc/v1_00_c/src/xatmc.h](#)
-

Generated on 29 May 2003 for Xilinx Device Drivers

XAtmc_Config Struct Reference

```
#include <xatmc.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xuint8 IpIfDmaConfig

Field Documentation

Xuint32 XAtmc_Config::BaseAddress

Base address of device

Xuint16 XAtmc_Config::DeviceId

Unique ID of device

Xuint8 XAtmc_Config::IpIfDmaConfig

IPIF/DMA hardware configuration

The documentation for this struct was generated from the following file:

- [atmc/v1_00_c/src/xatmc.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

atmc/v1_00_c/src/xatmc_cfg.c File Reference

Detailed Description

Functions in this file handle configuration (including initialization, reset, and self-test) of the Xilinx ATM driver component.

Note:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	JHL	07/31/01	First release
1.00b	rpm	12/12/02	Changed location of IsStarted assignment in XAtmc_Start to be sure the flag is set before the device and interrupts are enabled.
1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver an v1.23b of the IPIF driver

```
#include "xatmc.h"  
#include "xatmc_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Data Structures

struct **Mapping**

Functions

XStatus XAtmc_Initialize (**XAtmc** *InstancePtr, **Xuint16** DeviceId)
XStatus XAtmc_Start (**XAtmc** *InstancePtr)
XStatus XAtmc_Stop (**XAtmc** *InstancePtr)
void **XAtmc_Reset** (**XAtmc** *InstancePtr)
XStatus XAtmc_SelfTest (**XAtmc** *InstancePtr)
XStatus XAtmc_SetOptions (**XAtmc** *InstancePtr, **Xuint32** OptionsFlag)
Xuint32 XAtmc_GetOptions (**XAtmc** *InstancePtr)
XStatus XAtmc_SetPhyAddress (**XAtmc** *InstancePtr, **Xuint8** Address)
Xuint8 XAtmc_GetPhyAddress (**XAtmc** *InstancePtr)
XStatus XAtmc_SetHeader (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** Header)
Xuint32 XAtmc_GetHeader (**XAtmc** *InstancePtr, **Xuint32** Direction)
XStatus XAtmc_SetUserDefined (**XAtmc** *InstancePtr, **Xuint8** UserDefined)
Xuint8 XAtmc_GetUserDefined (**XAtmc** *InstancePtr)
XStatus XAtmc_SetPktThreshold (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint8** Threshold)
XStatus XAtmc_GetPktThreshold (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)
XStatus XAtmc_SetPktWaitBound (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)
XStatus XAtmc_GetPktWaitBound (**XAtmc** *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)
void **XAtmc_GetStats** (**XAtmc** *InstancePtr, **XAtmc_Stats** *StatsPtr)
void **XAtmc_ClearStats** (**XAtmc** *InstancePtr)
XStatus XAtmc_SetSgRecvSpace (**XAtmc** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)
XStatus XAtmc_SetSgSendSpace (**XAtmc** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)
void **XAtmc_SetSgRecvHandler** (**XAtmc** *InstancePtr, void *CallbackRef, **XAtmc_SgHandler** FuncPtr)
void **XAtmc_SetSgSendHandler** (**XAtmc** *InstancePtr, void *CallbackRef, **XAtmc_SgHandler** FuncPtr)
void **XAtmc_SetErrorHandler** (**XAtmc** *InstancePtr, void *CallbackRef, **XAtmc_ErrorHandler** FuncPtr)
XAtmc_Config * **XAtmc_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void XAtmc_ClearStats( XAtmc * InstancePtr)
```

Clears the [XAtmc_Stats](#) structure for this driver.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 XAtmc_GetHeader( XAtmc * InstancePtr,  
                        Xuint32 Direction  
                        )
```

Gets the send or receive ATM header in the ATM controller. The ATM controller attaches the send header to cells which are to be sent but contain only the payload.

If the ATM controller is configured appropriately, it will compare the header of received cells against the receive header and discard cells which don't match in the VCI and VPI fields of the header.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Direction indicates whether we're retrieving the send header or the receive header.

Returns:

The ATM header currently being used by the ATM controller for attachment to transmitted cells or the header which is being compared against received cells. An invalid specified direction will cause this function to return a value of 0.

Note:

None.

```
Xuint32 XAtmc_GetOptions( XAtmc * InstancePtr)
```


Gets Atmc driver/device options. The value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The 32-bit value of the Atmc options. The value is a bit-mask representing all options that are currently enabled. See **xatmc.h** for a detailed description of the options.

Note:

None.

Xuint8 XAtmc_GetPhyAddress(**XAtmc** * *InstancePtr*)

Gets the PHY address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The 5-bit PHY address (0 - 31) currently being used by the ATM controller.

Note:

None.

XStatus XAtmc_GetPktThreshold(**XAtmc** * *InstancePtr*,
 Xuint32 *Direction*,
 Xuint8 * *ThreshPtr*
)

Gets the value of the packet threshold register for this driver/device. The packet threshold is used for interrupt coalescing when the ATM controller is configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus XAtmc_GetPktWaitBound( XAtmc * InstancePtr,  
                               Xuint32 Direction,  
                               Xuint32 * WaitPtr  
                               )
```

Gets the packet wait bound register for this driver/device. The packet wait bound is used for interrupt coalescing when the ATM controller is configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
void XAtmc_GetStats( XAtmc * InstancePtr,  
                    XAtmc_Stats * StatsPtr  
                    )
```

Gets a copy of the **XAtmc_Stats** structure, which contains the current statistics for this driver.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None. Although the output parameter will contain a copy of the statistics upon return from this function.

Note:

None.

```
Xuint8 XAtmc_GetUserDefined( XAtmc * InstancePtr)
```

Gets the 2nd byte of the User Defined data in the ATM controller for the channel which is sending data. The ATM controller will attach the header to all cells which are being sent and do not have a header. The header of a 16 bit Utopia interface contains the User Defined data which is two bytes. The first byte contains the HEC field and the second byte is available for user data. This function only allows the second byte to be retrieved.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

The second byte of the User Defined data.

Note:

None.

```
XStatus XAtmc_Initialize( XAtmc * InstancePtr,  
                          Xuint16 DeviceId  
                          )
```

Initializes a specific ATM controller instance/driver. The initialization entails:

- Initialize fields of the **XAtmc** structure
- Clear the ATM statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- Configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists will be passed to the driver.
- Reset the ATM controller

The only driver function that should be called before this Initialize function is called is GetInstance.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XAtmc** instance. Passing in a device id associates the generic **XAtmc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started

Note:

None.

XAtmc_Config* XAtmc_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table AtmcConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId contains the unique device ID that for the device. This ID is used to lookup the configuration.

Returns:

A pointer to the configuration for the specified device, or XNULL if the device could not be found.

Note:

None.

void XAtmc_Reset(XAtmc * InstancePtr)

Resets the ATM controller. It resets the the DMA channels, the FIFOs, and the ATM controller. The reset does not remove any of the buffer descriptors from the scatter-gather list for DMA. Reset must only be called after the driver has been initialized.

The configuration after this reset is as follows:

- Disabled transmitter and receiver
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- PHY address of 0

The upper layer software is responsible for re-configuring (if necessary) and restarting the ATM controller after the reset.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

None.

Note:

The reset is accomplished by setting the IPIF reset register. This takes care of resetting all hardware blocks, including the ATM controller.

XStatus XAtmc_SelfTest(XAtmc * InstancePtr)

Performs a self-test on the ATM controller device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the ATM controller device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if self-test was successful
- XST_PFIFO_BAD_REG_VALUE if the FIFO failed register self-test
- XST_DMA_TRANSFER_ERROR if DMA failed data transfer self-test
- XST_DMA_RESET_REGISTER_ERROR if DMA control register value was incorrect after a reset
- XST_REGISTER_ERROR if the ATM controller failed register reset test
- XST_LOOPBACK_ERROR if the ATM controller internal loopback failed
- XST_IPIF_REG_WIDTH_ERROR if an invalid register width was passed into the function
- XST_IPIF_RESET_REGISTER_ERROR if the value of a register at reset was invalid
- XST_IPIF_DEVICE_STATUS_ERROR if a write to the device status register did not read back correctly
- XST_IPIF_DEVICE_ACK_ERROR if a bit in the device status register did not reset when acked
- XST_IPIF_DEVICE_ENABLE_ERROR if the device interrupt enable register was not updated correctly by the hardware when other registers were written to
- XST_IPIF_IP_STATUS_ERROR if a write to the IP interrupt status register did not read back correctly
- XST_IPIF_IP_ACK_ERROR if one or more bits in the IP status register did not reset when acked
- XST_IPIF_IP_ENABLE_ERROR if the IP interrupt enable register was not updated correctly when other registers were written to

Note:

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the self-test thread.

```
void XAtmc_SetErrorHandler( XAtmc *           InstancePtr,  
                           void *           CallbackRef,  
                           XAtmc_ErrorHandler FuncPtr  
                           )
```

Sets the callback function for handling errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback which should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- `XST_DMA_ERROR` indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- `XST_FIFO_ERROR` indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- `XST_RESET_ERROR` indicates an unrecoverable ATM controller error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- `XST_ATMC_ERROR_COUNT_MAX` indicates the counters of the ATM controller have reached the maximum value and that the statistics of the ATM controller should be cleared.

Parameters:

InstancePtr is a pointer to the `XAtmc` instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetHeader( XAtmc * InstancePtr,  
                        Xuint32 Direction,  
                        Xuint32 Header  
                        )
```

Sets the send or receive ATM header in the ATM controller. If cells with only payloads are given to the controller to be sent, it will attach the header to the cells. If the ATM controller is configured appropriately, it will compare the header of received cells against the receive header and discard cells which don't match in the VCI and VPI fields of the header.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the direction, send(transmit) or receive, for the header to set.

Header contains the ATM header to be attached to each transmitted cell for cells with only payloads or the expected header for cells which are received.

Returns:

- XST_SUCCESS if the PHY address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus XAtmc_SetOptions( XAtmc * InstancePtr,  
                          Xuint32 OptionsFlag  
                          )
```

Set Atmc driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option. A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off. See [xatmc.h](#) for a detailed description of the available options.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

OptionsFlag is a bit-mask representing the Atmc options to turn on or off

Returns:

- XST_SUCCESS if options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.


```
XStatus XAtmc_SetPhyAddress( XAtmc * InstancePtr,  
                             Xuint8   Address  
                             )
```

Sets the PHY address for this driver/device. The address is a 5-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Address contains the 5-bit PHY address (0 - 31).

Returns:

- XST_SUCCESS if the PHY address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus XAtmc_SetPktThreshold( XAtmc * InstancePtr,  
                               Xuint32 Direction,  
                               Xuint8   Threshold  
                               )
```

Sets the packet count threshold register for this driver/device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_DMA_SG_COUNT_EXCEEDED if the threshold must be equal to or less than the number of descriptors in the list
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
XStatus XAtmc_SetPktWaitBound( XAtmc * InstancePtr,  
                               Xuint32 Direction,  
                               Xuint32 TimerValue  
                               )
```

Sets the packet wait bound register for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold.

The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if an invalid direction was specified

Note:

None.

```
void XAtmc_SetSgRecvHandler( XAtmc * InstancePtr,  
                             void * CallbackRef,  
                             XAtmc_SgHandler FuncPtr  
                             )
```

Sets the callback function for handling received cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are received. The number of received cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each received cell from the list and should attach a new buffer to each descriptor. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are other potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

CallBackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetSgRecvSpace( XAtmc * InstancePtr,  
                             Xuint32 * MemoryPtr,  
                             Xuint32 ByteCount  
                             )
```

Gives the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Atmc driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XAtmc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
void XAtmc_SetSgSendHandler( XAtmc *           InstancePtr,  
                             void *           CallbackRef,  
                             XAtmc_SgHandler FuncPtr  
                             )
```

Sets the callback function for handling confirmation of transmitted cells in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called when a number of cells, determined by the DMA scatter-gather packet threshold, are sent. The number of sent cells is passed to the callback function. The callback function should communicate the data to a thread such that the scatter-gather list processing is not performed in an interrupt context.

The scatter-gather list processing of the thread context should call the function to get the buffer descriptors for each sent cell from the list and should also free the buffers attached to the descriptors if necessary. It is important that the specified number of cells passed to the callback function are handled by the scatter-gather list processing.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the XAtmc instance to be worked on.

CallbackRef is a reference pointer to be passed back to the application in the callback. This helps the application correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XAtmc_SetSgSendSpace( XAtmc *           InstancePtr,  
                              Xuint32 *        MemoryPtr,  
                              Xuint32         ByteCount  
                              )
```

Gives the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Atmc driver. The memory space must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the ATM controller is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if the list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the XAtmc_Initialize function because the DMA channel components must be initialized before the memory space is set.

```
XStatus XAtmc_SetUserDefined( XAtmc * InstancePtr,  
                             Xuint8 UserDefined  
                             )
```

Sets the 2nd byte of the User Defined data in the ATM controller for the channel which is sending data. The ATM controller will attach the header to all cells which are being sent and do not have a header. The header of a 16 bit Utopia interface contains the User Defined data which is two bytes. The first byte contains the HEC field and the second byte is available for user data. This function only allows the second byte to be set.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

UserDefined contains the second byte of the User Defined data.

Returns:

- XST_SUCCESS if the user-defined data was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus XAtmc_Start( XAtmc * InstancePtr)
```

Starts the ATM controller as follows:

- If not in polled mode enable interrupts
- Enable the transmitter
- Enable the receiver
- Start the DMA channels if the descriptor lists are not empty

It is necessary for the caller to connect the interrupt service routine of the ATM controller to the interrupt source, typically an interrupt controller, and enable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.
- XST_DMA_SG_LIST_EMPTY iff configured for scatter-gather DMA and no buffer descriptors have been put into the list for the receive channel.

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus XAtmc_Stop(XAtmc * InstancePtr)

Stops the ATM controller as follows:

- Stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode

It is the callers responsibility to disconnect the interrupt handler of the ATM controller from the interrupt source, typically an interrupt controller, and disable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

Generated on 29 May 2003 for Xilinx Device Drivers

atmc/v1_00_c/src/xatmc_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *****/
00019 /*****
00020 /**
00021 *
00022 * @file atmc/v1_00_c/src/xatmc_i.h
00023 *
00024 * This file contains data which is shared between files internal to the
00025 * XAtmc component. It is intended for internal use only.
00026 *
00027 * <pre>
00028 * MODIFICATION HISTORY:
00029 *
00030 * Ver    Who    Date      Changes
00031 * -----
00032 * 1.00a  JHL    07/31/01  First release
00033 * 1.00c  rpm    01/08/03  New release supports v2.00a of packet fifo driver
00034 *                               an v1.23b of the IPIF driver
00035 * </pre>
00036 *
00037 *****/
00038
00039 #ifndef XATMC_I_H /* prevent circular inclusions */
00040 #define XATMC_I_H /* by using protection macros */
00041
00042 /***** Include Files *****/
```



```

00095 XAT_IIX_XLFIFO_OVER_MASK | \
00096 XAT_IIX_XLFIFO_UNDER_MASK | \
00097 XAT_IIX_XSFIFO_OVER_MASK | \
00098 XAT_IIX_XSFIFO_UNDER_MASK | \
00099 XAT_IIX_COUNT_MAX_MASK)
00100
00101 /*
00102  * the following constant defines a mask for the DMA interrupt enable and
00103  * status registers which includes interrupts for DMA errors, packet threshold,
00104  * packet wait bound, and the end of the scatter gather list
00105  */
00106 #define XAT_DMA_SG_INTR_MASK (XDC_IXR_DMA_ERROR_MASK | \
00107 XDC_IXR_PKT_THRESHOLD_MASK | \
00108 XDC_IXR_PKT_WAIT_BOUND_MASK | \
00109 XDC_IXR_SG_END_MASK)
00110
00111 /***** Type Definitions *****/
00112
00113
00114 /***** Macros (Inline Functions) Definitions *****/
00115
00116 /*****
00117 **
00118 *
00119 * This macro determines if the device is currently configured for
00120 * scatter-gather DMA.
00121 *
00122 * @param InstancePtr is a pointer to the XAtmc instance to be worked on.
00123 *
00124 * @return
00125 *
00126 * Boolean XTRUE if the device is configured for scatter-gather DMA, or XFALSE
00127 * if it is not.
00128 *
00129 * @note
00130 *
00131 * Signature: Xboolean XAtmc_mIsSgDma(XAtmc *InstancePtr)
00132 *
00133 *****/
00134 #define XAtmc_mIsSgDma(InstancePtr) \
00135 ((InstancePtr)->IpIfDmaConfig == XAT_CFG_DMA_SG)
00136
00137 /***** Variable Definitions *****/
00138
00139 extern XAtmc_Config XAtmc_ConfigTable[];
00140
00141 /***** Function Prototypes *****/
00142
00143
00144 #endif /* end of protection macro */

```


atmc/v1_00_c/src/xatmc_i.h File Reference

Detailed Description

This file contains data which is shared between files internal to the **XAtmc** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	JHL	07/31/01	First release
1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver an v1.23b of the IPIF driver

```
#include "xatmc.h"  
#include "xatmc_1.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XAtmc_mIsSgDma(InstancePtr)
```

Variables

[XAtmc_Config](#) [XAtmc_ConfigTable](#) []

Define Documentation

```
#define XAtmc_mIsSgDma( InstancePtr )
```

This macro determines if the device is currently configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the [XAtmc](#) instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for scatter-gather DMA, or XFALSE if it is not.

Note:

Signature: Xboolean [XAtmc_mIsSgDma](#)(XAtmc *InstancePtr)

Variable Documentation

```
XAtmc\_Config XAtmc\_ConfigTable[]( )
```

This table contains configuration information for each ATMC device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

atmc/v1_00_c/src/xatmc_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xatmc_l.h,v 1.1 2003/01/16 21:02:46 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file atmc/v1_00_c/src/xatmc_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  High-level driver functions
00029 * are defined in xatmc.h.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver   Who   Date       Changes
00035 * ----  ---  -
00036 * 1.00b rpm  05/01/02 First release
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XATMC_L_H /* prevent circular inclusions */
00042 #define XATMC_L_H /* by using protection macros */
```

```

00043
00044 /***** Include Files *****/
00045
00046 #include "xbasic_types.h"
00047 #include "xio.h"
00048
00049 /***** Constant Definitions *****/
00050
00051 /* Offset of the ATMC registers from the IPIF base address */
00052 #define XAT_REG_OFFSET      0x1100UL
00053
00054 /*
00055  * Register offsets for the ATM controller. Each register is 32 bits and is
00056  * read/write unless specified otherwise.
00057  */
00058 #define XAT_AMIR_OFFSET     (XAT_REG_OFFSET + 0)    /* Module ID (read only) */
00059 #define XAT_ASRR_OFFSET     (XAT_REG_OFFSET + 0)    /* Soft reset (write only) */
00060 #define XAT_ATCR_OFFSET     (XAT_REG_OFFSET + 4)    /* Control */
00061 #define XAT_ATXHR_OFFSET    (XAT_REG_OFFSET + 8)    /* Tx header */
00062 #define XAT_ATUDR_OFFSET    (XAT_REG_OFFSET + 12)   /* Tx user defined */
00063 #define XAT_ATPLR_OFFSET    (XAT_REG_OFFSET + 16)   /* Tx packet length */
00064 #define XAT_ATSR_OFFSET     (XAT_REG_OFFSET + 20)   /* Tx status (read only) */
00065 #define XAT_ARXHR_OFFSET    (XAT_REG_OFFSET + 24)   /* Rx expected header */
00066 #define XAT_ARPLR_OFFSET    (XAT_REG_OFFSET + 28)   /* Rx pkt length (read only)
00067 */
00067 #define XAT_ARSR_OFFSET     (XAT_REG_OFFSET + 32)   /* Rx status (read only) */
00068 #define XAT_AUHC_OFFSET     (XAT_REG_OFFSET + 36)   /* Unexpected header count */
00069 #define XAT_ARCC_OFFSET     (XAT_REG_OFFSET + 40)   /* Runt (short) cell count */
00070 #define XAT_ARLC_OFFSET     (XAT_REG_OFFSET + 44)   /* Long cell count */
00071 #define XAT_AHECC_OFFSET    (XAT_REG_OFFSET + 48)   /* HEC error cell count */
00072 #define XAT_APECC_OFFSET    (XAT_REG_OFFSET + 52)   /* Parity error cell count */
00073
00074 /*
00075  * Register offsets for the IPIF components
00076  */
00077 #define XAT_ISR_OFFSET      0x20UL                /* Interrupt status */
00078
00079 #define XAT_DMA_OFFSET      0x2300UL
00080 #define XAT_DMA_SEND_OFFSET (XAT_DMA_OFFSET + 0x0) /* DMA send channel */
00081 #define XAT_DMA_RECV_OFFSET (XAT_DMA_OFFSET + 0x40) /* DMA recv channel */
00082
00083 #define XAT_PFIFO_OFFSET    0x2000UL
00084 #define XAT_PFIFO_TXREG_OFFSET (XAT_PFIFO_OFFSET + 0x0) /* Tx registers */
00085 #define XAT_PFIFO_RXREG_OFFSET (XAT_PFIFO_OFFSET + 0x10) /* Rx registers */
00086 #define XAT_PFIFO_TXDATA_OFFSET (XAT_PFIFO_OFFSET + 0x100) /* Tx keyhole */
00087 #define XAT_PFIFO_RXDATA_OFFSET (XAT_PFIFO_OFFSET + 0x200) /* Rx keyhole */
00088
00089
00090 /*
00091  * ATMC Module Identification Register (AMIR)
00092  */
00093 #define XAT_AMIR_VERSION_MASK 0xFFFF0000UL        /* Device version */

```

```

00094 #define XAT_AMIR_TYPE_MASK          0x0000FF00UL          /* Device type */
00095
00096 /*
00097  * ATMC Reset Register (ASRR)
00098  */
00099 #define XAT_ASRR_RCV_RESET_MASK      0x0A000000UL          /* receiver reset */
00100 #define XAT_ASRR_XMT_RESET_MASK      0xA0000000UL          /* transmitter reset */
00101
00102 /*
00103  * ATMC Control Register (ATCR)
00104  */
00105 #define XAT_ATCR_NO_LONG_MASK        0x80000000UL /* Discard long */
00106 #define XAT_ATCR_NO_RUNTS_MASK       0x40000000UL /* Discard runt/short */
00107 #define XAT_ATCR_NO_PARITY_MASK      0x20000000UL /* Discard bad parity */
00108 #define XAT_ATCR_NO_HEC_MASK         0x08000000UL /* Discard HEC errors */
00109 #define XAT_ATCR_NO_VXI_MASK         0x02000000UL /* Discard VPI/VCI errors */
00110 #define XAT_ATCR_PAYLOAD_ONLY_MASK   0x00800000UL /* Receive only payloads */
00111 #define XAT_ATCR_RECV_ENABLE_MASK    0x00400000UL /* Receiver enable */
00112 #define XAT_ATCR_LOOPBACK_MASK       0x00020000UL /* Loopback test mode */
00113 #define XAT_ATCR_NNI_MASK            0x00010000UL /* 1=network node interface
00114                                     0=user network interface */
00115 #define XAT_ATCR_MULTI_PHY_MASK      0x00008000UL /* Multiple PHYs */
00116 #define XAT_ATCR_PHY_ADDRESS_MASK    0x00007C00UL /* PHY address */
00117 #define XAT_ATCR_AAL_PRESENT_MASK    0x00000200UL /* An AAL is present */
00118 #define XAT_ATCR_PAYLOAD_IF_MASK     0x00000100UL /* Send/receive payloads */
00119 #define XAT_ATCR_UTOPIA_MASTER_MASK  0x00000080UL /* 1 = master, 0 = slave */
00120 #define XAT_ATCR_UTOPIA_8_BIT_MASK   0x00000040UL /* 1 = 8 bit, 0 = 16 bit */
00121 #define XAT_ATCR_NO_PARITY_GEN_MASK   0x00000020UL /* Don't generate parity */
00122 #define XAT_ATCR_XMIT_ENABLE_MASK     0x00000010UL /* Transmitter enable */
00123
00124 #define XAT_ATCR_PHY_ADDR_SHIFT_VALUE 10          /* location of PHY address in
00125                                     * in the control register */
00126
00127 /*
00128  * Transmit Status Register (TSR)
00129  */
00130 #define XAT_TSR_PACKET_DONE_MASK     0x00000001UL /* transmit packet complete */
00131
00132 /*
00133  * ATMC Receive Status Register (ARSR)
00134  */
00135 #define XAT_ARSR_RUNT_MASK            0x00000020UL /* Recv runt/short cell */
00136 #define XAT_ARSR_PARITY_MASK          0x00000010UL /* Recv parity error cell */
00137 #define XAT_ARSR_HEC_MASK             0x00000008UL /* Recv HEC error cell */
00138 #define XAT_ARSR_VXI_MASK             0x00000004UL /* Recv VPI/VCI error cell */
00139 #define XAT_ARSR_NO_ERROR_MASK        0x00000002UL /* Recv cell without error */
00140 #define XAT_ARSR_COMPLETE_MASK        0x00000001UL /* Recv cell complete */
00141
00142
00143 /*
00144  * IPIF Interrupt Registers (Status and Enable) masks. These registers are
00145  * part of the IPIF device interrupt registers
00146  */

```



```

00147 #define XAT_IIX_XMIT_DONE_MASK      0x08000000UL /* Xmit complete */
00148 #define XAT_IIX_RECV_DONE_MASK      0x04000000UL /* Recv complete */
00149 #define XAT_IIX_RECV_RUNT_MASK      0x02000000UL /* Recv runt/short cell */
00150 #define XAT_IIX_RECV_LONG_MASK      0x01000000UL /* Recv long cell */
00151 #define XAT_IIX_RECV_HEC_MASK       0x00800000UL /* Recv HEC error cell */
00152 #define XAT_IIX_RECV_VXI_MASK       0x00400000UL /* Recv VPI/VCI error cell */
00153 #define XAT_IIX_RECV_PARITY_MASK     0x00200000UL /* Recv parity error cell */
00154 #define XAT_IIX_XSFIFO_EMPTY_MASK    0x00100000UL /* Xmit status fifo empty */
00155 #define XAT_IIX_RSFFIFO_EMPTY_MASK   0x00080000UL /* Recv status fifo empty */
00156 #define XAT_IIX_XLFIFO_EMPTY_MASK    0x00040000UL /* Xmit length fifo empty */
00157 #define XAT_IIX_RLFIFO_EMPTY_MASK    0x00020000UL /* Recv length fifo empty */
00158 #define XAT_IIX_XSFIFO_FULL_MASK     0x00010000UL /* Xmit status fifo full */
00159 #define XAT_IIX_RSFFIFO_FULL_MASK    0x00008000UL /* Recv status fifo full */
00160 #define XAT_IIX_XLFIFO_FULL_MASK     0x00004000UL /* Xmit length fifo full */
00161 #define XAT_IIX_RLFIFO_FULL_MASK     0x00002000UL /* Xmit length fifo full */
00162 #define XAT_IIX_XSFIFO_OVER_MASK     0x00001000UL /* Xmit status fifo overrun */
00163 #define XAT_IIX_RSFFIFO_OVER_MASK    0x00000800UL /* Xmit status fifo overrun */
00164 #define XAT_IIX_XLFIFO_OVER_MASK     0x00000400UL /* Xmit length fifo overrun */
00165 #define XAT_IIX_RLFIFO_OVER_MASK     0x00000200UL /* Recv length fifo overrun */
00166 #define XAT_IIX_XSFIFO_UNDER_MASK    0x00000100UL /* Xmit status fifo underrun */
00167 #define XAT_IIX_RSFFIFO_UNDER_MASK   0x00000080UL /* Recv status fifo underrun */
00168 #define XAT_IIX_XLFIFO_UNDER_MASK    0x00000040UL /* Xmit length fifo underrun */
00169 #define XAT_IIX_RLFIFO_UNDER_MASK    0x00000020UL /* Recv length fifo underrun */
00170 #define XAT_IIX_XDFIFO_EMPTY_MASK    0x00000010UL /* Xmit data fifo empty */
00171 #define XAT_IIX_RDFIFO_EMPTY_MASK    0x00000008UL /* Recv data fifo empty */
00172 #define XAT_IIX_XDFIFO_FULL_MASK     0x00000004UL /* Xmit data fifo full */
00173 #define XAT_IIX_RDFIFO_FULL_MASK     0x00000002UL /* Recv data fifo full */
00174 #define XAT_IIX_COUNT_MAX_MASK       0x00000001UL /* Recv errored count max */
00175
00176
00177 /***** Type Definitions *****/
00178
00179 /***** Macros (Inline Functions) Definitions *****/
00180
00181 /*****
00182 *
00183 * Low-level driver macros and functions. The list below provides signatures
00184 * to help the user use the macros.
00185 *
00186 * Xuint32 XAtmc_mReadReg(Xuint32 BaseAddress, int RegOffset)
00187 * void XAtmc_mWriteReg(Xuint32 BaseAddress, int RegOffset, Xuint32 Mask)
00188 *
00189 * void XAtmc_mEnable(Xuint32 BaseAddress)
00190 * void XAtmc_mDisable(Xuint32 BaseAddress)
00191 *
00192 * void XAtmc_SendCell(Xuint32 BaseAddress, Xuint8 *CellPtr, int Size)
00193 * int XAtmc_RecvCell(Xuint32 BaseAddress, Xuint8 *CellPtr)
00194 *
00195 * Xboolean XAtmc_mIsTxDone(Xuint32 BaseAddress)
00196 * Xboolean XAtmc_mIsRxEmpty(Xuint32 BaseAddress)
00197 *
00198 *****/

```

```

00199
00200 /*****
00201 /**
00202 *
00203 * Read the given register.
00204 *
00205 * @param   BaseAddress is the base address of the device
00206 * @param   RegOffset is the register offset to be read
00207 *
00208 * @return  The 32-bit value of the register
00209 *
00210 * @note   None.
00211 *
00212 *****/
00213 #define XAtmc_mReadReg(BaseAddress, RegOffset) \
00214         XIo_In32((BaseAddress) + (RegOffset))
00215
00216
00217 /*****
00218 /**
00219 *
00220 * Write the given register.
00221 *
00222 * @param   BaseAddress is the base address of the device
00223 * @param   RegOffset is the register offset to be written
00224 * @param   Data is the 32-bit value to write to the register
00225 *
00226 * @return  None.
00227 *
00228 * @note   None.
00229 *
00230 *****/
00231 #define XAtmc_mWriteReg(BaseAddress, RegOffset, Data) \
00232         XIo_Out32((BaseAddress) + (RegOffset), (Data))
00233
00234
00235 /*****
00236 /**
00237 *
00238 * Enable the transmitter and receiver. Preserve the contents of the control
00239 * register.
00240 *
00241 * @param   BaseAddress is the base address of the device
00242 *
00243 * @return  None.
00244 *
00245 * @note   None.
00246 *
00247 *****/
00248 #define XAtmc_mEnable(BaseAddress) \
00249 { \
00250     Xuint32 Control; \

```

```

00251     Control = XIo_In32((BaseAddress) + XAT_ATCR_OFFSET); \
00252     Control |= (XAT_ATCR_XMIT_ENABLE_MASK | XAT_ATCR_RECV_ENABLE_MASK); \
00253     XIo_Out32((BaseAddress) + XAT_ATCR_OFFSET, Control); \
00254 }
00255
00256
00257 /*****
00258 /**
00259 *
00260 * Disable the transmitter and receiver. Preserve the contents of the control
00261 * register.
00262 *
00263 * @param   BaseAddress is the base address of the device
00264 *
00265 * @return  None.
00266 *
00267 * @note    None.
00268 *
00269 *****/
00270 #define XAtmc_mDisable(BaseAddress) \
00271     XIo_Out32((BaseAddress) + XAT_ATCR_OFFSET, \
00272     XIo_In32((BaseAddress) + XAT_ATCR_OFFSET) & \
00273     ~(XAT_ATCR_XMIT_ENABLE_MASK | XAT_ATCR_RECV_ENABLE_MASK))
00274
00275
00276 /*****
00277 /**
00278 *
00279 * Check to see if the transmission is complete.
00280 *
00281 * @param   BaseAddress is the base address of the device
00282 *
00283 * @return  XTRUE if it is done, or XFALSE if it is not.
00284 *
00285 * @note    None.
00286 *
00287 *****/
00288 #define XAtmc_mIsTxDone(BaseAddress) \
00289     (XIo_In32((BaseAddress) + XAT_ISR_OFFSET) & XAT_IIX_XMIT_DONE_MASK)
00290
00291
00292 /*****
00293 /**
00294 *
00295 * Check to see if the receive FIFO is empty.
00296 *
00297 * @param   BaseAddress is the base address of the device
00298 *
00299 * @return  XTRUE if it is empty, or XFALSE if it is not.
00300 *
00301 * @note    None.
00302 *

```

```
00303 *****/
00304 #define XAtmc_mIsRxEmpty(BaseAddress) \
00305     (!(XIo_In32((BaseAddress) + XAT_ISR_OFFSET) & XAT_IIX_RECV_DONE_MASK))
00306
00307
00308 /***** Function Prototypes *****/
00309
00310 void XAtmc_SendCell(Xuint32 BaseAddress, Xuint8 *CellPtr, int Size);
00311 int XAtmc_RecvCell(Xuint32 BaseAddress, Xuint8 *CellPtr, Xuint32
*CellStatusPtr);
00312
00313
00314 #endif /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

atmc/v1_00_c/src/xatmc_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xatmc.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00b	rpm	05/01/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XAtmc_mReadReg(BaseAddress, RegOffset)  
#define XAtmc_mWriteReg(BaseAddress, RegOffset, Data)  
#define XAtmc_mEnable(BaseAddress)  
#define XAtmc_mDisable(BaseAddress)  
#define XAtmc_mIsTxDone(BaseAddress)  
#define XAtmc_mIsRxEmpty(BaseAddress)
```

Functions

void **XAtmc_SendCell** (**Xuint32** BaseAddress, **Xuint8** *CellPtr, int Size)

int **XAtmc_RecvCell** (**Xuint32** BaseAddress, **Xuint8** *CellPtr, **Xuint32** *CellStatusPtr)

Define Documentation

```
#define XAtmc_mDisable( BaseAddress )
```

Disable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XAtmc_mEnable( BaseAddress )
```

Enable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XAtmc_mIsRxEmpty( BaseAddress )
```

Check to see if the receive FIFO is empty.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is empty, or XFALSE if it is not.

Note:

None.

```
#define XAtmc_mIsTxDone( BaseAddress )
```

Check to see if the transmission is complete.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is done, or XFALSE if it is not.

Note:

None.

```
#define XAtmc_mReadReg( BaseAddress,  
                        RegOffset )
```

Read the given register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the register offset to be read

Returns:

The 32-bit value of the register

Note:

None.

```
#define XAtmc_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data      )
```

Write the given register.

Parameters:

BaseAddress is the base address of the device
RegOffset is the register offset to be written
Data is the 32-bit value to write to the register

Returns:

None.

Note:

None.

Function Documentation

```
int XAtmc_RecvCell( Xuint32  BaseAddress,  
                  Xuint8 *  CellPtr,  
                  Xuint32 *  CellStatusPtr  
                  )
```

Receive a cell. Wait for a cell to arrive.

Parameters:

BaseAddress is the base address of the device
CellPtr is a pointer to a word-aligned buffer where the cell will be stored.
CellStatusPtr is a pointer to a cell status that will be valid after this function returns.

Returns:

The size, in bytes, of the cell received.

Note:

None.


```
void XAtmc_SendCell( Xuint32  BaseAddress,  
                    Xuint8 * CellPtr,  
                    int      Size  
                    )
```

Send an ATM cell. This function blocks waiting for the cell to be transmitted.

Parameters:

BaseAddress is the base address of the device

CellPtr is a pointer to word-aligned cell

Size is the size, in bytes, of the cell

Returns:

None.

Note:

None.

atmc/v1_00_c/src/xatmc_l.c File Reference

Detailed Description

This file contains low-level polled functions to send and receive ATM cells.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	05/01/02	First release

```
#include "xatmc_l.h"
```

Functions

```
void XAtmc_SendCell (Xuint32 BaseAddress, Xuint8 *CellPtr, int Size)  
int XAtmc_RecvCell (Xuint32 BaseAddress, Xuint8 *CellPtr, Xuint32 *CellStatusPtr)
```

Function Documentation

```
int XAtmc_RecvCell( Xuint32 BaseAddress,  
                   Xuint8 * CellPtr,  
                   Xuint32 * CellStatusPtr  
                   )
```

Receive a cell. Wait for a cell to arrive.

Parameters:

BaseAddress is the base address of the device

CellPtr is a pointer to a word-aligned buffer where the cell will be stored.

CellStatusPtr is a pointer to a cell status that will be valid after this function returns.

Returns:

The size, in bytes, of the cell received.

Note:

None.

```
void XAtmc_SendCell( Xuint32 BaseAddress,  
                    Xuint8 * CellPtr,  
                    int Size  
                    )
```

Send an ATM cell. This function blocks waiting for the cell to be transmitted.

Parameters:

BaseAddress is the base address of the device

CellPtr is a pointer to word-aligned cell

Size is the size, in bytes, of the cell

Returns:

None.

Note:

None.

atmc/v1_00_c/src/xatmc.c File Reference

Detailed Description

This file contains the ATM controller driver. This file contains send and receive functions as well as interrupt service routines.

There is one interrupt service routine registered with the interrupt controller. This function determines the source of the interrupt and calls an appropriate handler function.

Note:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	JHL	07/31/01	First release
1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver an v1.23b of the IPIF driver

```
#include "xatmc.h"  
#include "xatmc_i.h"  
#include "xipif_v1_23_b.h"  
#include "xpacket_fifo_v2_00_a.h"  
#include "xio.h"
```

Functions

```
XStatus XAtmc_SgSend (XAtmc *InstancePtr, XBufDescriptor *BdPtr)
XStatus XAtmc_SgRecv (XAtmc *InstancePtr, XBufDescriptor *BdPtr)
XStatus XAtmc_SgGetSendCell (XAtmc *InstancePtr, XBufDescriptor **PtrToBdPtr, int
    *BdCountPtr)
XStatus XAtmc_SgGetRecvCell (XAtmc *InstancePtr, XBufDescriptor **PtrToBdPtr, int
    *BdCountPtr)
XStatus XAtmc_PollSend (XAtmc *InstancePtr, Xuint8 *BufPtr, Xuint32 ByteCount)
XStatus XAtmc_PollRecv (XAtmc *InstancePtr, Xuint8 *BufPtr, Xuint32 *ByteCountPtr, Xuint32
    *CellStatusPtr)
void XAtmc_InterruptHandler (void *InstancePtr)
```

Function Documentation

```
void XAtmc_InterruptHandler( void * InstancePtr)
```

Interrupt handler for the ATM controller driver. It performs the following processing:

- Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: the ATM controller, the send packet FIFO, the receive packet FIFO, the send DMA channel, or the receive DMA channel. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the ATM controller.
- Call the appropriate handler based on the source of the interrupt.

Parameters:

InstancePtr contains a pointer to the ATMC controller instance for the interrupt.

Returns:

None.

Note:

None.

```

XStatus XAtmc_PollRecv( XAtmc * InstancePtr,
                        Xuint8 * BufPtr,
                        Xuint32 * ByteCountPtr,
                        Xuint32 * CellStatusPtr
                    )

```

Receives an ATM cell in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the cell directly from the ATM controller packet FIFO. This is a non-blocking receive, in that if there is no cell ready to be received at the device, the function returns with an error. The buffer into which the cell will be received must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received Atmc cell will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to the size of the buffer on entry into the function and the size the received cell on return from the function.

CellStatusPtr is both an input and an output parameter. It is a pointer to the status of the cell which is received. It is only valid if the return value indicates success. The status is necessary when cells with errors are not being discarded. This status is a bit mask which may contain one or more of the following values with the exception of XAT_CELL_STATUS_NO_ERROR which is mutually exclusive. The status values are:

- XAT_CELL_STATUS_NO_ERROR indicates the cell was received without any errors
- XAT_CELL_STATUS_BAD_PARITY indicates the cell parity was not correct
- XAT_CELL_STATUS_BAD_HEC indicates the cell HEC was not correct
- XAT_CELL_STATUS_SHORT indicates the cell was not the correct length
- XAT_CELL_STATUS_VXI_MISMATCH indicates the cell VPI/VCI fields did not match the expected header values

Returns:

- XST_SUCCESS if the cell was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no cell to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the cell is too small for the cell waiting in the FIFO.

Note:

The input buffer must be big enough to hold the largest ATM cell. The buffer must also be 32-bit aligned.

```
XStatus XAtmc_PollSend( XAtmc * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 ByteCount  
                        )
```

Sends an ATM cell in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the cell directly to the ATM controller packet FIFO, then enters a loop checking the device status for completion or error. The buffer to be sent must be word-aligned.

It is assumed that the upper layer software supplies a correctly formatted ATM cell based upon the configuration of the ATM controller (attaching header or not).

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the ATM cell to be sent.

ByteCount is the size of the ATM cell. An ATM cell for a 16 bit Utopia interface is 54 bytes with a 6 byte header and 48 bytes of payload. This function may be used to send short cells with or without headers depending on the configuration of the ATM controller.

Returns:

- XST_SUCCESS if the cell was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_PFIFO_NO_ROOM if there is no room in the FIFO for this cell
- XST_FIFO_ERROR if the FIFO was overrun or underrun

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread.

The input buffer must be big enough to hold the largest ATM cell. The buffer must also be 32-bit aligned.

```
XStatus XAtmc_SgGetRecvCell( XAtmc * InstancePtr,  
                             XBufDescriptor ** PtrToBdPtr,  
                             int * BdCountPtr  
                             )
```

Gets the first buffer descriptor of the oldest cell which was received by the scatter-gather DMA channel of the ATM controller. This function is provided to be called from a callback function such that the buffer descriptors for received cells can be processed. The function should be called by the application repetitively for the number of cells indicated as an argument in the callback function. This function may also be used when only payloads are being sent and received by the ATM controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the cell. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the cell. This input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the cell pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus XAtmc_SgGetSendCell( XAtmc * InstancePtr,  
                              XBufDescriptor ** PtrToBdPtr,  
                              int * BdCountPtr  
                              )
```


Gets the first buffer descriptor of the oldest cell which was sent by the scatter-gather DMA channel of the ATM controller. This function is provided to be called from a callback function such that the buffer descriptors for sent cells can be processed. The function should be called by the application repetitively for the number of cells indicated as an argument in the callback function. This function may also be used when only payloads are being sent and received by the ATM controller.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

PtrToBdPtr is a pointer to a buffer descriptor pointer which will be modified to point to the first buffer descriptor of the cell. This input argument is also an output.

BdCountPtr is a pointer to a buffer descriptor count which will be modified to indicate the number of buffer descriptors for the cell. this input argument is also an output.

Returns:

A status is returned which contains one of values below. The pointer to a buffer descriptor pointed to by *PtrToBdPtr* and a count of the number of buffer descriptors for the cell pointed to by *BdCountPtr* are both modified if the return status indicates success. The status values are:

- XST_SUCCESS if a descriptor was successfully returned to the driver.
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode.
- XST_DMA_SG_NO_LIST if the scatter gather list has not been created.
- XST_DMA_SG_LIST_EMPTY if no buffer descriptor was retrieved from the list because there are no buffer descriptors to be processed in the list.

Note:

None.

```
XStatus XAtmc_SgRecv( XAtmc * InstancePtr,  
                    XBufDescriptor * BdPtr  
                    )
```

Adds this descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of cells to replace filled buffers with empty buffers. The contents of the specified buffer descriptor are copied into the scatter-gather transmit list. This function can be called when the device is started or stopped.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

Note:

None.

```
XStatus XAtmc_SgSend( XAtmc * InstancePtr,  
                    XBufDescriptor * BdPtr  
                    )
```

Sends an ATM cell using scatter-gather DMA. The caller attaches the cell to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire ATM cell may or may not be contained within one descriptor. The contents of the buffer descriptor are copied into the scatter-gather transmit list. The caller is responsible for providing mutual exclusion to guarantee that a cell is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the cell, the inserts are committed, which means the descriptors for this cell are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted ATM cell based upon the configuration of the ATM controller (attaching header or not). The ATM controller must be started before calling this function.

Parameters:

InstancePtr is a pointer to the **XAtmc** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Returns:

- XST_SUCCESS if the buffer was successfully sent
- XST_DEVICE_IS_STOPPED if the ATM controller has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

atmc/v1_00_c/src/xatmc_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of ATMC devices in the system. Each ATMC device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	JHL	07/31/01	First release
1.00b	rpm	05/01/02	Condensed base addresses into one
1.00c	rpm	01/08/03	New release supports v2.00a of packet fifo driver an v1.23b of the IPIF driver

```
#include "xatmc.h"  
#include "xparameters.h"
```

Variables

[XAtmc_Config](#) [XAtmc_ConfigTable](#) [XPAR_XATMC_NUM_INSTANCES]

Variable Documentation

XAtmc_Config XAtmc_ConfigTable[XPAR_XATMC_NUM_INSTANCES]

This table contains configuration information for each ATMC device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

emac/v1_00_c/src/xemac.h File Reference

Detailed Description

The Xilinx Ethernet driver component. This component supports the Xilinx Ethernet 10/100 MAC (EMAC).

The Xilinx Ethernet 10/100 MAC supports the following features:

- Simple and scatter-gather DMA operations, as well as simple memory mapped direct I/O interface (FIFOs).
- Media Independent Interface (MII) for connection to external 10/100 Mbps PHY transceivers.
- MII management control reads and writes with MII PHYs
- Independent internal transmit and receive FIFOs
- CSMA/CD compliant operations for half-duplex modes
- Programmable PHY reset signal
- Unicast, broadcast, and promiscuous address filtering (no multicast yet)
- Internal loopback
- Automatic source address insertion or overwrite (programmable)
- Automatic FCS insertion and stripping (programmable)
- Automatic pad insertion and stripping (programmable)
- Pause frame (flow control) detection in full-duplex mode
- Programmable interframe gap
- VLAN frame support.
- Pause frame support

The device driver supports all the features listed above.

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to the EMAC. The driver handles transmission and reception of Ethernet frames, as well as configuration of the controller. It does not handle protocol stack functionality such as Link Layer Control (LLC) or the Address Resolution Protocol (ARP). The protocol stack that makes use of the driver handles this functionality. This implies that the driver is simply a pass-through mechanism between a protocol stack and the EMAC. A single device driver can support multiple EMACs.

The driver is designed for a zero-copy buffer scheme. That is, the driver will not copy buffers. This avoids potential throughput bottlenecks within the driver.

Since the driver is a simple pass-through mechanism between a protocol stack and the EMAC, no assembly or disassembly of Ethernet frames is done at the driver-level. This assumes that the protocol stack passes a correctly formatted Ethernet frame to the driver for transmission, and that the driver does not validate the contents of an incoming frame

PHY Communication

The driver provides rudimentary read and write functions to allow the higher layer software to access the PHY. The EMAC provides MII registers for the driver to access. This management interface can be parameterized away in the FPGA implementation process. If this is the case, the PHY read and write functions of the driver return XST_NO_FEATURE.

External loopback is usually supported at the PHY. It is up to the user to turn external loopback on or off at the PHY. The driver simply provides pass-through functions for configuring the PHY. The driver does not read, write, or reset the PHY on its own. All control of the PHY must be done by the user.

Asynchronous Callbacks

The driver services interrupts and passes Ethernet frames to the higher layer software through asynchronous callback functions. When using the driver directly (i.e., not with the RTOS protocol stack), the higher layer software must register its callback functions during initialization. The driver requires callback functions for received frames, for confirmation of transmitted frames, and for asynchronous errors.

Interrupts

The driver has no dependencies on the interrupt controller. The driver provides two interrupt handlers.

[XEmac_IntrHandlerDma\(\)](#) handles interrupts when the EMAC is configured with scatter-gather DMA.

[XEmac_IntrHandlerFifo\(\)](#) handles interrupts when the EMAC is configured for direct FIFO I/O or simple DMA. Either of these routines can be connected to the system interrupt controller by the user.

Interrupt Frequency

When the EMAC is configured with scatter-gather DMA, the frequency of interrupts can be controlled with the interrupt coalescing features of the scatter-gather DMA engine. The frequency of interrupts can be adjusted using the driver API functions for setting the packet count threshold and the packet wait bound values.

The scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

The packet wait bound is a timer value used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold.

These values can be tuned by the user to meet their needs. If there appear to be interrupt latency problems or delays in packet arrival that are longer than might be expected, the user should verify that the packet count threshold is set low enough to receive interrupts before the wait bound timer goes off.

Device Reset

Some errors that can occur in the device require a device reset. These errors are listed in the [XEmac_SetErrorHandler\(\)](#) function header. The user's error handler is responsible for resetting the device and re-configuring it based on its needs (the driver does not save the current configuration). When integrating into an RTOS, these reset and re-configure obligations are taken care of by the Xilinx adapter software if it exists for that RTOS.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the [xemac_g.c](#) files. A table is defined where each entry contains configuration information for an EMAC device. This

information includes such things as the base address of the memory-mapped device, the base addresses of IPIF, DMA, and FIFO modules within the device, and whether the device has DMA, counter registers, multicast support, MII support, and flow control.

The driver tries to use the features built into the device. So if, for example, the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has set up the buffer descriptor lists already. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used. Modifying the configuration table is a workaround for now until we get some experience with how users are intending to use the hardware in its different configurations. For example, if the hardware is built with scatter-gather DMA but the user is intending to use only simple DMA, the user either needs to modify the config table as a workaround or rebuild the hardware with only simple DMA. The recommendation at this point is to build the hardware with the features you intend to use. If you're inclined to modify the table, do so before the call to `XEmac_Initialize()`. Here is a snippet of code that changes a device to simple DMA (the hardware needs to have DMA for this to work of course):

```
XEmac_Config *ConfigPtr;

ConfigPtr = XEmac_LookupConfig(DeviceId);
ConfigPtr->IpIfDmaConfig = XEM_CFG_SIMPLE_DMA;
```

Simple DMA

Simple DMA is supported through the FIFO functions, `FifoSend` and `FifoRecv`, of the driver (i.e., there is no separate interface for it). The driver makes use of the DMA engine for a simple DMA transfer if the device is configured with DMA, otherwise it uses the FIFOs directly. While the simple DMA interface is therefore transparent to the user, the caching of network buffers is not. If the device is configured with DMA and the FIFO interface is used, the user must ensure that the network buffers are not cached or are cache coherent, since DMA will be used to transfer to and from the Emac device. If the device is configured with DMA and the user really wants to use the FIFOs directly, the user should rebuild the hardware without DMA. If unable to do this, there is a workaround (described above in Device Configuration) to modify the configuration table of the driver to fake the driver into thinking the device has no DMA. A code snippet follows:

```
XEmac_Config *ConfigPtr;

ConfigPtr = XEmac_LookupConfig(DeviceId);
ConfigPtr->IpIfDmaConfig = XEM_CFG_NO_DMA;
```

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the `NDEBUG` identifier. By default, asserts are turned on and it is recommended that users leave asserts on during development.

Building the driver

The `XEmac` driver is composed of several source files. Why so many? This allows the user to build and link only those parts of the driver that are necessary. Since the EMAC hardware can be configured in various ways (e.g., with or without DMA), the driver too can be built with varying features. For the most part, this means that besides always linking in `xemac.c`, you link in only the driver functionality you want. Some of the choices you have are polled vs. interrupt, interrupt with FIFOs only vs. interrupt with DMA, self-test diagnostics, and driver statistics. Note that currently the DMA code must be linked in, even if

you don't have DMA in the device.

Note:

Xilinx drivers are typically composed of two components, one is the driver and the other is the adapter. The driver is independent of OS and processor and is intended to be highly portable. The adapter is OS-specific and facilitates communication between the driver and an OS.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00b	rpm	10/08/02	Replaced HasSgDma boolean with IpifDmaConfig enumerated configuration parameter
1.00c	rpm	12/05/02	New version includes support for simple DMA and the delay argument to SgSend
1.00c	rpm	02/03/03	The XST_DMA_SG_COUNT_EXCEEDED return code was removed from SetPktThreshold in the internal DMA driver. Also avoided compiler warnings by initializing Result in the DMA interrupt service routines.

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xparameters.h"
#include "xpacket_fifo_v1_00_b.h"
#include "xdma_channel.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XEmac
struct XEmac_Config
struct XEmac_Stats
```

Configuration options

Device configuration options (see the [XEmac_SetOptions\(\)](#) and [XEmac_GetOptions\(\)](#) for information on how to use these options)

```
#define XEM_UNICAST_OPTION
```

```

#define XEM_BROADCAST_OPTION
#define XEM_PROMISC_OPTION
#define XEM_FDUPLEX_OPTION
#define XEM_POLLED_OPTION
#define XEM_LOOPBACK_OPTION
#define XEM_FLOW_CONTROL_OPTION
#define XEM_INSERT_PAD_OPTION
#define XEM_INSERT_FCS_OPTION
#define XEM_INSERT_ADDR_OPTION
#define XEM_OVWRT_ADDR_OPTION
#define XEM_STRIP_PAD_FCS_OPTION

```

Typedefs for callbacks

Callback functions.

```

typedef void(* XEmac_SgHandler )(void *CallBackRef, XBufDescriptor *BdPtr, Xuint32 NumBds)
typedef void(* XEmac_FifoHandler )(void *CallBackRef)
typedef void(* XEmac_ErrorHandler )(void *CallBackRef, XStatus ErrorCode)

```

Defines

```

#define XEmac_mIsSgDma(InstancePtr)
#define XEmac_mIsSimpleDma(InstancePtr)
#define XEmac_mIsDma(InstancePtr)

```

Functions

```

XStatus XEmac_Initialize (XEmac *InstancePtr, Xuint16 DeviceId)
XStatus XEmac_Start (XEmac *InstancePtr)
XStatus XEmac_Stop (XEmac *InstancePtr)
void XEmac_Reset (XEmac *InstancePtr)
XEmac_Config * XEmac_LookupConfig (Xuint16 DeviceId)
XStatus XEmac_SelfTest (XEmac *InstancePtr)
XStatus XEmac_PollSend (XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 ByteCount)
XStatus XEmac_PollRecv (XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 *ByteCountPtr)
XStatus XEmac_SgSend (XEmac *InstancePtr, XBufDescriptor *BdPtr, int Delay)
XStatus XEmac_SgRecv (XEmac *InstancePtr, XBufDescriptor *BdPtr)
XStatus XEmac_SetPktThreshold (XEmac *InstancePtr, Xuint32 Direction, Xuint8 Threshold)
XStatus XEmac_GetPktThreshold (XEmac *InstancePtr, Xuint32 Direction, Xuint8 *ThreshPtr)
XStatus XEmac_SetPktWaitBound (XEmac *InstancePtr, Xuint32 Direction, Xuint32 TimerValue)
XStatus XEmac_GetPktWaitBound (XEmac *InstancePtr, Xuint32 Direction, Xuint32 *WaitPtr)
XStatus XEmac_SetSgRecvSpace (XEmac *InstancePtr, Xuint32 *MemoryPtr, Xuint32 ByteCount)

```

XStatus XEmac_SetSgSendSpace (XEmac *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)
 void **XEmac_SetSgRecvHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_SgHandler** FuncPtr)
 void **XEmac_SetSgSendHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_SgHandler** FuncPtr)
 void **XEmac_IntrHandlerDma** (void *InstancePtr)
XStatus XEmac_FifoSend (XEmac *InstancePtr, **Xuint8** *BufPtr, **Xuint32** ByteCount)
XStatus XEmac_FifoRecv (XEmac *InstancePtr, **Xuint8** *BufPtr, **Xuint32** *ByteCountPtr)
 void **XEmac_SetFifoRecvHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_FifoHandler** FuncPtr)
 void **XEmac_SetFifoSendHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_FifoHandler** FuncPtr)
 void **XEmac_IntrHandlerFifo** (void *InstancePtr)
 void **XEmac_SetErrorHandler** (XEmac *InstancePtr, void *CallBackRef, **XEmac_ErrorHandler** FuncPtr)
XStatus XEmac_SetOptions (XEmac *InstancePtr, **Xuint32** OptionFlag)
Xuint32 XEmac_GetOptions (XEmac *InstancePtr)
XStatus XEmac_SetMacAddress (XEmac *InstancePtr, **Xuint8** *AddressPtr)
 void **XEmac_GetMacAddress** (XEmac *InstancePtr, **Xuint8** *BufferPtr)
XStatus XEmac_SetInterframeGap (XEmac *InstancePtr, **Xuint8** Part1, **Xuint8** Part2)
 void **XEmac_GetInterframeGap** (XEmac *InstancePtr, **Xuint8** *Part1Ptr, **Xuint8** *Part2Ptr)
XStatus XEmac_MulticastAdd (XEmac *InstancePtr, **Xuint8** *AddressPtr)
XStatus XEmac_MulticastClear (XEmac *InstancePtr)
XStatus XEmac_PhyRead (XEmac *InstancePtr, **Xuint32** PhyAddress, **Xuint32** RegisterNum, **Xuint16** *PhyDataPtr)
XStatus XEmac_PhyWrite (XEmac *InstancePtr, **Xuint32** PhyAddress, **Xuint32** RegisterNum, **Xuint16** PhyData)
 void **XEmac_GetStats** (XEmac *InstancePtr, **XEmac_Stats** *StatsPtr)
 void **XEmac_ClearStats** (XEmac *InstancePtr)

Define Documentation

#define XEM_BROADCAST_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_FDUPLEX_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_FLOW_CONTROL_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_INSERT_ADDR_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_INSERT_FCS_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_INSERT_PAD_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_LOOPBACK_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_OVWRT_ADDR_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_POLLED_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_PROMISC_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_STRIP_PAD_FCS_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEM_UNICAST_OPTION

XEM_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XEM_UNICAST_OPTION	Unicast addressing on or off (default is on)
XEM_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XEM_FDUPLEX_OPTION	Full duplex on or off (default is off)
XEM_POLLED_OPTION	Polled mode on or off (default is off)
XEM_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XEM_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XEM_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XEM_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XEM_INSERT_ADDR_OPTION	Insert source address on transmit (default is on)
XEM_OVWRT_ADDR_OPTION	Overwrite source address on transmit. This is only used if source address insertion is on. (default is on)
XEM_STRIP_PAD_FCS_OPTION	Strip FCS and padding from received frames (default is off)

#define XEmac_mIsDma(InstancePtr)

This macro determines if the device is currently configured with DMA (either simple DMA or scatter-gather DMA)

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

Boolean XTRUE if the device is configured with DMA, or XFALSE otherwise

Note:

Signature: Xboolean **XEmac_mIsDma**(XEmac *InstancePtr)

#define XEmac_mIsSgDma(InstancePtr)

This macro determines if the device is currently configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for scatter-gather DMA, or XFALSE if it is not.

Note:

Signature: Xboolean **XEmac_mIsSgDma**(XEmac *InstancePtr)

#define XEmac_mIsSimpleDma(InstancePtr)

This macro determines if the device is currently configured for simple DMA.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for simple DMA, or XFALSE otherwise

Note:

Signature: Xboolean [XEmac_mIsSimpleDma](#)(XEmac *InstancePtr)

Typedef Documentation

```
typedef void(* XEmac_ErrorHandler)(void *CallbackRef, XStatus ErrorCode)
```

Callback when an asynchronous error occurs.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

ErrorCode is a Xilinx error code defined in [xstatus.h](#). Also see [XEmac_SetErrorHandler\(\)](#) for a description of possible errors.

```
typedef void(* XEmac_FifoHandler)(void *CallbackRef)
```

Callback when data is sent or received with direct FIFO communication or simple DMA. The user typically defines two callacks, one for send and one for receive.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

```
typedef void(* XEmac_SgHandler)(void *CallbackRef, XBufDescriptor *BdPtr, Xuint32 NumBds)
```

Callback when data is sent or received with scatter-gather DMA.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

BdPtr is a pointer to the first buffer descriptor in a list of buffer descriptors.

NumBds is the number of buffer descriptors in the list pointed to by BdPtr.

Function Documentation

```
void XEmac_ClearStats( XEmac * InstancePtr)
```

Clear the XEmacStats structure for this driver.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus XEmac_FifoRecv( XEmac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 * ByteCountPtr  
                        )
```

Receive an Ethernet frame into the buffer passed as an argument. This function is called in response to the callback function for received frames being called by the driver. The callback function is set up using `SetFifoRecvHandler`, and is invoked when the driver receives an interrupt indicating a received frame. The driver expects the upper layer software to call this function, `FifoRecv`, to receive the frame. The buffer supplied should be large enough to hold a maximum-size Ethernet frame.

The buffer into which the frame will be received must be word-aligned.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from the Emac to memory. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xemac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- `XST_SUCCESS` if the frame was sent successfully
- `XST_DEVICE_IS_STOPPED` if the device has not yet been started
- `XST_NOT_INTERRUPT` if the device is not in interrupt mode
- `XST_NO_DATA` if there is no frame to be received from the FIFO
- `XST_BUFFER_TOO_SMALL` if the buffer to receive the frame is too small for the frame waiting in the FIFO.
- `XST_DEVICE_BUSY` if configured for simple DMA and the DMA engine is busy
- `XST_DMA_ERROR` if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

The input buffer must be big enough to hold the largest Ethernet frame.

```
XStatus XEmac_FifoSend( XEmac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 ByteCount  
                        )
```

Send an Ethernet frame using direct FIFO I/O or simple DMA with interrupts. The caller provides a contiguous-memory buffer and its length. The buffer must be word-aligned. The callback function set by using `SetFifoSendHandler` is invoked when the transmission is complete.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from memory to the Emac. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xemac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was successfully sent. An interrupt is generated when the EMAC transmits the frame and the driver calls the callback set with `XEmac_SetFifoSendHandler()`
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_FIFO_NO_ROOM if there is no room in the FIFO for this frame
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```
void XEmac_GetInterframeGap( XEmac * InstancePtr,  
                             Xuint8 * Part1Ptr,  
                             Xuint8 * Part2Ptr  
                             )
```

Get the interframe gap, parts 1 and 2. See the description of interframe gap above in [XEmac_SetInterframeGap\(\)](#).

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Part1Ptr is a pointer to an 8-bit buffer into which the interframe gap part 1 value will be copied.

Part2Ptr is a pointer to an 8-bit buffer into which the interframe gap part 2 value will be copied.

Returns:

None. The values of the interframe gap parts are copied into the output parameters.

```
void XEmac_GetMacAddress( XEmac * InstancePtr,  
                        Xuint8 * BufferPtr  
                        )
```

Get the MAC address for this driver/device.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

BufferPtr is an output parameter, and is a pointer to a buffer into which the current MAC address will be copied. The buffer must be at least 6 bytes.

Returns:

None.

Note:

None.

```
Xuint32 XEmac_GetOptions( XEmac * InstancePtr)
```

Get Ethernet driver/device options. The 32-bit value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

The 32-bit value of the Ethernet options. The value is a bit-mask representing all options that are currently enabled. See [xemac.h](#) for a description of the available options.

Note:

None.

```
XStatus XEmac_GetPktThreshold( XEmac * InstancePtr,  
                                Xuint32 Direction,  
                                Xuint8 * ThreshPtr  
                                )
```

Get the value of the packet count threshold for this driver/device. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
XStatus XEmac_GetPktWaitBound( XEmac * InstancePtr,  
                                Xuint32 Direction,  
                                Xuint32 * WaitPtr  
                                )
```

Get the packet wait bound timer for this driver/device. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void XEmac_GetStats( XEmac * InstancePtr,  
                   XEmac_Stats * StatsPtr  
                   )
```

Get a copy of the XEmacStats structure, which contains the current statistics for this driver. The statistics are only cleared at initialization or on demand using the [XEmac_ClearStats\(\)](#) function.

The DmaErrors and FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the upper layer software.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

```
XStatus XEmac_Initialize( XEmac * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initialize a specific [XEmac](#) instance/driver. The initialization entails:

- Initialize fields of the [XEmac](#) structure
- Clear the Ethernet statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- If the device is configured with DMA, configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists may be passed to the driver.
- Reset the Ethernet MAC

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

DeviceId is the unique id of the device controlled by this [XEmac](#) instance. Passing in a device id associates the generic [XEmac](#) instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

void XEmac_IntrHandlerDma(void * *InstancePtr*)

The interrupt handler for the Ethernet driver when configured with scatter- gather DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, Send Packet FIFO, Recv DMA channel, or Send DMA channel. The packet FIFOs only interrupt during "deadlock" conditions.

Parameters:

InstancePtr is a pointer to the **XEmac** instance that just interrupted.

Returns:

None.

Note:

None.

void XEmac_IntrHandlerFifo(void * *InstancePtr*)

The interrupt handler for the Ethernet driver when configured for direct FIFO communication or simple DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, or Send Packet FIFO. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the MAC.

Parameters:

InstancePtr is a pointer to the **XEmac** instance that just interrupted.

Returns:

None.

Note:

None.

XEmac_Config* XEmac_LookupConfig(Xuint16 *DeviceId*)

Lookup the device configuration based on the unique device ID. The table `EmacConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or `XNULL` if no match is found.

Note:

None.

```
XStatus XEmac_MulticastAdd( XEmac * InstancePtr,  
                             Xuint8 * AddressPtr  
                             )
```

Add a multicast address to the list of multicast addresses from which the EMAC accepts frames. The EMAC uses a hash table for multicast address filtering. Obviously, the more multicast addresses that are added reduces the accuracy of the address filtering. The upper layer software that receives multicast frames should perform additional filtering when accuracy must be guaranteed. There is no way to retrieve a multicast address or the multicast address list once added. The upper layer software should maintain its own list of multicast addresses. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the `XEmac` instance to be worked on.

AddressPtr is a pointer to a 6-byte multicast address.

Returns:

- `XST_SUCCESS` if the multicast address was added successfully
- `XST_NO_FEATURE` if the device is not configured with multicast support
- `XST_DEVICE_IS_STARTED` if the device has not yet been stopped

Note:

Not currently supported.

```
XStatus XEmac_MulticastClear( XEmac * InstancePtr)
```

Clear the hash table used by the EMAC for multicast address filtering. The entire hash table is cleared, meaning no multicast frames will be accepted after this function is called. If this function is used to delete one or more multicast addresses, the upper layer software is responsible for adding back those addresses still needed for address filtering. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the `XEmac` instance to be worked on.

Returns:

- `XST_SUCCESS` if the multicast address list was cleared
- `XST_NO_FEATURE` if the device is not configured with multicast support

- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

Not currently supported.

```
XStatus XEmac_PhyRead( XEmac * InstancePtr,  
                      Xuint32 PhyAddress,  
                      Xuint32 RegisterNum,  
                      Xuint16 * PhyDataPtr  
                      )
```

Read the current value of the PHY register indicated by the *PhyAddress* and the *RegisterNum* parameters. The MAC provides the driver with the ability to talk to a PHY that adheres to the Media Independent Interface (MII) as defined in the IEEE 802.3 standard.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

PhyAddress is the address of the PHY to be read (supports multiple PHYs)

RegisterNum is the register number, 0-31, of the specific PHY register to read

PhyDataPtr is an output parameter, and points to a 16-bit buffer into which the current value of the register will be copied.

Returns:

- XST_SUCCESS if the PHY was read from successfully
- XST_NO_FEATURE if the device is not configured with MII support
- XST_EMAC_MII_BUSY if there is another PHY operation in progress
- XST_EMAC_MII_READ_ERROR if a read error occurred between the MAC and the PHY

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that the read is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the *PhyRead* thread.

```
XStatus XEmac_PhyWrite( XEmac * InstancePtr,  
                       Xuint32 PhyAddress,  
                       Xuint32 RegisterNum,  
                       Xuint16 PhyData  
                       )
```

Write data to the specified PHY register. The Ethernet driver does not require the device to be stopped before writing to the PHY. Although it is probably a good idea to stop the device, it is the responsibility of the application to deem this necessary. The MAC provides the driver with the ability to talk to a PHY that adheres to the Media Independent Interface (MII) as defined in the IEEE 802.3 standard.

Parameters:

- InstancePtr* is a pointer to the **XEmac** instance to be worked on.
- PhyAddress* is the address of the PHY to be written (supports multiple PHYs)
- RegisterNum* is the register number, 0-31, of the specific PHY register to write
- PhyData* is the 16-bit value that will be written to the register

Returns:

- XST_SUCCESS if the PHY was written to successfully. Since there is no error status from the MAC on a write, the user should read the PHY to verify the write was successful.
- XST_NO_FEATURE if the device is not configured with MII support
- XST_EMAC_MII_BUSY if there is another PHY operation in progress

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that the write is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PhyWrite thread.

```
XStatus XEmac_PollRecv( XEmac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 * ByteCountPtr  
                        )
```

Receive an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the frame directly from the MAC's packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The MAC's error status is not checked, so statistics are not updated for polled receive. The buffer into which the frame will be received must be word-aligned.

Parameters:

- InstancePtr* is a pointer to the **XEmac** instance to be worked on.
- BufPtr* is a pointer to a word-aligned buffer into which the received Ethernet frame will be copied.
- ByteCountPtr* is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.

Note:

Input buffer must be big enough to hold the largest Ethernet frame. Buffer must also be 32-bit aligned.

```
XStatus XEmac_PollSend( XEmac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 ByteCount  
                        )
```

Send an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the frame directly to the MAC's packet FIFO, then enters a loop checking the device status for completion or error. Statistics are updated if an error occurs. The buffer to be sent must be word-aligned.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_FIFO_NO_ROOM if there is no room in the EMAC's length FIFO for this frame
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.
- XST_EMAC_COLLISION if the send failed due to excess deferral or late collision

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread. On a 10Mbps MAC, it takes about 1.21 msec to transmit a maximum size Ethernet frame (1518 bytes). On a 100Mbps MAC, it takes about 121 usecs to transmit a maximum size Ethernet frame.

```
void XEmac_Reset( XEmac * InstancePtr)
```

Reset the Ethernet MAC. This is a graceful reset in that the device is stopped first. Resets the DMA channels, the FIFOs, the transmitter, and the receiver. The PHY is not reset. Any frames in the scatter-gather descriptor lists will remain in the lists. The side effect of doing this is that after a reset and following a restart of the device, frames that were in the list before the reset may be transmitted or received. Reset must only be called after the driver has been initialized.

The driver is also taken out of polled mode if polled mode was set. The user is responsible for re-configuring the driver into polled mode after the reset if desired.

The configuration after this reset is as follows:

- Half duplex
- Disabled transmitter and receiver
- Enabled PHY (the PHY is not reset)
- MAC transmitter does pad insertion, FCS insertion, and source address overwrite.
- MAC receiver does not strip padding or FCS
- Interframe Gap as recommended by IEEE Std. 802.3 (96 bit times)
- Unicast addressing enabled
- Broadcast addressing enabled
- Multicast addressing disabled (addresses are preserved)
- Promiscuous addressing disabled
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- MAC address of all zeros
- Non-polled mode

The upper layer software is responsible for re-configuring (if necessary) and restarting the MAC after the reset. Note that the PHY is not reset. PHY control is left to the upper layer software. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

None.

Note:

None.

XStatus XEmac_SelfTest(**XEmac** * *InstancePtr*)

Performs a self-test on the Ethernet device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the Ethernet device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_PFIFO_BAD_REG_VALUE	FIFO failed register self-test
XST_DMA_TRANSFER_ERROR	DMA failed data transfer self-test
XST_DMA_RESET_REGISTER_ERROR	DMA control register value was incorrect after a reset
XST_REGISTER_ERROR	Ethernet failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed
XST_IPIF_REG_WIDTH_ERROR	An invalid register width was passed into the function
XST_IPIF_RESET_REGISTER_ERROR	The value of a register at reset was invalid
XST_IPIF_DEVICE_STATUS_ERROR	A write to the device status register did not read back correctly
XST_IPIF_DEVICE_ACK_ERROR	A bit in the device status register did not reset when acked
XST_IPIF_DEVICE_ENABLE_ERROR	The device interrupt enable register was not updated correctly by the hardware when other registers were written to
XST_IPIF_IP_STATUS_ERROR	A write to the IP interrupt status register did not read back correctly
XST_IPIF_IP_ACK_ERROR	One or more bits in the IP status register did not reset when acked
XST_IPIF_IP_ENABLE_ERROR	The IP interrupt enable register was not updated correctly when other registers were written to

Note:

This function makes use of options-related functions, and the [XEmac_PollSend\(\)](#) and [XEmac_PollRecv\(\)](#) functions.

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread

to monitor the self-test thread.

```
void XEmac_SetErrorHandler( XEmac *           InstancePtr,  
                           void *           CallbackRef,  
                           XEmac_ErrorHandler FuncPtr  
                           )
```

Set the callback function for handling asynchronous errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable MAC error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_DMA_SG_NO_LIST indicates an attempt was made to access a scatter-gather DMA list that has not yet been created.
- XST_DMA_SG_LIST_EMPTY indicates the driver tried to get a descriptor from the receive descriptor list, but the list was empty.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XEmac_SetFifoRecvHandler( XEmac *           InstancePtr,  
                              void *           CallbackRef,  
                              XEmac_FifoHandler FuncPtr  
                              )
```

Set the callback function for handling confirmation of transmitted frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called by the driver once per frame sent. The callback is responsible for freeing the transmitted buffer if necessary.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XEmac_SetFifoSendHandler( XEmac *           InstancePtr,
                              void *           CallbackRef,
                              XEmac_FifoHandler FuncPtr
                              )
```

Set the callback function for handling received frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called once per frame received. During the callback, the upper layer software should call `FifoRecv` to retrieve the received frame.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.


```
XStatus XEmac_SetInterframeGap( XEmac * InstancePtr,  
                                Xuint8  Part1,  
                                Xuint8  Part2  
                                )
```

Set the Interframe Gap (IFG), which is the time the MAC delays between transmitting frames. There are two parts required. The total interframe gap is the total of the two parts. The values provided for the Part1 and Part2 parameters are multiplied by 4 to obtain the bit-time interval. The first part should be the first 2/3 of the total interframe gap. The MAC will reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part1. Part1 may be shorter than 2/3 the total and can be as small as zero. The second part should be the last 1/3 of the total interframe gap, but can be as large as the total interframe gap. The MAC will not reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part2.

The device must be stopped before setting the interframe gap.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Part1 is the interframe gap part 1 (which will be multiplied by 4 to get the bit-time interval).

Part2 is the interframe gap part 2 (which will be multiplied by 4 to get the bit-time interval).

Returns:

- XST_SUCCESS if the interframe gap was set successfully
- XST_DEVICE_IS_STARTED if the device has not been stopped

Note:

None.

```
XStatus XEmac_SetMacAddress( XEmac * InstancePtr,  
                             Xuint8 * AddressPtr  
                             )
```

Set the MAC address for this driver/device. The address is a 48-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address.

Returns:

- XST_SUCCESS if the MAC address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus XEmac_SetOptions( XEmac * InstancePtr,  
                          Xuint32 OptionsFlag  
                          )
```

Set Ethernet driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

OptionsFlag is a bit-mask representing the Ethernet options to turn on or off. See [xemac.h](#) for a description of the available options.

Returns:

- XST_SUCCESS if the options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
XStatus XEmac_SetPktThreshold( XEmac * InstancePtr,  
                              Xuint32 Direction,  
                              Xuint8 Threshold  
                              )
```

Set the packet count threshold for this device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

The packet threshold could be set to larger than the number of descriptors allocated to the DMA channel. In this case, the wait bound will take over and always indicate data arrival. There was a check in this function that returned

an error if the threshold was larger than the number of descriptors, but that was removed because users would then have to set the threshold only after they set descriptor space, which is an order dependency that caused confusion.

```
XStatus XEmac_SetPktWaitBound( XEmac * InstancePtr,  
                               Xuint32 Direction,  
                               Xuint32 TimerValue  
                               )
```

Set the packet wait bound timer for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void XEmac_SetSgRecvHandler( XEmac * InstancePtr,  
                             void * CallbackRef,  
                             XEmac_SgHandler FuncPtr  
                             )
```

Set the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame received. The head of a descriptor list is passed in along with the number of descriptors in the list. Before leaving the callback, the upper layer software should attach a new buffer to each descriptor in the list.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XEmac_SetSgRecvSpace( XEmac * InstancePtr,  
                             Xuint32 * MemoryPtr,  
                             Xuint32 ByteCount  
                             )
```

Give the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be word-aligned. An assert will occur if asserts are turned on and the memory is not word-aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XEmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
void XEmac_SetSgSendHandler( XEmac * InstancePtr,  
                             void * CallbackRef,  
                             XEmac_SgHandler FuncPtr  
                             )
```

Set the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame sent. The head of a descriptor list is passed in along with the number of descriptors in the list. The callback is responsible for freeing buffers attached to these descriptors.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XEmac_SetSgSendSpace( XEmac * InstancePtr,  
                               Xuint32 * MemoryPtr,  
                               Xuint32 ByteCount  
                               )
```

Give the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be word-aligned. An assert will occur if asserts are turned on and the memory is not word-aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XEmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
XStatus XEmac_SgRecv( XEmac * InstancePtr,  
                    XBufDescriptor * BdPtr  
                    )
```

Add a descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. This function can be called when the device is started or stopped. Note that it does start the scatter-gather DMA engine. Although this is not necessary during initialization, it is not a problem during initialization because the MAC receiver is not yet started.

The buffer attached to the descriptor must be word-aligned on both the front end and the back end.

Notification of received frames are done asynchronously through the receive callback function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

```
XStatus XEmac_SgSend( XEmac * InstancePtr,  
                    XBufDescriptor * BdPtr,  
                    int Delay  
                    )
```

Send an Ethernet frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire Ethernet frame may or may not be contained within one descriptor. This function simply inserts the descriptor into the scatter-gather engine's transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

The buffer attached to the descriptor must be word-aligned on the front end.

This call is non-blocking. Notification of error or successful transmission is done asynchronously through the send or error callback function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Delay indicates whether to start the scatter-gather DMA channel immediately, or whether to wait. This allows the user to build up a list of more than one descriptor before starting the transmission of the packets, which allows the application to keep up with DMA and have a constant stream of frames being transmitted. Use XEM_SGDMA_NODELAY or XEM_SGDMA_DELAY, defined in **xemac.h**, as the value of this argument. If the user chooses to delay and build a list, the user must call this function with the XEM_SGDMA_NODELAY option or call **XEmac_Start()** to kick off the transmissions.

Returns:

- XST_SUCCESS if the buffer was successful sent
- XST_DEVICE_IS_STOPPED if the Ethernet MAC has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

XStatus XEmac_Start(XEmac * InstancePtr)

Start the Ethernet controller as follows:

- If not in polled mode
 - Set the internal interrupt enable registers appropriately
 - Enable interrupts within the device itself. Note that connection of the driver's interrupt handler to the interrupt source (typically done using the interrupt controller component) is done by the higher layer software.
 - If the device is configured with scatter-gather DMA, start the DMA channels if the descriptor lists are not empty
- Enable the transmitter
- Enable the receiver

The PHY is enabled after driver initialization. We assume the upper layer software has configured it and the EMAC appropriately before this function is called.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.

- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.

Note:

The driver tries to match the hardware configuration. So if the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has set up the buffer descriptor lists already. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used. Modifying the config table is a workaround for now until we get some experience with how users are intending to use the hardware in its different configurations. For example, if the hardware is built with scatter-gather DMA but the user is intending to use only simple DMA, the user either needs to modify the config table as a workaround or rebuild the hardware with only simple DMA.

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus XEmac_Stop(XEmac * InstancePtr)

Stop the Ethernet MAC as follows:

- If the device is configured with scatter-gather DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode (the higher layer software is responsible for disabling interrupts at the interrupt controller)

The PHY is left enabled after a Stop is called.

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the Ethernet frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

emac/v1_00_c/src/xemac_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of EMAC devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA

```
#include "xemac.h"  
#include "xparameters.h"
```

Variables

[XEmac_Config](#) [XEmac_ConfigTable](#) [XPAR_XEMAC_NUM_INSTANCES]

Variable Documentation

[XEmac_Config](#) [XEmac_ConfigTable](#)[XPAR_XEMAC_NUM_INSTANCES]

This table contains configuration information for each EMAC device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

emac/v1_00_c/src/xemac.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xemac.h,v 1.3 2003/02/03 19:49:38 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *****/
00019 /*****
00020 /**
00021 *
00022 * @file emac/v1_00_c/src/xemac.h
00023 *
00024 * The Xilinx Ethernet driver component.  This component supports the Xilinx
00025 * Ethernet 10/100 MAC (EMAC).
00026 *
00027 * The Xilinx Ethernet 10/100 MAC supports the following features:
00028 * - Simple and scatter-gather DMA operations, as well as simple memory
00029 *   mapped direct I/O interface (FIFOs).
00030 * - Media Independent Interface (MII) for connection to external
00031 *   10/100 Mbps PHY transceivers.
00032 * - MII management control reads and writes with MII PHYs
00033 * - Independent internal transmit and receive FIFOs
00034 * - CSMA/CD compliant operations for half-duplex modes
00035 * - Programmable PHY reset signal
00036 * - Unicast, broadcast, and promiscuous address filtering (no multicast yet)
00037 * - Internal loopback
00038 * - Automatic source address insertion or overwrite (programmable)
00039 * - Automatic FCS insertion and stripping (programmable)
00040 * - Automatic pad insertion and stripping (programmable)
00041 * - Pause frame (flow control) detection in full-duplex mode
00042 * - Programmable interframe gap
```

00043 * - VLAN frame support.
00044 * - Pause frame support
00045 *
00046 * The device driver supports all the features listed above.
00047 *
00048 * **Driver Description**
00049 *
00050 * The device driver enables higher layer software (e.g., an application) to
00051 * communicate to the EMAC. The driver handles transmission and reception of
00052 * Ethernet frames, as well as configuration of the controller. It does not
00053 * handle protocol stack functionality such as Link Layer Control (LLC) or the
00054 * Address Resolution Protocol (ARP). The protocol stack that makes use of the
00055 * driver handles this functionality. This implies that the driver is simply a
00056 * pass-through mechanism between a protocol stack and the EMAC. A single device
00057 * driver can support multiple EMACs.
00058 *
00059 * The driver is designed for a zero-copy buffer scheme. That is, the driver
will
00060 * not copy buffers. This avoids potential throughput bottlenecks within the
00061 * driver.
00062 *
00063 * Since the driver is a simple pass-through mechanism between a protocol stack
00064 * and the EMAC, no assembly or disassembly of Ethernet frames is done at the
00065 * driver-level. This assumes that the protocol stack passes a correctly
00066 * formatted Ethernet frame to the driver for transmission, and that the driver
00067 * does not validate the contents of an incoming frame
00068 *
00069 * **PHY Communication**
00070 *
00071 * The driver provides rudimentary read and write functions to allow the higher
00072 * layer software to access the PHY. The EMAC provides MII registers for the
00073 * driver to access. This management interface can be parameterized away in the
00074 * FPGA implementation process. If this is the case, the PHY read and write
00075 * functions of the driver return XST_NO_FEATURE.
00076 *
00077 * External loopback is usually supported at the PHY. It is up to the user to
00078 * turn external loopback on or off at the PHY. The driver simply provides pass-
00079 * through functions for configuring the PHY. The driver does not read, write,
00080 * or reset the PHY on its own. All control of the PHY must be done by the user.
00081 *
00082 * **Asynchronous Callbacks**
00083 *
00084 * The driver services interrupts and passes Ethernet frames to the higher layer
00085 * software through asynchronous callback functions. When using the driver
00086 * directly (i.e., not with the RTOS protocol stack), the higher layer
00087 * software must register its callback functions during initialization. The
00088 * driver requires callback functions for received frames, for confirmation of
00089 * transmitted frames, and for asynchronous errors.
00090 *
00091 * **Interrupts**
00092 *
00093 * The driver has no dependencies on the interrupt controller. The driver
00094 * provides two interrupt handlers. XEmac_IntrHandlerDma() handles interrupts

00095 * when the EMAC is configured with scatter-gather DMA. XEmac_IntrHandlerFifo()
00096 * handles interrupts when the EMAC is configured for direct FIFO I/O or simple
00097 * DMA. Either of these routines can be connected to the system interrupt
00098 * controller by the user.
00099 *
00100 * **Interrupt Frequency**
00101 *
00102 * When the EMAC is configured with scatter-gather DMA, the frequency of
00103 * interrupts can be controlled with the interrupt coalescing features of the
00104 * scatter-gather DMA engine. The frequency of interrupts can be adjusted using
00105 * the driver API functions for setting the packet count threshold and the
packet
00106 * wait bound values.
00107 *
00108 * The scatter-gather DMA engine only interrupts when the packet count threshold
00109 * is reached, instead of interrupting for each packet. A packet is a generic
00110 * term used by the scatter-gather DMA engine, and is equivalent to an Ethernet
00111 * frame in our case.
00112 *
00113 * The packet wait bound is a timer value used during interrupt coalescing to
00114 * trigger an interrupt when not enough packets have been received to reach the
00115 * packet count threshold.
00116 *
00117 * These values can be tuned by the user to meet their needs. If there appear to
00118 * be interrupt latency problems or delays in packet arrival that are longer
than
00119 * might be expected, the user should verify that the packet count threshold is
00120 * set low enough to receive interrupts before the wait bound timer goes off.
00121 *
00122 * **Device Reset**
00123 *
00124 * Some errors that can occur in the device require a device reset. These errors
00125 * are listed in the XEmac_SetErrorHandler() function header. The user's error
00126 * handler is responsible for resetting the device and re-configuring it based
on
00127 * its needs (the driver does not save the current configuration). When
00128 * integrating into an RTOS, these reset and re-configure obligations are
00129 * taken care of by the Xilinx adapter software if it exists for that RTOS.
00130 *
00131 * **Device Configuration**
00132 *
00133 * The device can be configured in various ways during the FPGA implementation
00134 * process. Configuration parameters are stored in the xemac_g.c files.
00135 * A table is defined where each entry contains configuration information
00136 * for an EMAC device. This information includes such things as the base
address
00137 * of the memory-mapped device, the base addresses of IPIF, DMA, and FIFO
modules
00138 * within the device, and whether the device has DMA, counter registers,
00139 * multicast support, MII support, and flow control.
00140 *
00141 * The driver tries to use the features built into the device. So if, for

00142 * example, the hardware is configured with scatter-gather DMA, the driver
00143 * expects to start the scatter-gather channels and expects that the user has
set
00144 * up the buffer descriptor lists already. If the user expects to use the driver
00145 * in a mode different than how the hardware is configured, the user should
00146 * modify the configuration table to reflect the mode to be used. Modifying the
00147 * configuration table is a workaround for now until we get some experience with
00148 * how users are intending to use the hardware in its different configurations.
00149 * For example, if the hardware is built with scatter-gather DMA but the user is
00150 * intending to use only simple DMA, the user either needs to modify the config
00151 * table as a workaround or rebuild the hardware with only simple DMA. The
00152 * recommendation at this point is to build the hardware with the features you
00153 * intend to use. If you're inclined to modify the table, do so before the call
00154 * to XEmac_Initialize(). Here is a snippet of code that changes a device to
00155 * simple DMA (the hardware needs to have DMA for this to work of course):
00156 * <pre>
00157 * XEmac_Config *ConfigPtr;
00158 *
00159 * ConfigPtr = XEmac_LookupConfig(DeviceId);
00160 * ConfigPtr->IpIfDmaConfig = XEM_CFG_SIMPLE_DMA;
00161 * </pre>
00162 *
00163 * Simple DMA
00164 *
00165 * Simple DMA is supported through the FIFO functions, FifoSend and FifoRecv, of
00166 * the driver (i.e., there is no separate interface for it). The driver makes
use
00167 * of the DMA engine for a simple DMA transfer if the device is configured with
00168 * DMA, otherwise it uses the FIFOs directly. While the simple DMA interface is
00169 * therefore transparent to the user, the caching of network buffers is not.
00170 * If the device is configured with DMA and the FIFO interface is used, the user
00171 * must ensure that the network buffers are not cached or are cache coherent,
00172 * since DMA will be used to transfer to and from the Emac device. If the device
00173 * is configured with DMA and the user really wants to use the FIFOs directly,
00174 * the user should rebuild the hardware without DMA. If unable to do this, there
00175 * is a workaround (described above in Device Configuration) to modify the
00176 * configuration table of the driver to fake the driver into thinking the device
00177 * has no DMA. A code snippet follows:
00178 * <pre>
00179 * XEmac_Config *ConfigPtr;
00180 *
00181 * ConfigPtr = XEmac_LookupConfig(DeviceId);
00182 * ConfigPtr->IpIfDmaConfig = XEM_CFG_NO_DMA;
00183 * </pre>
00184 *
00185 * Asserts
00186 *
00187 * Asserts are used within all Xilinx drivers to enforce constraints on argument
00188 * values. Asserts can be turned off on a system-wide basis by defining, at
00189 * compile time, the NDEBUB identifier. By default, asserts are turned on and it
00190 * is recommended that users leave asserts on during development.
00191 *
00192 * Building the driver

```

00193 *
00194 * The XEmac driver is composed of several source files. Why so many? This
00195 * allows the user to build and link only those parts of the driver that are
00196 * necessary. Since the EMAC hardware can be configured in various ways (e.g.,
00197 * with or without DMA), the driver too can be built with varying features.
00198 * For the most part, this means that besides always linking in xemac.c, you
00199 * link in only the driver functionality you want. Some of the choices you have
00200 * are polled vs. interrupt, interrupt with FIFOs only vs. interrupt with DMA,
00201 * self-test diagnostics, and driver statistics. Note that currently the DMA
code
00202 * must be linked in, even if you don't have DMA in the device.
00203 *
00204 * @note
00205 *
00206 * Xilinx drivers are typically composed of two components, one is the driver
00207 * and the other is the adapter. The driver is independent of OS and processor
00208 * and is intended to be highly portable. The adapter is OS-specific and
00209 * facilitates communication between the driver and an OS.
00210 * <br><br>
00211 * This driver is intended to be RTOS and processor independent. It works
00212 * with physical addresses only. Any needs for dynamic memory management,
00213 * threads or thread mutual exclusion, virtual memory, or cache control must
00214 * be satisfied by the layer above this driver.
00215 *
00216 * <pre>
00217 * MODIFICATION HISTORY:
00218 *
00219 * Ver    Who    Date      Changes
00220 * -----
00221 * 1.00a rpm  07/31/01 First release
00222 * 1.00b rpm  02/20/02 Repartitioned files and functions
00223 * 1.00b rpm  10/08/02 Replaced HasSgDma boolean with IpifDmaConfig enumerated
00224 *          configuration parameter
00225 * 1.00c rpm  12/05/02 New version includes support for simple DMA and the delay
00226 *          argument to SgSend
00227 * 1.00c rpm  02/03/03 The XST_DMA_SG_COUNT_EXCEEDED return code was removed
00228 *          from SetPktThreshold in the internal DMA driver. Also
00229 *          avoided compiler warnings by initializing Result in the
00230 *          DMA interrupt service routines.
00231 * </pre>
00232 *
00233 * *****/
00234
00235 #ifndef XEMAC_H /* prevent circular inclusions */
00236 #define XEMAC_H /* by using protection macros */
00237
00238 /***** Include Files *****/
00239
00240 #include "xbasic_types.h"
00241 #include "xstatus.h"
00242 #include "xparameters.h"
00243 #include "xpacket_fifo_v1_00_b.h" /* Uses v1.00b of Packet Fifo */

```

```

00244 #include "xdma_channel.h"
00245
00246 /***** Constant Definitions *****/
00247
00248 /*
00249  * Device information
00250  */
00251 #define XEM_DEVICE_NAME      "xemac"
00252 #define XEM_DEVICE_DESC     "Xilinx Ethernet 10/100 MAC"
00253
00254 /** @name Configuration options
00255  *
00256  * Device configuration options (see the XEmac_SetOptions() and
00257  * XEmac_GetOptions() for information on how to use these options)
00258  * @{
00259  */
00260 /**
00261  * <pre>
00262  *   XEM_BROADCAST_OPTION      Broadcast addressing on or off (default is on)
00263  *   XEM_UNICAST_OPTION        Unicast addressing on or off (default is on)
00264  *   XEM_PROMISC_OPTION        Promiscuous addressing on or off (default is
off)
00265  *   XEM_FDUPLEX_OPTION        Full duplex on or off (default is off)
00266  *   XEM_POLLED_OPTION          Polled mode on or off (default is off)
00267  *   XEM_LOOPBACK_OPTION        Internal loopback on or off (default is off)
00268  *   XEM_FLOW_CONTROL_OPTION    Interpret pause frames in full duplex mode
00269  *                               (default is off)
00270  *   XEM_INSERT_PAD_OPTION      Pad short frames on transmit (default is on)
00271  *   XEM_INSERT_FCS_OPTION      Insert FCS (CRC) on transmit (default is on)
00272  *   XEM_INSERT_ADDR_OPTION     Insert source address on transmit (default is
on)
00273  *   XEM_OVWRT_ADDR_OPTION      Overwrite source address on transmit. This is
00274  *                               only used if source address insertion is on.
00275  *                               (default is on)
00276  *   XEM_STRIP_PAD_FCS_OPTION    Strip FCS and padding from received frames
00277  *                               (default is off)
00278  * </pre>
00279  */
00280 #define XEM_UNICAST_OPTION    0x00000001UL
00281 #define XEM_BROADCAST_OPTION  0x00000002UL
00282 #define XEM_PROMISC_OPTION    0x00000004UL
00283 #define XEM_FDUPLEX_OPTION    0x00000008UL
00284 #define XEM_POLLED_OPTION     0x00000010UL
00285 #define XEM_LOOPBACK_OPTION   0x00000020UL
00286 #define XEM_FLOW_CONTROL_OPTION 0x00000080UL
00287 #define XEM_INSERT_PAD_OPTION  0x00000100UL
00288 #define XEM_INSERT_FCS_OPTION  0x00000200UL
00289 #define XEM_INSERT_ADDR_OPTION 0x00000400UL
00290 #define XEM_OVWRT_ADDR_OPTION  0x00000800UL
00291 #define XEM_STRIP_PAD_FCS_OPTION 0x00002000UL
00292 /* @} */

```



```

00293 /*
00294  * Not supported yet:
00295  *   XEM_MULTICAST_OPTION           Multicast addressing on or off (default is
off)
00296  */
00297 /* NOT SUPPORTED YET... */
00298 #define XEM_MULTICAST_OPTION       0x00000040UL
00299
00300 /*
00301  * Some default values for interrupt coalescing within the scatter-gather
00302  * DMA engine.
00303  */
00304 #define XEM_SGDMA_DFT_THRESHOLD     1           /* Default pkt threshold */
00305 #define XEM_SGDMA_MAX_THRESHOLD     255        /* Maximum pkt threshold */
00306 #define XEM_SGDMA_DFT_WAITBOUND     5          /* Default pkt wait bound (msec) */
00307 #define XEM_SGDMA_MAX_WAITBOUND     1023       /* Maximum pkt wait bound (msec) */
00308
00309 /*
00310  * Direction identifiers. These are used for setting values like packet
00311  * thresholds and wait bound for specific channels
00312  */
00313 #define XEM_SEND      1
00314 #define XEM_RECV     2
00315
00316 /*
00317  * Arguments to SgSend function to indicate whether to hold off starting
00318  * the scatter-gather engine.
00319  */
00320 #define XEM_SGDMA_NODELAY     0           /* start SG DMA immediately */
00321 #define XEM_SGDMA_DELAY      1           /* do not start SG DMA */
00322
00323 /*
00324  * Constants to determine the configuration of the hardware device. They are
00325  * used to allow the driver to verify it can operate with the hardware.
00326  */
00327 #define XEM_CFG_NO_IPIF      0           /* Not supported by the driver */
00328 #define XEM_CFG_NO_DMA      1           /* No DMA */
00329 #define XEM_CFG_SIMPLE_DMA   2           /* Simple DMA */
00330 #define XEM_CFG_DMA_SG      3           /* DMA scatter gather */
00331
00332 /*
00333  * The next few constants help upper layers determine the size of memory
00334  * pools used for Ethernet buffers and descriptor lists.
00335  */
00336 #define XEM_MAC_ADDR_SIZE    6           /* six-byte MAC address */
00337 #define XEM_MTU              1500       /* max size of Ethernet frame */
00338 #define XEM_HDR_SIZE         14          /* size of Ethernet header */
00339 #define XEM_HDR_VLAN_SIZE    18          /* size of Ethernet header with VLAN */
00340 #define XEM_TRL_SIZE         4           /* size of Ethernet trailer (FCS) */
00341 #define XEM_MAX_FRAME_SIZE   (XEM_MTU + XEM_HDR_SIZE + XEM_TRL_SIZE)
00342 #define XEM_MAX_VLAN_FRAME_SIZE (XEM_MTU + XEM_HDR_VLAN_SIZE + XEM_TRL_SIZE)
00343
00344 /*

```



```

00395     Xuint32 RecvShortErrors;           /**< Number of short frames discarded */
00396     Xuint32 RecvLongErrors;            /**< Number of long frames discarded */
00397     Xuint32 DmaErrors;                  /**< Number of DMA errors since init */
00398     Xuint32 FifoErrors;                 /**< Number of FIFO errors since init */
00399     Xuint32 RecvInterrupts;             /**< Number of receive interrupts */
00400     Xuint32 XmitInterrupts;             /**< Number of transmit interrupts */
00401     Xuint32 EmacInterrupts;            /**< Number of MAC (device) interrupts */
00402     Xuint32 TotalIntrs;                 /**< Total interrupts */
00403 } XEmac_Stats;
00404
00405 /**
00406  * This typedef contains configuration information for a device.
00407  */
00408 typedef struct
00409 {
00410     Xuint16 DeviceId;                   /**< Unique ID of device */
00411     Xuint32 BaseAddress;                 /**< Register base address */
00412     Xboolean HasCounters;                /**< Does device have counters? */
00413     Xuint8 IpIfDmaConfig;                /**< IPIF/DMA hardware configuration */
00414     Xboolean HasMii;                     /**< Does device support MII? */
00415
00416 } XEmac_Config;
00417
00418
00419 /** @name Typedefs for callbacks
00420  * Callback functions.
00421  * @{
00422  */
00423 /**
00424  * Callback when data is sent or received with scatter-gather DMA.
00425  *
00426  * @param CallbackRef is a callback reference passed in by the upper layer
00427  *         when setting the callback functions, and passed back to the upper
00428  *         layer when the callback is invoked.
00429  * @param BdPtr is a pointer to the first buffer descriptor in a list of
00430  *         buffer descriptors.
00431  * @param NumBds is the number of buffer descriptors in the list pointed
00432  *         to by BdPtr.
00433  */
00434 typedef void (*XEmac_SgHandler)(void *CallbackRef, XBufDescriptor *BdPtr,
00435                                 Xuint32 NumBds);
00436
00437 /**
00438  * Callback when data is sent or received with direct FIFO communication or
00439  * simple DMA. The user typically defines two callacks, one for send and one
00440  * for receive.
00441  *
00442  * @param CallbackRef is a callback reference passed in by the upper layer
00443  *         when setting the callback functions, and passed back to the upper
00444  *         layer when the callback is invoked.

```

```

00445 */
00446 typedef void (*XEmac_FifoHandler)(void *CallBackRef);
00447
00448 /**
00449  * Callback when an asynchronous error occurs.
00450  *
00451  * @param CallBackRef is a callback reference passed in by the upper layer
00452  *         when setting the callback functions, and passed back to the upper
00453  *         layer when the callback is invoked.
00454  * @param ErrorCode is a Xilinx error code defined in xstatus.h. Also see
00455  *         XEmac_SetErrorHandler() for a description of possible errors.
00456  */
00457 typedef void (*XEmac_ErrorHandler)(void *CallBackRef, XStatus ErrorCode);
00458 /*@}*/
00459
00460 /**
00461  * The XEmac driver instance data. The user is required to allocate a
00462  * variable of this type for every EMAC device in the system. A pointer
00463  * to a variable of this type is then passed to the driver API functions.
00464  */
00465 typedef struct
00466 {
00467     Xuint32 BaseAddress;           /* Base address (of IPIF) */
00468     Xuint32 IsStarted;            /* Device is currently started */
00469     Xuint32 IsReady;             /* Device is initialized and ready */
00470     Xboolean IsPolled;           /* Device is in polled mode */
00471     Xuint8 IpIfDmaConfig;        /* IPIF/DMA hardware configuration */
00472     Xboolean HasMii;             /* Does device support MII? */
00473     Xboolean HasMulticastHash;   /* Does device support multicast hash table?
00474 */
00474
00475     XEmac_Stats Stats;
00476     XPacketFifoV100b RecvFifo;   /* FIFO used to receive frames */
00477     XPacketFifoV100b SendFifo;   /* FIFO used to send frames */
00478
00479     /*
00480      * Callbacks
00481      */
00482     XEmac_FifoHandler FifoRecvHandler; /* for non-DMA/simple DMA interrupts */
00483     void *FifoRecvRef;
00484     XEmac_FifoHandler FifoSendHandler; /* for non-DMA/simple DMA interrupts */
00485     void *FifoSendRef;
00486     XEmac_ErrorHandler ErrorHandler; /* for asynchronous errors */
00487     void *ErrorRef;
00488
00489     XDmaChannel RecvChannel;      /* DMA receive channel driver */
00490     XDmaChannel SendChannel;     /* DMA send channel driver */
00491
00492     XEmac_SgHandler SgRecvHandler; /* callback for scatter-gather DMA */
00493     void *SgRecvRef;
00494     XEmac_SgHandler SgSendHandler; /* callback for scatter-gather DMA */

```

```
00495     void *SgSendRef;
00496 } XEmac;
00497
00498
00499 /***** Macros (Inline Functions) Definitions *****/
00500
00501 /*****
00502 /**
00503 *
00504 * This macro determines if the device is currently configured for
00505 * scatter-gather DMA.
00506 *
00507 * @param InstancePtr is a pointer to the XEmac instance to be worked on.
00508 *
00509 * @return
00510 *
00511 * Boolean XTRUE if the device is configured for scatter-gather DMA, or XFALSE
00512 * if it is not.
00513 *
00514 * @note
00515 *
00516 * Signature: Xboolean XEmac_mIsSgDma(XEmac *InstancePtr)
00517 *
00518 *****/
00519 #define XEmac_mIsSgDma(InstancePtr) \
00520     ((InstancePtr)->IpIfDmaConfig == XEM_CFG_DMA_SG)
00521
00522
00523 /*****
00524 /**
00525 *
00526 * This macro determines if the device is currently configured for simple DMA.
00527 *
00528 * @param InstancePtr is a pointer to the XEmac instance to be worked on.
00529 *
00530 * @return
00531 *
00532 * Boolean XTRUE if the device is configured for simple DMA, or XFALSE otherwise
00533 *
00534 * @note
00535 *
00536 * Signature: Xboolean XEmac_mIsSimpleDma(XEmac *InstancePtr)
00537 *
00538 *****/
00539 #define XEmac_mIsSimpleDma(InstancePtr) \
00540     ((InstancePtr)->IpIfDmaConfig == XEM_CFG_SIMPLE_DMA)
00541
00542
00543 /*****
00544 /**
00545 *
00546 * This macro determines if the device is currently configured with DMA (either
```

```

00547 * simple DMA or scatter-gather DMA)
00548 *
00549 * @param InstancePtr is a pointer to the XEmac instance to be worked on.
00550 *
00551 * @return
00552 *
00553 * Boolean XTRUE if the device is configured with DMA, or XFALSE otherwise
00554 *
00555 * @note
00556 *
00557 * Signature: Xboolean XEmac_mIsDma(XEmac *InstancePtr)
00558 *
00559 *****/
00560 #define XEmac_mIsDma(InstancePtr) \
00561     (XEmac_mIsSimpleDma(InstancePtr) || XEmac_mIsSgDma(InstancePtr))
00562
00563 /***** Function Prototypes *****/
00564
00565 /*
00566  * Initialization functions in xemac.c
00567  */
00568 XStatus XEmac_Initialize(XEmac *InstancePtr, Xuint16 DeviceId);
00569 XStatus XEmac_Start(XEmac *InstancePtr);
00570 XStatus XEmac_Stop(XEmac *InstancePtr);
00571 void XEmac_Reset(XEmac *InstancePtr);
00572 XEmac_Config *XEmac_LookupConfig(Xuint16 DeviceId);
00573
00574 /*
00575  * Diagnostic functions in xemac_selftest.c
00576  */
00577 XStatus XEmac_SelfTest(XEmac *InstancePtr);
00578
00579 /*
00580  * Polled functions in xemac_polled.c
00581  */
00582 XStatus XEmac_PollSend(XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 ByteCount);
00583 XStatus XEmac_PollRecv(XEmac *InstancePtr, Xuint8 *BufPtr,
00584                        Xuint32 *ByteCountPtr);
00585
00586 /*
00587  * Interrupts with scatter-gather DMA functions in xemac_intr_dma.c
00588  */
00589 XStatus XEmac_SgSend(XEmac *InstancePtr, XBufDescriptor *BdPtr, int Delay);
00590 XStatus XEmac_SgRecv(XEmac *InstancePtr, XBufDescriptor *BdPtr);
00591 XStatus XEmac_SetPktThreshold(XEmac *InstancePtr, Xuint32 Direction,
00592                               Xuint8 Threshold);
00593 XStatus XEmac_GetPktThreshold(XEmac *InstancePtr, Xuint32 Direction,
00594                               Xuint8 *ThreshPtr);
00595 XStatus XEmac_SetPktWaitBound(XEmac *InstancePtr, Xuint32 Direction,
00596                               Xuint32 TimerValue);

```

```

00597 XStatus XEmac_GetPktWaitBound(XEmac *InstancePtr, Xuint32 Direction,
00598                               Xuint32 *WaitPtr);
00599 XStatus XEmac_SetSgRecvSpace(XEmac *InstancePtr, Xuint32 *MemoryPtr,
00600                               Xuint32 ByteCount);
00601 XStatus XEmac_SetSgSendSpace(XEmac *InstancePtr, Xuint32 *MemoryPtr,
00602                               Xuint32 ByteCount);
00603 void XEmac_SetSgRecvHandler(XEmac *InstancePtr, void *CallBackRef,
00604                               XEmac_SgHandler FuncPtr);
00605 void XEmac_SetSgSendHandler(XEmac *InstancePtr, void *CallBackRef,
00606                               XEmac_SgHandler FuncPtr);
00607
00608 void XEmac_IntrHandlerDma(void *InstancePtr);          /* interrupt handler */
00609
00610 /*
00611  * Interrupts with direct FIFO functions in xemac_intr_fifo.c. Also used
00612  * for simple DMA.
00613  */
00614 XStatus XEmac_FifoSend(XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 ByteCount);
00615 XStatus XEmac_FifoRecv(XEmac *InstancePtr, Xuint8 *BufPtr,
00616                               Xuint32 *ByteCountPtr);
00617 void XEmac_SetFifoRecvHandler(XEmac *InstancePtr, void *CallBackRef,
00618                               XEmac_FifoHandler FuncPtr);
00619 void XEmac_SetFifoSendHandler(XEmac *InstancePtr, void *CallBackRef,
00620                               XEmac_FifoHandler FuncPtr);
00621
00622 void XEmac_IntrHandlerFifo(void *InstancePtr);        /* interrupt handler */
00623
00624 /*
00625  * General interrupt-related functions in xemac_intr.c
00626  */
00627 void XEmac_SetErrorHandler(XEmac *InstancePtr, void *CallBackRef,
00628                               XEmac_ErrorHandler FuncPtr);
00629
00630 /*
00631  * MAC configuration in xemac_options.c
00632  */
00633 XStatus XEmac_SetOptions(XEmac *InstancePtr, Xuint32 OptionFlag);
00634 Xuint32 XEmac_GetOptions(XEmac *InstancePtr);
00635 XStatus XEmac_SetMacAddress(XEmac *InstancePtr, Xuint8 *AddressPtr);
00636 void XEmac_GetMacAddress(XEmac *InstancePtr, Xuint8 *BufferPtr);
00637 XStatus XEmac_SetInterframeGap(XEmac *InstancePtr, Xuint8 Part1, Xuint8 Part2);
00638 void XEmac_GetInterframeGap(XEmac *InstancePtr, Xuint8 *Part1Ptr,
00639                               Xuint8* Part2Ptr);
00640
00641 /*
00642  * Multicast functions in xemac_multicast.c (not supported by EMAC yet)
00643  */
00644 XStatus XEmac_MulticastAdd(XEmac *InstancePtr, Xuint8 *AddressPtr);
00645 XStatus XEmac_MulticastClear(XEmac *InstancePtr);

```

```
00646
00647 /*
00648  * PHY configuration in xemac_phy.c
00649  */
00650 XStatus XEmac_PhyRead(XEmac *InstancePtr, Xuint32 PhyAddress,
00651                      Xuint32 RegisterNum, Xuint16 *PhyDataPtr);
00652 XStatus XEmac_PhyWrite(XEmac *InstancePtr, Xuint32 PhyAddress,
00653                       Xuint32 RegisterNum, Xuint16 PhyData);
00654
00655 /*
00656  * Statistics in xemac_stats.c
00657  */
00658 void XEmac_GetStats(XEmac *InstancePtr, XEmac_Stats *StatsPtr);
00659 void XEmac_ClearStats(XEmac *InstancePtr);
00660
00661
00662 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

XEmac_Stats Struct Reference

```
#include <xemac.h>
```

Detailed Description

Ethernet statistics (see [XEmac_GetStats\(\)](#) and [XEmac_ClearStats\(\)](#))

Data Fields

[Xuint32 XmitFrames](#)

[Xuint32 XmitBytes](#)

[Xuint32 XmitLateCollisionErrors](#)

[Xuint32 XmitExcessDeferral](#)

[Xuint32 XmitOverrunErrors](#)

[Xuint32 XmitUnderrunErrors](#)

[Xuint32 RecvFrames](#)

[Xuint32 RecvBytes](#)

[Xuint32 RecvFcsErrors](#)

[Xuint32 RecvAlignmentErrors](#)

[Xuint32 RecvOverrunErrors](#)

[Xuint32 RecvUnderrunErrors](#)

[Xuint32 RecvMissedFrameErrors](#)

[Xuint32 RecvCollisionErrors](#)

[Xuint32 RecvLengthFieldErrors](#)

[Xuint32 RecvShortErrors](#)

[Xuint32 RecvLongErrors](#)

[Xuint32 DmaErrors](#)

Xuint32 FifoErrors
Xuint32 RecvInterrupts
Xuint32 XmitInterrupts
Xuint32 EmacInterrupts
Xuint32 TotalIntrs

Field Documentation

Xuint32 XEmac_Stats::DmaErrors

Number of DMA errors since init

Xuint32 XEmac_Stats::EmacInterrupts

Number of MAC (device) interrupts

Xuint32 XEmac_Stats::FifoErrors

Number of FIFO errors since init

Xuint32 XEmac_Stats::RecvAlignmentErrors

Number of frames received with alignment errors

Xuint32 XEmac_Stats::RecvBytes

Number of bytes received

Xuint32 XEmac_Stats::RecvCollisionErrors

Number of frames discarded due to collisions

Xuint32 XEmac_Stats::RecvFcsErrors

Number of frames discarded due to FCS errors

Xuint32 XEmac_Stats::RecvFrames

Number of frames received

Xuint32 XEmac_Stats::RecvInterrupts

Number of receive interrupts

Xuint32 XEmac_Stats::RecvLengthFieldErrors

Number of frames discarded with invalid length field

Xuint32 XEmac_Stats::RecvLongErrors

Number of long frames discarded

Xuint32 XEmac_Stats::RecvMissedFrameErrors

Number of frames missed by MAC

Xuint32 XEmac_Stats::RecvOverrunErrors

Number of frames discarded due to overrun errors

Xuint32 XEmac_Stats::RecvShortErrors

Number of short frames discarded

Xuint32 XEmac_Stats::RecvUnderrunErrors

Number of recv underrun errors

Xuint32 XEmac_Stats::TotalIntrs

Total interrupts

Xuint32 XEmac_Stats::XmitBytes

Number of bytes transmitted

Xuint32 XEmac_Stats::XmitExcessDeferral

Number of transmission failures due to excess collision deferrals

Xuint32 XEmac_Stats::XmitFrames

Number of frames transmitted

Xuint32 XEmac_Stats::XmitInterrupts

Number of transmit interrupts

Xuint32 XEmac_Stats::XmitLateCollisionErrors

Number of transmission failures due to late collisions

Xuint32 XEmac_Stats::XmitOverrunErrors

Number of transmit overrun errors

Xuint32 XEmac_Stats::XmitUnderrunErrors

Number of transmit underrun errors

The documentation for this struct was generated from the following file:

- [emac/v1_00_c/src/xemac.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XEmac_Config Struct Reference

```
#include <xemac.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xboolean HasCounters

Xuint8 IpIfDmaConfig

Xboolean HasMii

Field Documentation

Xuint32 XEmac_Config::BaseAddress

Register base address

Xuint16 XEmac_Config::DeviceId

Unique ID of device

Xboolean XEmac_Config::HasCounters

Does device have counters?

Xboolean XEmac_Config::HasMii

Does device support MII?

Xuint8 XEmac_Config::IpIfDmaConfig

IPIF/DMA hardware configuration

The documentation for this struct was generated from the following file:

- [emac/v1_00_c/src/xemac.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XEmac Struct Reference

```
#include <xemac.h>
```

Detailed Description

The XEmac driver instance data. The user is required to allocate a variable of this type for every EMAC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `emac/v1_00_c/src/xemac.h`
-

emac/v1_00_c/src/xemac.c File Reference

Detailed Description

The [XEmac](#) driver. Functions in this file are the minimum required functions for this driver. See [xemac.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00b	rpm	07/23/02	Removed the PHY reset from Initialize()
1.00b	rmm	09/23/02	Removed commented code in Initialize(). Recycled as XEmac_mPhyReset macro in xemac_1.h .
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00c	rpm	12/12/02	Changed location of IsStarted assignment in XEmac_Start to be sure the flag is set before the device and interrupts are enabled.
1.00c	rpm	02/03/03	SelfTest was not clearing polled mode. Take driver out of polled mode in XEmac_Reset() to fix this problem.
1.00c	rmm	05/13/03	Fixed diab compiler warnings relating to asserts.

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

[XStatus XEmac_Initialize](#) ([XEmac](#) *InstancePtr, [Xuint16](#) DeviceId)

XStatus XEmac_Start (**XEmac** *InstancePtr)

XStatus XEmac_Stop (**XEmac** *InstancePtr)

void **XEmac_Reset** (**XEmac** *InstancePtr)

XStatus XEmac_SetMacAddress (**XEmac** *InstancePtr, **Xuint8** *AddressPtr)

void **XEmac_GetMacAddress** (**XEmac** *InstancePtr, **Xuint8** *BufferPtr)

XEmac_Config * **XEmac_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void XEmac_GetMacAddress( XEmac * InstancePtr,  
                        Xuint8 * BufferPtr  
                        )
```

Get the MAC address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufferPtr is an output parameter, and is a pointer to a buffer into which the current MAC address will be copied. The buffer must be at least 6 bytes.

Returns:

None.

Note:

None.

```
XStatus XEmac_Initialize( XEmac * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initialize a specific **XEmac** instance/driver. The initialization entails:

- Initialize fields of the **XEmac** structure
- Clear the Ethernet statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- If the device is configured with DMA, configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists may be passed to the driver.
- Reset the Ethernet MAC

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XEmac** instance. Passing in a device id associates the generic **XEmac** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

XEmac_Config* XEmac_LookupConfig(Xuint16 DeviceId)

Lookup the device configuration based on the unique device ID. The table EmacConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

void XEmac_Reset(XEmac * InstancePtr)

Reset the Ethernet MAC. This is a graceful reset in that the device is stopped first. Resets the DMA channels, the FIFOs, the transmitter, and the receiver. The PHY is not reset. Any frames in the scatter-gather descriptor lists will remain in the lists. The side effect of doing this is that after a reset and following a restart of the device, frames that were in the list before the reset may be transmitted or received. Reset must only be called after the driver has been initialized.

The driver is also taken out of polled mode if polled mode was set. The user is responsible for re-configuring the driver into polled mode after the reset if desired.

The configuration after this reset is as follows:

- Half duplex
- Disabled transmitter and receiver
- Enabled PHY (the PHY is not reset)
- MAC transmitter does pad insertion, FCS insertion, and source address overwrite.
- MAC receiver does not strip padding or FCS
- Interframe Gap as recommended by IEEE Std. 802.3 (96 bit times)
- Unicast addressing enabled
- Broadcast addressing enabled
- Multicast addressing disabled (addresses are preserved)
- Promiscuous addressing disabled
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- MAC address of all zeros
- Non-polled mode

The upper layer software is responsible for re-configuring (if necessary) and restarting the MAC after the reset. Note that the PHY is not reset. PHY control is left to the upper layer software. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus XEmac_SetMacAddress( XEmac * InstancePtr,  
                             Xuint8 * AddressPtr  
                             )
```

Set the MAC address for this driver/device. The address is a 48-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address.

Returns:

- XST_SUCCESS if the MAC address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus XEmac_Start( XEmac * InstancePtr)
```

Start the Ethernet controller as follows:

- If not in polled mode
 - Set the internal interrupt enable registers appropriately
 - Enable interrupts within the device itself. Note that connection of the driver's interrupt handler to the interrupt source (typically done using the interrupt controller component) is done by the higher layer software.
 - If the device is configured with scatter-gather DMA, start the DMA channels if the descriptor lists are not empty
- Enable the transmitter
- Enable the receiver

The PHY is enabled after driver initialization. We assume the upper layer software has configured it and the EMAC appropriately before this function is called.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not

yet been created for the send or receive channel.

Note:

The driver tries to match the hardware configuration. So if the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has set up the buffer descriptor lists already. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used. Modifying the config table is a workaround for now until we get some experience with how users are intending to use the hardware in its different configurations. For example, if the hardware is built with scatter-gather DMA but the user is intending to use only simple DMA, the user either needs to modify the config table as a workaround or rebuild the hardware with only simple DMA.

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus XEmac_Stop(XEmac * InstancePtr)

Stop the Ethernet MAC as follows:

- If the device is configured with scatter-gather DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode (the higher layer software is responsible for disabling interrupts at the interrupt controller)

The PHY is left enabled after a Stop is called.

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the Ethernet frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the

user is required to provide protection of this shared data (typically using a semaphore).

Generated on 29 May 2003 for Xilinx Device Drivers

emac/v1_00_c/src/xemac_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xemac.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	rpm	04/26/02	First release
1.00b	rmm	09/23/02	Added XEmac_mPhyReset macro
1.00c	rpm	12/05/02	New version includes support for simple DMA

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XEmac_mReadReg(BaseAddress, RegOffset)  
#define XEmac_mWriteReg(BaseAddress, RegOffset, Data)  
#define XEmac_mSetControlReg(BaseAddress, Mask)  
#define XEmac_mSetMacAddress(BaseAddress, AddressPtr)
```

```
#define XEmac_mEnable(BaseAddress)
#define XEmac_mDisable(BaseAddress)
#define XEmac_mIsTxDone(BaseAddress)
#define XEmac_mIsRxEmpty(BaseAddress)
#define XEmac_mPhyReset(BaseAddress)
```

Functions

```
void XEmac_SendFrame (Xuint32 BaseAddress, Xuint8 *FramePtr, int Size)
int XEmac_RecvFrame (Xuint32 BaseAddress, Xuint8 *FramePtr)
```

Define Documentation

```
#define XEmac_mDisable( BaseAddress )
```

Disable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XEmac_mEnable( BaseAddress )
```


Enable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XEmac_mIsRxEmpty( BaseAddress )
```

Check to see if the receive FIFO is empty.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is empty, or XFALSE if it is not.

Note:

None.

```
#define XEmac_mIsTxDone( BaseAddress )
```

Check to see if the transmission is complete.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is done, or XFALSE if it is not.

Note:

None.

```
#define XEmac_mPhyReset( BaseAddress )
```

Reset MII compliant PHY

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XEmac_mReadReg( BaseAddress,  
                          RegOffset )
```

Read the given register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the register offset to be read

Returns:

The 32-bit value of the register

Note:

None.

```
#define XEmac_mSetControlReg( BaseAddress,  
                          Mask )
```

Set the contents of the control register. Use the XEM_ECR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device

Mask is the 16-bit value to write to the control register

Returns:

None.

Note:

None.

```
#define XEmac_mSetMacAddress( BaseAddress,  
                             AddressPtr  )
```

Set the station address of the EMAC device.

Parameters:

BaseAddress is the base address of the device

AddressPtr is a pointer to a 6-byte MAC address

Returns:

None.

Note:

None.

```
#define XEmac_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data      )
```

Write the given register.

Parameters:

BaseAddress is the base address of the device
RegOffset is the register offset to be written
Data is the 32-bit value to write to the register

Returns:

None.

Note:

None.

Function Documentation

```
int XEmac_RecvFrame( Xuint32 BaseAddress,  
                    Xuint8 * FramePtr  
                    )
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device
FramePtr is a pointer to a word-aligned buffer where the frame will be stored.

Returns:

The size, in bytes, of the frame received.

Note:

None.

```
void XEmac_SendFrame( Xuint32 BaseAddress,  
                    Xuint8 * FramePtr,  
                    int Size  
                    )
```

Send an Ethernet frame. This size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device
FramePtr is a pointer to word-aligned frame
Size is the size, in bytes, of the frame

Returns:

None.

Note:

None.

emac/v1_00_c/src/xemac_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xemac_l.h,v 1.1 2002/12/09 17:42:43 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file emac/v1_00_c/src/xemac_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  High-level driver functions
00029 * are defined in xemac.h.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date        Changes
00035 * -----
00036 * 1.00b rpm  04/26/02  First release
00037 * 1.00b rmm  09/23/02  Added XEmac_mPhyReset macro
00038 * 1.00c rpm  12/05/02  New version includes support for simple DMA
00039 * </pre>
00040 *
00041 *****/
00042
```

```

00043 #ifndef XEMAC_L_H /* prevent circular inclusions */
00044 #define XEMAC_L_H /* by using protection macros */
00045
00046 /***** Include Files *****/
00047
00048 #include "xbasic_types.h"
00049 #include "xio.h"
00050
00051 /***** Constant Definitions *****/
00052
00053 /* Offset of the MAC registers from the IPIF base address */
00054 #define XEM_REG_OFFSET      0x1100UL
00055
00056 /*
00057  * Register offsets for the Ethernet MAC. Each register is 32 bits.
00058  */
00059 #define XEM_EMIR_OFFSET     (XEM_REG_OFFSET + 0x0)    /* EMAC Module ID */
00060 #define XEM_ECR_OFFSET     (XEM_REG_OFFSET + 0x4)    /* MAC Control */
00061 #define XEM_IFGP_OFFSET   (XEM_REG_OFFSET + 0x8)    /* Interframe Gap */
00062 #define XEM_SAH_OFFSET    (XEM_REG_OFFSET + 0xC)    /* Station addr, high */
00063 #define XEM_SAL_OFFSET    (XEM_REG_OFFSET + 0x10)   /* Station addr, low */
00064 #define XEM_MGTCR_OFFSET  (XEM_REG_OFFSET + 0x14)   /* MII mgmt control */
00065 #define XEM_MGTDNR_OFFSET (XEM_REG_OFFSET + 0x18)   /* MII mgmt data */
00066 #define XEM_RPLR_OFFSET   (XEM_REG_OFFSET + 0x1C)   /* Rx packet length */
00067 #define XEM_TPLR_OFFSET   (XEM_REG_OFFSET + 0x20)   /* Tx packet length */
00068 #define XEM_TSR_OFFSET    (XEM_REG_OFFSET + 0x24)   /* Tx status */
00069 #define XEM_RMFC_OFFSET   (XEM_REG_OFFSET + 0x28)   /* Rx missed frames */
00070 #define XEM_RCC_OFFSET    (XEM_REG_OFFSET + 0x2C)   /* Rx collisions */
00071 #define XEM_RFCSEC_OFFSET (XEM_REG_OFFSET + 0x30)   /* Rx FCS errors */
00072 #define XEM_RAEC_OFFSET   (XEM_REG_OFFSET + 0x34)   /* Rx alignment errors */
00073 #define XEM_TEDC_OFFSET   (XEM_REG_OFFSET + 0x38)   /* Transmit excess
00074                                     * deferral cnt */
00075
00076 /*
00077  * Register offsets for the IPIF components
00078  */
00079 #define XEM_ISR_OFFSET     0x20UL                    /* Interrupt status */
00080
00081 #define XEM_DMA_OFFSET     0x2300UL
00082 #define XEM_DMA_SEND_OFFSET (XEM_DMA_OFFSET + 0x0) /* DMA send channel */
00083 #define XEM_DMA_RECV_OFFSET (XEM_DMA_OFFSET + 0x40) /* DMA rcv channel */
00084
00085 #define XEM_PFIFO_OFFSET   0x2000UL
00086 #define XEM_PFIFO_TXREG_OFFSET (XEM_PFIFO_OFFSET + 0x0) /* Tx registers */
00087 #define XEM_PFIFO_RXREG_OFFSET (XEM_PFIFO_OFFSET + 0x10) /* Rx registers */
00088 #define XEM_PFIFO_TXDATA_OFFSET (XEM_PFIFO_OFFSET + 0x100) /* Tx keyhole */
00089 #define XEM_PFIFO_RXDATA_OFFSET (XEM_PFIFO_OFFSET + 0x200) /* Rx keyhole */
00090
00091 /*
00092  * EMAC Module Identification Register (EMIR)
00093  */
00094 #define XEM_EMIR_VERSION_MASK 0xFFFF0000UL          /* Device version */

```

```

00095 #define XEM_EMIR_TYPE_MASK          0x0000FF00UL          /* Device type */
00096
00097 /*
00098  * EMAC Control Register (ECR)
00099  */
00100 #define XEM_ECR_FULL_DUPLEX_MASK      0x80000000UL /* Full duplex mode */
00101 #define XEM_ECR_XMIT_RESET_MASK       0x40000000UL /* Reset transmitter */
00102 #define XEM_ECR_XMIT_ENABLE_MASK      0x20000000UL /* Enable transmitter */
00103 #define XEM_ECR_RECV_RESET_MASK       0x10000000UL /* Reset receiver */
00104 #define XEM_ECR_RECV_ENABLE_MASK      0x08000000UL /* Enable receiver */
00105 #define XEM_ECR_PHY_ENABLE_MASK       0x04000000UL /* Enable PHY */
00106 #define XEM_ECR_XMIT_PAD_ENABLE_MASK  0x02000000UL /* Enable xmit pad insert
*/
00107 #define XEM_ECR_XMIT_FCS_ENABLE_MASK  0x01000000UL /* Enable xmit FCS insert
*/
00108 #define XEM_ECR_XMIT_ADDR_INSERT_MASK 0x00800000UL /* Enable xmit source
addr
00109
*/ insertion */
00110 #define XEM_ECR_XMIT_ERROR_INSERT_MASK 0x00400000UL /* Insert xmit error */
00111 #define XEM_ECR_XMIT_ADDR_OVWRT_MASK  0x00200000UL /* Enable xmit source
addr
00112
*/ overwrite */
00113 #define XEM_ECR_LOOPBACK_MASK         0x00100000UL /* Enable internal
00114
*/ loopback */
00115 #define XEM_ECR_RECV_STRIP_ENABLE_MASK 0x00080000UL /* Enable recv pad/fcs
strip */
00116 #define XEM_ECR_UNICAST_ENABLE_MASK    0x00020000UL /* Enable unicast addr */
00117 #define XEM_ECR_MULTI_ENABLE_MASK     0x00010000UL /* Enable multicast addr
*/
00118 #define XEM_ECR_BROAD_ENABLE_MASK     0x00008000UL /* Enable broadcast addr
*/
00119 #define XEM_ECR_PROMISC_ENABLE_MASK    0x00004000UL /* Enable promiscuous
mode */
00120 #define XEM_ECR_RECV_ALL_MASK         0x00002000UL /* Receive all frames */
00121 #define XEM_ECR_RESERVED2_MASK        0x00001000UL /* Reserved */
00122 #define XEM_ECR_MULTI_HASH_ENABLE_MASK 0x00000800UL /* Enable multicast hash
*/
00123 #define XEM_ECR_PAUSE_FRAME_MASK      0x00000400UL /* Interpret pause frames
*/
00124 #define XEM_ECR_CLEAR_HASH_MASK        0x00000200UL /* Clear hash table */
00125 #define XEM_ECR_ADD_HASH_ADDR_MASK    0x00000100UL /* Add hash table address
*/
00126
00127 /*
00128  * Interframe Gap Register (IFGR)
00129  */
00130 #define XEM_IFGP_PART1_MASK           0xF8000000UL /* Interframe Gap Part1 */
00131 #define XEM_IFGP_PART1_SHIFT          27
00132 #define XEM_IFGP_PART2_MASK           0x07C00000UL /* Interframe Gap Part2 */
00133 #define XEM_IFGP_PART2_SHIFT          22
00134
00135 /*
00136  * Station Address High Register (SAH)

```



```

00137 */
00138 #define XEM_SAH_ADDR_MASK          0x0000FFFFUL /* Station address high bytes
*/
00139
00140 /*
00141 * Station Address Low Register (SAL)
00142 */
00143 #define XEM_SAL_ADDR_MASK          0xFFFFFFFFUL /* Station address low bytes
*/
00144
00145 /*
00146 * MII Management Control Register (MGTCR)
00147 */
00148 #define XEM_MGTCR_START_MASK       0x80000000UL /* Start/Busy */
00149 #define XEM_MGTCR_RW_NOT_MASK      0x40000000UL /* Read/Write Not (direction)
*/
00150 #define XEM_MGTCR_PHY_ADDR_MASK    0x3E000000UL /* PHY address */
00151 #define XEM_MGTCR_PHY_ADDR_SHIFT   25             /* PHY address shift */
00152 #define XEM_MGTCR_REG_ADDR_MASK    0x01F00000UL /* Register address */
00153 #define XEM_MGTCR_REG_ADDR_SHIFT   20             /* Register addr shift */
00154 #define XEM_MGTCR_MII_ENABLE_MASK  0x00080000UL /* Enable MII from EMAC */
00155 #define XEM_MGTCR_RD_ERROR_MASK    0x00040000UL /* MII mgmt read error */
00156
00157 /*
00158 * MII Management Data Register (MGTD R)
00159 */
00160 #define XEM_MGTD R_DATA_MASK       0x0000FFFFUL /* MII data */
00161
00162 /*
00163 * Receive Packet Length Register (RPLR)
00164 */
00165 #define XEM_RPLR_LENGTH_MASK       0x0000FFFFUL /* Receive packet length */
00166
00167 /*
00168 * Transmit Packet Length Register (TPLR)
00169 */
00170 #define XEM_TPLR_LENGTH_MASK       0x0000FFFFUL /* Transmit packet length */
00171
00172 /*
00173 * Transmit Status Register (TSR)
00174 */
00175 #define XEM_TSR_EXCESS_DEFERRAL_MASK 0x80000000UL /* Transmit excess deferral
*/
00176 #define XEM_TSR_FIFO_UNDERRUN_MASK 0x40000000UL /* Packet FIFO underrun */
00177 #define XEM_TSR_ATTEMPTS_MASK      0x3E000000UL /* Transmission attempts */
00178 #define XEM_TSR_LATE_COLLISION_MASK 0x01000000UL /* Transmit late collision */
00179
00180 /*
00181 * Receive Missed Frame Count (RMFC)
00182 */
00183 #define XEM_RMFC_DATA_MASK         0x0000FFFFUL
00184

```

```

00185 /*
00186  * Receive Collision Count (RCC)
00187  */
00188 #define XEM_RCC_DATA_MASK          0x0000FFFFUL
00189
00190 /*
00191  * Receive FCS Error Count (RFCSEC)
00192  */
00193 #define XEM_RFCSEC_DATA_MASK       0x0000FFFFUL
00194
00195 /*
00196  * Receive Alignment Error Count (RALN)
00197  */
00198 #define XEM_RAEC_DATA_MASK         0x0000FFFFUL
00199
00200 /*
00201  * Transmit Excess Deferral Count (TEDC)
00202  */
00203 #define XEM_TEDC_DATA_MASK         0x0000FFFFUL
00204
00205
00206 /*
00207  * EMAC Interrupt Registers (Status and Enable) masks. These registers are
00208  * part of the IPIF IP Interrupt registers
00209  */
00210 #define XEM_EIR_XMIT_DONE_MASK     0x00000001UL /* Xmit complete */
00211 #define XEM_EIR_RECV_DONE_MASK    0x00000002UL /* Recv complete */
00212 #define XEM_EIR_XMIT_ERROR_MASK   0x00000004UL /* Xmit error */
00213 #define XEM_EIR_RECV_ERROR_MASK   0x00000008UL /* Recv error */
00214 #define XEM_EIR_XMIT_SFIFO_EMPTY_MASK 0x00000010UL /* Xmit status fifo empty
*/
00215 #define XEM_EIR_RECV_LFIFO_EMPTY_MASK 0x00000020UL /* Recv length fifo empty
*/
00216 #define XEM_EIR_XMIT_LFIFO_FULL_MASK 0x00000040UL /* Xmit length fifo full */
00217 #define XEM_EIR_RECV_LFIFO_OVER_MASK 0x00000080UL /* Recv length fifo
00218  * overrun */
00219 #define XEM_EIR_RECV_LFIFO_UNDER_MASK 0x00000100UL /* Recv length fifo
00220  * underrun */
00221 #define XEM_EIR_XMIT_SFIFO_OVER_MASK 0x00000200UL /* Xmit status fifo
00222  * overrun */
00223 #define XEM_EIR_XMIT_SFIFO_UNDER_MASK 0x00000400UL /* Transmit status fifo
00224  * underrun */
00225 #define XEM_EIR_XMIT_LFIFO_OVER_MASK 0x00000800UL /* Transmit length fifo
00226  * overrun */
00227 #define XEM_EIR_XMIT_LFIFO_UNDER_MASK 0x00001000UL /* Transmit length fifo
00228  * underrun */
00229 #define XEM_EIR_XMIT_PAUSE_MASK    0x00002000UL /* Transmit pause pkt
00230  * received */
00231 #define XEM_EIR_RECV_DFIFO_OVER_MASK 0x00004000UL /* Receive data fifo
00232  * overrun */
00233 #define XEM_EIR_RECV_MISSED_FRAME_MASK 0x00008000UL /* Receive missed frame
00234  * error */
00235 #define XEM_EIR_RECV_COLLISION_MASK 0x00010000UL /* Receive collision

```

```

00236 * error */
00237 #define XEM_EIR_RECV_FCS_ERROR_MASK 0x00020000UL /* Receive FCS error */
00238 #define XEM_EIR_RECV_LEN_ERROR_MASK 0x00040000UL /* Receive length field
00239 * error */
00240 #define XEM_EIR_RECV_SHORT_ERROR_MASK 0x00080000UL /* Receive short frame
00241 * error */
00242 #define XEM_EIR_RECV_LONG_ERROR_MASK 0x00100000UL /* Receive long frame
00243 * error */
00244 #define XEM_EIR_RECV_ALIGN_ERROR_MASK 0x00200000UL /* Receive alignment
00245 * error */
00246
00247
00248 /***** Type Definitions *****/
00249
00250 /***** Macros (Inline Functions) Definitions *****/
00251
00252 /*****
00253 *
00254 * Low-level driver macros and functions. The list below provides signatures
00255 * to help the user use the macros.
00256 *
00257 * Xuint32 XEmac_mReadReg(Xuint32 BaseAddress, int RegOffset)
00258 * void XEmac_mWriteReg(Xuint32 BaseAddress, int RegOffset, Xuint32 Mask)
00259 *
00260 * void XEmac_mSetControlReg(Xuint32 BaseAddress, Xuint32 Mask)
00261 * void XEmac_mSetMacAddress(Xuint32 BaseAddress, Xuint8 *AddressPtr)
00262 *
00263 * void XEmac_mEnable(Xuint32 BaseAddress)
00264 * void XEmac_mDisable(Xuint32 BaseAddress)
00265 *
00266 * Xboolean XEmac_mIsTxDone(Xuint32 BaseAddress)
00267 * Xboolean XEmac_mIsRxEmpty(Xuint32 BaseAddress)
00268 *
00269 * void XEmac_SendFrame(Xuint32 BaseAddress, Xuint8 *FramePtr, int Size)
00270 * int XEmac_RecvFrame(Xuint32 BaseAddress, Xuint8 *FramePtr)
00271 *
00272 *****/
00273
00274 /*****
00275 /**
00276 *
00277 * Read the given register.
00278 *
00279 * @param BaseAddress is the base address of the device
00280 * @param RegOffset is the register offset to be read
00281 *
00282 * @return The 32-bit value of the register
00283 *
00284 * @note None.
00285 *
00286 *****/
00287 #define XEmac_mReadReg(BaseAddress, RegOffset) \

```

```

00288         XIo_In32((BaseAddress) + (RegOffset))
00289
00290
00291 /*****
00292 /**
00293 *
00294 * Write the given register.
00295 *
00296 * @param   BaseAddress is the base address of the device
00297 * @param   RegOffset is the register offset to be written
00298 * @param   Data is the 32-bit value to write to the register
00299 *
00300 * @return  None.
00301 *
00302 * @note   None.
00303 *
00304 *****/
00305 #define XEmac_mWriteReg(BaseAddress, RegOffset, Data) \
00306         XIo_Out32((BaseAddress) + (RegOffset), (Data))
00307
00308
00309 /*****
00310 /**
00311 *
00312 * Set the contents of the control register. Use the XEM_ECR_* constants
00313 * defined above to create the bit-mask to be written to the register.
00314 *
00315 * @param   BaseAddress is the base address of the device
00316 * @param   Mask is the 16-bit value to write to the control register
00317 *
00318 * @return  None.
00319 *
00320 * @note   None.
00321 *
00322 *****/
00323 #define XEmac_mSetControlReg(BaseAddress, Mask) \
00324         XIo_Out32((BaseAddress) + XEM_ECR_OFFSET, (Mask))
00325
00326
00327 /*****
00328 /**
00329 *
00330 * Set the station address of the EMAC device.
00331 *
00332 * @param   BaseAddress is the base address of the device
00333 * @param   AddressPtr is a pointer to a 6-byte MAC address
00334 *
00335 * @return  None.
00336 *
00337 * @note   None.
00338 *
00339 *****/

```

```

00340 #define XEmac_mSetMacAddress(BaseAddress, AddressPtr) \
00341 { \
00342     Xuint32 MacAddr; \
00343 \
00344     MacAddr = ((AddressPtr)[0] << 8) | (AddressPtr)[1]; \
00345     XIo_Out32((BaseAddress) + XEM_SAH_OFFSET, MacAddr); \
00346 \
00347     MacAddr = ((AddressPtr)[2] << 24) | ((AddressPtr)[3] << 16) | \
00348             ((AddressPtr)[4] << 8) | (AddressPtr)[5]; \
00349 \
00350     XIo_Out32((BaseAddress) + XEM_SAL_OFFSET, MacAddr); \
00351 }
00352
00353
00354 /*****
00355 **
00356 *
00357 * Enable the transmitter and receiver. Preserve the contents of the control
00358 * register.
00359 *
00360 * @param    BaseAddress is the base address of the device
00361 *
00362 * @return   None.
00363 *
00364 * @note    None.
00365 *
00366 *****/
00367 #define XEmac_mEnable(BaseAddress) \
00368 { \
00369     Xuint32 Control; \
00370     Control = XIo_In32((BaseAddress) + XEM_ECR_OFFSET); \
00371     Control &= ~(XEM_ECR_XMIT_RESET_MASK | XEM_ECR_RECV_RESET_MASK); \
00372     Control |= (XEM_ECR_XMIT_ENABLE_MASK | XEM_ECR_RECV_ENABLE_MASK); \
00373     XIo_Out32((BaseAddress) + XEM_ECR_OFFSET, Control); \
00374 }
00375
00376
00377 /*****
00378 **
00379 *
00380 * Disable the transmitter and receiver. Preserve the contents of the control
00381 * register.
00382 *
00383 * @param    BaseAddress is the base address of the device
00384 *
00385 * @return   None.
00386 *
00387 * @note    None.
00388 *
00389 *****/
00390 #define XEmac_mDisable(BaseAddress) \
00391     XIo_Out32((BaseAddress) + XEM_ECR_OFFSET, \

```

```

00392             XIo_In32((BaseAddress) + XEM_ECR_OFFSET) & \
00393             ~(XEM_ECR_XMIT_ENABLE_MASK | XEM_ECR_RECV_ENABLE_MASK))
00394
00395
00396 /*****
00397 /**
00398 *
00399 * Check to see if the transmission is complete.
00400 *
00401 * @param    BaseAddress is the base address of the device
00402 *
00403 * @return    XTRUE if it is done, or XFALSE if it is not.
00404 *
00405 * @note     None.
00406 *
00407 *****/
00408 #define XEmac_mIsTxDone(BaseAddress) \
00409         (XIo_In32((BaseAddress) + XEM_ISR_OFFSET) &
XEM_EIR_XMIT_DONE_MASK)
00410
00411
00412 /*****
00413 /**
00414 *
00415 * Check to see if the receive FIFO is empty.
00416 *
00417 * @param    BaseAddress is the base address of the device
00418 *
00419 * @return    XTRUE if it is empty, or XFALSE if it is not.
00420 *
00421 * @note     None.
00422 *
00423 *****/
00424 #define XEmac_mIsRxEmpty(BaseAddress) \
00425         (!(XIo_In32((BaseAddress) + XEM_ISR_OFFSET) &
XEM_EIR_RECV_DONE_MASK))
00426
00427
00428 /*****
00429 /**
00430 *
00431 * Reset MII compliant PHY
00432 *
00433 * @param    BaseAddress is the base address of the device
00434 *
00435 * @return    None.
00436 *
00437 * @note     None.
00438 *
00439 *****/
00440 #define XEmac_mPhyReset(BaseAddress) \
00441 { \

```

```
00442     Xuint32 Control; \
00443     Control = XIo_In32((BaseAddress) + XEM_ECR_OFFSET); \
00444     Control &= ~XEM_ECR_PHY_ENABLE_MASK; \
00445     XIo_Out32((BaseAddress) + XEM_ECR_OFFSET, Control); \
00446     Control |= XEM_ECR_PHY_ENABLE_MASK; \
00447     XIo_Out32((BaseAddress) + XEM_ECR_OFFSET, Control); \
00448 }
00449
00450
00451 /***** Function Prototypes *****/
00452
00453 void XEmac_SendFrame(Xuint32 BaseAddress, Xuint8 *FramePtr, int Size);
00454 int XEmac_RecvFrame(Xuint32 BaseAddress, Xuint8 *FramePtr);
00455
00456
00457 #endif /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a word-aligned buffer where the frame will be stored.

Returns:

The size, in bytes, of the frame received.

Note:

None.

```
void XEmac_SendFrame( Xuint32 BaseAddress,  
                    Xuint8 * FramePtr,  
                    int     Size  
                    )
```

Send an Ethernet frame. This size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to word-aligned frame

Size is the size, in bytes, of the frame

Returns:

None.

Note:

None.

emacs/v1_00_c/src/xemacs_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xemacs_i.h,v 1.1 2002/12/09 17:42:43 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file emacs/v1_00_c/src/xemacs_i.h
00026 *
00027 * This header file contains internal identifiers, which are those shared
00028 * between XEmac components.  The identifiers in this file are not intended for
00029 * use external to the driver.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date      Changes
00035 * -----
00036 * 1.00a rpm    07/31/01 First release
00037 * 1.00b rpm    02/20/02 Repartitioned files and functions
00038 * 1.00b rpm    04/29/02 Moved register definitions to xemacs_l.h
00039 * 1.00c rpm    12/05/02 New version includes support for simple DMA
00040 * </pre>
00041 *
00042 *****/
```

```

00043
00044 #ifndef XEMAC_I_H /* prevent circular inclusions */
00045 #define XEMAC_I_H /* by using protection macros */
00046
00047 /***** Include Files *****/
00048
00049 #include "xemac.h"
00050 #include "xemac_1.h"
00051
00052 /***** Constant Definitions *****/
00053
00054 /*
00055  * Default buffer descriptor control word masks. The default send BD control
00056  * is set for incrementing the source address by one for each byte transferred,
00057  * and specify that the destination address (FIFO) is local to the device. The
00058  * default receive BD control is set for incrementing the destination address
00059  * by one for each byte transferred, and specify that the source address is
00060  * local to the device.
00061  */
00062 #define XEM_DFT_SEND_BD_MASK      (XDC_DMCCR_SOURCE_INCR_MASK | \
00063                                   XDC_DMCCR_DEST_LOCAL_MASK)
00064 #define XEM_DFT_RECV_BD_MASK     (XDC_DMCCR_DEST_INCR_MASK | \
00065                                   XDC_DMCCR_SOURCE_LOCAL_MASK)
00066
00067 /*
00068  * Masks for the IPIF Device Interrupt enable and status registers.
00069  */
00070 #define XEM_IPIF_EMAC_MASK       0x00000004UL /* MAC interrupt */
00071 #define XEM_IPIF_SEND_DMA_MASK   0x00000008UL /* Send DMA interrupt */
00072 #define XEM_IPIF_RECV_DMA_MASK   0x00000010UL /* Receive DMA interrupt */
00073 #define XEM_IPIF_RECV_FIFO_MASK  0x00000020UL /* Receive FIFO interrupt */
00074 #define XEM_IPIF_SEND_FIFO_MASK  0x00000040UL /* Send FIFO interrupt */
00075
00076 /*
00077  * Default IPIF Device Interrupt mask when configured for DMA
00078  */
00079 #define XEM_IPIF_DMA_DFT_MASK    (XEM_IPIF_SEND_DMA_MASK | \
00080                                   XEM_IPIF_RECV_DMA_MASK | \
00081                                   XEM_IPIF_EMAC_MASK | \
00082                                   XEM_IPIF_SEND_FIFO_MASK | \
00083                                   XEM_IPIF_RECV_FIFO_MASK)
00084
00085 /*
00086  * Default IPIF Device Interrupt mask when configured without DMA
00087  */
00088 #define XEM_IPIF_FIFO_DFT_MASK   (XEM_IPIF_EMAC_MASK | \
00089                                   XEM_IPIF_SEND_FIFO_MASK | \
00090                                   XEM_IPIF_RECV_FIFO_MASK)
00091
00092 #define XEM_IPIF_DMA_DEV_INTR_COUNT 7 /* Number of interrupt sources */
00093 #define XEM_IPIF_FIFO_DEV_INTR_COUNT 5 /* Number of interrupt sources */
00094 #define XEM_IPIF_DEVICE_INTR_COUNT 7 /* Number of interrupt sources */

```



```

00148 XDC_IXR_PKT_WAIT_BOUND_MASK | \
00149 XDC_IXR_SG_END_MASK)
00150
00151
00152 /***** Type Definitions *****/
00153
00154 /***** Macros (Inline Functions) Definitions *****/
00155
00156
00157 /*****
00158 */
00159 *
00160 * Clears a structure of given size, in bytes, by setting each byte to 0.
00161 *
00162 * @param StructPtr is a pointer to the structure to be cleared.
00163 * @param NumBytes is the number of bytes in the structure.
00164 *
00165 * @return
00166 *
00167 * None.
00168 *
00169 * @note
00170 *
00171 * Signature: void XEmac_mClearStruct(Xuint8 *StructPtr, unsigned int NumBytes)
00172 *
00173 *****/
00174 #define XEmac_mClearStruct(StructPtr, NumBytes) \
00175 { \
00176     int i; \
00177     Xuint8 *BytePtr = (Xuint8 *) (StructPtr); \
00178     for (i=0; i < (unsigned int) (NumBytes); i++) \
00179     { \
00180         *BytePtr++ = 0; \
00181     } \
00182 }
00183
00184 /***** Variable Definitions *****/
00185
00186 extern XEmac_Config XEmac_ConfigTable[];
00187
00188 /***** Function Prototypes *****/
00189
00190 void XEmac_CheckEmacError(XEmac *InstancePtr, Xuint32 IntrStatus);
00191 void XEmac_CheckFifoRecvError(XEmac *InstancePtr);
00192 void XEmac_CheckFifoSendError(XEmac *InstancePtr);
00193
00194 #endif /* end of protection macro */

```

emac/v1_00_c/src/xemac_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between **XEmac** components. The identifiers in this file are not intended for use external to the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00b	rpm	04/29/02	Moved register definitions to xemac_1.h
1.00c	rpm	12/05/02	New version includes support for simple DMA

```
#include "xemac.h"  
#include "xemac\_1.h"
```

[Go to the source code of this file.](#)

Variables

[XEmac_Config](#) [XEmac_ConfigTable](#) []

Variable Documentation

[XEmac_Config](#) XEmac_ConfigTable[]()

This table contains configuration information for each EMAC device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

emac/v1_00_c/src/xemac_selftest.c File Reference

Detailed Description

Self-test and diagnostic functions of the **XEmac** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

XStatus **XEmac_SelfTest** (**XEmac** *InstancePtr)

Function Documentation

XStatus **XEmac_SelfTest**(**XEmac** * *InstancePtr*)

Performs a self-test on the Ethernet device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the Ethernet device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the [XEmac](#) instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_PFIPO_BAD_REG_VALUE	FIFO failed register self-test
XST_DMA_TRANSFER_ERROR	DMA failed data transfer self-test
XST_DMA_RESET_REGISTER_ERROR	DMA control register value was incorrect after a reset
XST_REGISTER_ERROR	Ethernet failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed
XST_IPIF_REG_WIDTH_ERROR	An invalid register width was passed into the function
XST_IPIF_RESET_REGISTER_ERROR	The value of a register at reset was invalid
XST_IPIF_DEVICE_STATUS_ERROR	A write to the device status register did not read back correctly
XST_IPIF_DEVICE_ACK_ERROR	A bit in the device status register did not reset when acked
XST_IPIF_DEVICE_ENABLE_ERROR	The device interrupt enable register was not updated correctly by the hardware when other registers were written to
XST_IPIF_IP_STATUS_ERROR	A write to the IP interrupt status register did not read back correctly
XST_IPIF_IP_ACK_ERROR	One or more bits in the IP status register did not reset when acked
XST_IPIF_IP_ENABLE_ERROR	The IP interrupt enable register was not updated correctly when other registers were written to

Note:

This function makes use of options-related functions, and the [XEmac_PollSend\(\)](#) and [XEmac_PollRecv\(\)](#) functions.

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread

to monitor the self-test thread.

Generated on 29 May 2003 for Xilinx Device Drivers

emac/v1_00_c/src/xemac_polled.c File Reference

Detailed Description

Contains functions used when the driver is in polled mode. Use the [XEmac_SetOptions\(\)](#) function to put the driver into polled mode.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

[XStatus XEmac_PollSend](#) ([XEmac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) ByteCount)

[XStatus XEmac_PollRecv](#) ([XEmac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) *ByteCountPtr)

Function Documentation

```
XStatus XEmac_PollRecv( XEmac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 * ByteCountPtr  
                        )
```

Receive an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the frame directly from the MAC's packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The MAC's error status is not checked, so statistics are not updated for polled receive. The buffer into which the frame will be received must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.

Note:

Input buffer must be big enough to hold the largest Ethernet frame. Buffer must also be 32-bit aligned.

```
XStatus XEmac_PollSend( XEmac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 ByteCount  
                        )
```

Send an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the frame directly to the MAC's packet FIFO, then enters a loop checking the device status for completion or error. Statistics are updated if an error occurs. The buffer to be sent must be word-aligned.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_FIFO_NO_ROOM if there is no room in the EMAC's length FIFO for this frame
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.
- XST_EMAC_COLLISION if the send failed due to excess deferral or late collision

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread. On a 10Mbps MAC, it takes about 1.21 msecs to transmit a maximum size Ethernet frame (1518 bytes). On a 100Mbps MAC, it takes about 121 usecs to transmit a maximum size Ethernet frame.

emac/v1_00_c/src/xemac_intr_dma.c File Reference

Detailed Description

Contains functions used in interrupt mode when configured with scatter-gather DMA.

The interrupt handler, [XEmac_IntrHandlerDma\(\)](#), must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA and the delay argument to SgSend
1.00c	rpm	02/03/03	The XST_DMA_SG_COUNT_EXCEEDED return code was removed from SetPktThreshold in the internal DMA driver. Also avoided compiler warnings by initializing Result in the interrupt service routines.
1.00c	rpm	03/26/03	Fixed a problem in the interrupt service routines where the interrupt status was toggled clear after a call to ErrorHandler, but if ErrorHandler reset the device the toggle actually asserted the interrupt because the reset had cleared it.

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"  
#include "xbuf_descriptor.h"  
#include "xdma_channel.h"  
#include "xipif_v1_23_b.h"
```

Functions

XStatus XEmac_SgSend (**XEmac** *InstancePtr, **XBufDescriptor** *BdPtr, int Delay)

XStatus XEmac_SgRecv (**XEmac** *InstancePtr, **XBufDescriptor** *BdPtr)

void **XEmac_IntrHandlerDma** (void *InstancePtr)

XStatus XEmac_SetPktThreshold (**XEmac** *InstancePtr, **Xuint32** Direction, **Xuint8** Threshold)

XStatus XEmac_GetPktThreshold (**XEmac** *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)

XStatus XEmac_SetPktWaitBound (**XEmac** *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)

XStatus XEmac_GetPktWaitBound (**XEmac** *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)

XStatus XEmac_SetSgRecvSpace (**XEmac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

XStatus XEmac_SetSgSendSpace (**XEmac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)

void **XEmac_SetSgRecvHandler** (**XEmac** *InstancePtr, void *CallBackRef, **XEmac_SgHandler** FuncPtr)

void **XEmac_SetSgSendHandler** (**XEmac** *InstancePtr, void *CallBackRef, **XEmac_SgHandler** FuncPtr)

Function Documentation

```
XStatus XEmac_GetPktThreshold(XEmac * InstancePtr,  
                             Xuint32 Direction,  
                             Xuint8 * ThreshPtr  
                             )
```

Get the value of the packet count threshold for this driver/device. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
XStatus XEmac_GetPktWaitBound( XEmac * InstancePtr,  
                                Xuint32 Direction,  
                                Xuint32 * WaitPtr  
                                )
```

Get the packet wait bound timer for this driver/device. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void XEmac_IntrHandlerDma( void * InstancePtr)
```

The interrupt handler for the Ethernet driver when configured with scatter- gather DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, Send Packet FIFO, Recv DMA channel, or Send DMA channel. The packet FIFOs only interrupt during "deadlock" conditions.

Parameters:

InstancePtr is a pointer to the **XEmac** instance that just interrupted.

Returns:

None.

Note:

None.


```
XStatus XEmac_SetPktThreshold( XEmac * InstancePtr,  
                                Xuint32 Direction,  
                                Xuint8 Threshold  
                                )
```

Set the packet count threshold for this device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

The packet threshold could be set to larger than the number of descriptors allocated to the DMA channel. In this case, the wait bound will take over and always indicate data arrival. There was a check in this function that returned an error if the threshold was larger than the number of descriptors, but that was removed because users would then have to set the threshold only after they set descriptor space, which is an order dependency that caused confusion.

```
XStatus XEmac_SetPktWaitBound( XEmac * InstancePtr,  
                                Xuint32 Direction,  
                                Xuint32 TimerValue  
                                )
```

Set the packet wait bound timer for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void XEmac_SetSgRecvHandler( XEmac *           InstancePtr,
                             void *           CallbackRef,
                             XEmac_SgHandler FuncPtr
                             )
```

Set the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame received. The head of a descriptor list is passed in along with the number of descriptors in the list. Before leaving the callback, the upper layer software should attach a new buffer to each descriptor in the list.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XEmac_SetSgRecvSpace( XEmac * InstancePtr,  
                             Xuint32 * MemoryPtr,  
                             Xuint32 ByteCount  
                             )
```

Give the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be word-aligned. An assert will occur if asserts are turned on and the memory is not word-aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XEmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
void XEmac_SetSgSendHandler( XEmac * InstancePtr,  
                             void * CallbackRef,  
                             XEmac_SgHandler FuncPtr  
                             )
```

Set the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame sent. The head of a descriptor list is passed in along with the number of descriptors in the list. The callback is responsible for freeing buffers attached to these descriptors.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XEmac_SetSgSendSpace( XEmac * InstancePtr,  
                               Xuint32 * MemoryPtr,  
                               Xuint32 ByteCount  
                               )
```

Give the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The **xemac.h** file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be word-aligned. An assert will occur if asserts are turned on and the memory is not word-aligned.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

MemoryPtr is a pointer to the word-aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XEmac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
XStatus XEmac_SgRecv( XEmac * InstancePtr,  
                    XBufDescriptor * BdPtr  
                    )
```

Add a descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be word-aligned. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. This function can be called when the device is started or stopped. Note that it does start the scatter-gather DMA engine. Although this is not necessary during initialization, it is not a problem during initialization because the MAC receiver is not yet started.

The buffer attached to the descriptor must be word-aligned on both the front end and the back end.

Notification of received frames are done asynchronously through the receive callback function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

```
XStatus XEmac_SgSend( XEmac * InstancePtr,  
                    XBufDescriptor * BdPtr,  
                    int Delay  
                    )
```

Send an Ethernet frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire Ethernet frame may or may not be contained within one descriptor. This function simply inserts the descriptor into the scatter-gather engine's transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be word-aligned.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

The buffer attached to the descriptor must be word-aligned on the front end.

This call is non-blocking. Notification of error or successful transmission is done asynchronously through the send or error callback function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Delay indicates whether to start the scatter-gather DMA channel immediately, or whether to wait. This allows the user to build up a list of more than one descriptor before starting the transmission of the packets, which allows the application to keep up with DMA and have a constant stream of frames being transmitted. Use `XEM_SGDMA_NODELAY` or `XEM_SGDMA_DELAY`, defined in **xemac.h**, as the value of this argument. If the user chooses to delay and build a list, the user must call this function with the `XEM_SGDMA_NODELAY` option or call **XEmac_Start()** to kick off the transmissions.

Returns:

- `XST_SUCCESS` if the buffer was successful sent
- `XST_DEVICE_IS_STOPPED` if the Ethernet MAC has not been started yet
- `XST_NOT_SGDMA` if the device is not in scatter-gather DMA mode
- `XST_DMA_SG_LIST_FULL` if the descriptor list for the DMA channel is full
- `XST_DMA_SG_BD_LOCKED` if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- `XST_DMA_SG_NOTHING_TO_COMMIT` if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

emac/v1_00_c/src/xemac_intr_fifo.c File Reference

Detailed Description

Contains functions related to interrupt mode using direct FIFO I/O or simple DMA. The driver uses simple DMA if the device is configured with DMA, otherwise it uses direct FIFO access.

The interrupt handler, [XEmac_IntrHandlerFifo\(\)](#), must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00c	rpm	04/01/03	Added check in FifoSend for room in the data FIFO before starting a simple DMA transfer.

```
#include "xbasic_types.h"
#include "xemac_i.h"
#include "xio.h"
#include "xipif_v1_23_b.h"
```

Functions


```
XStatus XEmac_FifoSend (XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 ByteCount)
XStatus XEmac_FifoRecv (XEmac *InstancePtr, Xuint8 *BufPtr, Xuint32 *ByteCountPtr)
void XEmac_IntrHandlerFifo (void *InstancePtr)
void XEmac_SetFifoRecvHandler (XEmac *InstancePtr, void *CallBackRef,
    XEmac_FifoHandler FuncPtr)
void XEmac_SetFifoSendHandler (XEmac *InstancePtr, void *CallBackRef,
    XEmac_FifoHandler FuncPtr)
```

Function Documentation

```
XStatus XEmac_FifoRecv( XEmac * InstancePtr,
                        Xuint8 * BufPtr,
                        Xuint32 * ByteCountPtr
                        )
```

Receive an Ethernet frame into the buffer passed as an argument. This function is called in response to the callback function for received frames being called by the driver. The callback function is set up using SetFifoRecvHandler, and is invoked when the driver receives an interrupt indicating a received frame. The driver expects the upper layer software to call this function, FifoRecv, to receive the frame. The buffer supplied should be large enough to hold a maximum-size Ethernet frame.

The buffer into which the frame will be received must be word-aligned.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from the Emac to memory. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xemac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode

- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

The input buffer must be big enough to hold the largest Ethernet frame.

```
XStatus XEmac_FifoSend( XEmac * InstancePtr,
                        Xuint8 * BufPtr,
                        Xuint32 ByteCount
                        )
```

Send an Ethernet frame using direct FIFO I/O or simple DMA with interrupts. The caller provides a contiguous-memory buffer and its length. The buffer must be word-aligned. The callback function set by using SetFifoSendHandler is invoked when the transmission is complete.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from memory to the Emac. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xemac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was successfully sent. An interrupt is generated when the EMAC transmits the frame and the driver calls the callback set with **XEmac_SetFifoSendHandler()**
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_FIFO_NO_ROOM if there is no room in the FIFO for this frame
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The

user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```
void XEmac_IntrHandlerFifo( void * InstancePtr)
```

The interrupt handler for the Ethernet driver when configured for direct FIFO communication or simple DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, or Send Packet FIFO. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the MAC.

Parameters:

InstancePtr is a pointer to the **XEmac** instance that just interrupted.

Returns:

None.

Note:

None.

```
void XEmac_SetFifoRecvHandler( XEmac * InstancePtr,  
                               void * CallBackRef,  
                               XEmac_FifoHandler FuncPtr  
                               )
```

Set the callback function for handling confirmation of transmitted frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called by the driver once per frame sent. The callback is responsible for freeing the transmitted buffer if necessary.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XEmac_SetFifoSendHandler( XEmac *           InstancePtr,  
                             void *           CallbackRef,  
                             XEmac_FifoHandler FuncPtr  
                             )
```

Set the callback function for handling received frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called once per frame received. During the callback, the upper layer software should call `FifoRecv` to retrieve the received frame.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

emac/v1_00_c/src/xemac_intr.c File Reference

Detailed Description

This file contains general interrupt-related functions of the [XEmac](#) driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA
1.00c	rpm	03/31/03	Added comment to indicate that no Receive Length FIFO overrun interrupts occur in v1.001 and later of the EMAC device. This avoids the need to reset the device on receive overruns.

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

```
void XEmac\_SetErrorHandler (XEmac *InstancePtr, void *CallBackRef, XEmac\_ErrorHandler FuncPtr)
```

Function Documentation

```
void XEmac_SetErrorHandler( XEmac *           InstancePtr,  
                           void *           CallbackRef,  
                           XEmac_ErrorHandler FuncPtr  
                           )
```

Set the callback function for handling asynchronous errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable MAC error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_DMA_SG_NO_LIST indicates an attempt was made to access a scatter-gather DMA list that has not yet been created.
- XST_DMA_SG_LIST_EMPTY indicates the driver tried to get a descriptor from the receive descriptor list, but the list was empty.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

emac/v1_00_c/src/xemac_options.c File Reference

Detailed Description

Functions in this file handle configuration of the **XEmac** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"
```

Data Structures

```
struct OptionMap
```

Functions

XStatus **XEmac_SetOptions** (**XEmac** *InstancePtr, **Xuint32** OptionsFlag)

Xuint32 XEmac_GetOptions (**XEmac** *InstancePtr)

XStatus XEmac_SetInterframeGap (**XEmac** *InstancePtr, **Xuint8** Part1, **Xuint8** Part2)

void **XEmac_GetInterframeGap** (**XEmac** *InstancePtr, **Xuint8** *Part1Ptr, **Xuint8** *Part2Ptr)

Function Documentation

```
void XEmac_GetInterframeGap( XEmac * InstancePtr,  
                             Xuint8 * Part1Ptr,  
                             Xuint8 * Part2Ptr  
                             )
```

Get the interframe gap, parts 1 and 2. See the description of interframe gap above in [XEmac_SetInterframeGap\(\)](#).

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Part1Ptr is a pointer to an 8-bit buffer into which the interframe gap part 1 value will be copied.

Part2Ptr is a pointer to an 8-bit buffer into which the interframe gap part 2 value will be copied.

Returns:

None. The values of the interframe gap parts are copied into the output parameters.

```
Xuint32 XEmac_GetOptions( XEmac * InstancePtr)
```

Get Ethernet driver/device options. The 32-bit value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

The 32-bit value of the Ethernet options. The value is a bit-mask representing all options that are currently enabled. See [xemac.h](#) for a description of the available options.

Note:

None.

```
XStatus XEmac_SetInterframeGap( XEmac * InstancePtr,  
                                Xuint8   Part1,  
                                Xuint8   Part2  
                                )
```

Set the Interframe Gap (IFG), which is the time the MAC delays between transmitting frames. There are two parts required. The total interframe gap is the total of the two parts. The values provided for the Part1 and Part2 parameters are multiplied by 4 to obtain the bit-time interval. The first part should be the first 2/3 of the total interframe gap. The MAC will reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part1. Part1 may be shorter than 2/3 the total and can be as small as zero. The second part should be the last 1/3 of the total interframe gap, but can be as large as the total interframe gap. The MAC will not reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part2.

The device must be stopped before setting the interframe gap.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Part1 is the interframe gap part 1 (which will be multiplied by 4 to get the bit-time interval).

Part2 is the interframe gap part 2 (which will be multiplied by 4 to get the bit-time interval).

Returns:

- XST_SUCCESS if the interframe gap was set successfully
- XST_DEVICE_IS_STARTED if the device has not been stopped

Note:

None.

```
XStatus XEmac_SetOptions( XEmac * InstancePtr,  
                          Xuint32 OptionsFlag  
                          )
```

Set Ethernet driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

OptionsFlag is a bit-mask representing the Ethernet options to turn on or off. See [xemac.h](#) for a description of the available options.

Returns:

- o XST_SUCCESS if the options were set successfully
- o XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

emac/v1_00_c/src/xemac_multicast.c File Reference

Detailed Description

Contains functions to configure multicast addressing in the Ethernet MAC.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"
```

Functions

XStatus XEmac_MulticastAdd (XEmac *InstancePtr, Xuint8 *AddressPtr)

XStatus XEmac_MulticastClear (XEmac *InstancePtr)

Function Documentation

```
XStatus XEmac_MulticastAdd( XEmac * InstancePtr,  
                             Xuint8 * AddressPtr  
                             )
```

Add a multicast address to the list of multicast addresses from which the EMAC accepts frames. The EMAC uses a hash table for multicast address filtering. Obviously, the more multicast addresses that are added reduces the accuracy of the address filtering. The upper layer software that receives multicast frames should perform additional filtering when accuracy must be guaranteed. There is no way to retrieve a multicast address or the multicast address list once added. The upper layer software should maintain its own list of multicast addresses. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

AddressPtr is a pointer to a 6-byte multicast address.

Returns:

- XST_SUCCESS if the multicast address was added successfully
- XST_NO_FEATURE if the device is not configured with multicast support
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

Not currently supported.

```
XStatus XEmac_MulticastClear( XEmac * InstancePtr)
```

Clear the hash table used by the EMAC for multicast address filtering. The entire hash table is cleared, meaning no multicast frames will be accepted after this function is called. If this function is used to delete one or more multicast addresses, the upper layer software is responsible for adding back those addresses still needed for address filtering. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

- XST_SUCCESS if the multicast address list was cleared
- XST_NO_FEATURE if the device is not configured with multicast support
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

Not currently supported.

Generated on 29 May 2003 for Xilinx Device Drivers

emac/v1_00_c/src/xemac_phy.c File Reference

Detailed Description

Contains functions to read and write the PHY through the Ethernet MAC MII registers. These assume an MII-compliant PHY.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA

```
#include "xbasic_types.h"  
#include "xemac_i.h"  
#include "xio.h"
```

Functions

XStatus XEmac_PhyRead (**XEmac** *InstancePtr, **Xuint32** PhyAddress, **Xuint32** RegisterNum, **Xuint16** *PhyDataPtr)

XStatus XEmac_PhyWrite (**XEmac** *InstancePtr, **Xuint32** PhyAddress, **Xuint32** RegisterNum, **Xuint16** PhyData)

Function Documentation

```
XStatus XEmac_PhyRead( XEmac * InstancePtr,  
                      Xuint32 PhyAddress,  
                      Xuint32 RegisterNum,  
                      Xuint16 * PhyDataPtr  
                      )
```

Read the current value of the PHY register indicated by the *PhyAddress* and the *RegisterNum* parameters. The MAC provides the driver with the ability to talk to a PHY that adheres to the Media Independent Interface (MII) as defined in the IEEE 802.3 standard.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

PhyAddress is the address of the PHY to be read (supports multiple PHYs)

RegisterNum is the register number, 0-31, of the specific PHY register to read

PhyDataPtr is an output parameter, and points to a 16-bit buffer into which the current value of the register will be copied.

Returns:

- o XST_SUCCESS if the PHY was read from successfully
- o XST_NO_FEATURE if the device is not configured with MII support
- o XST_EMAC_MII_BUSY if there is another PHY operation in progress
- o XST_EMAC_MII_READ_ERROR if a read error occurred between the MAC and the PHY

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that the read is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the *PhyRead* thread.


```
XStatus XEmac_PhyWrite( XEmac * InstancePtr,  
                        Xuint32 PhyAddress,  
                        Xuint32 RegisterNum,  
                        Xuint16 PhyData  
                        )
```

Write data to the specified PHY register. The Ethernet driver does not require the device to be stopped before writing to the PHY. Although it is probably a good idea to stop the device, it is the responsibility of the application to deem this necessary. The MAC provides the driver with the ability to talk to a PHY that adheres to the Media Independent Interface (MII) as defined in the IEEE 802.3 standard.

Parameters:

- InstancePtr* is a pointer to the **XEmac** instance to be worked on.
- PhyAddress* is the address of the PHY to be written (supports multiple PHYs)
- RegisterNum* is the register number, 0-31, of the specific PHY register to write
- PhyData* is the 16-bit value that will be written to the register

Returns:

- XST_SUCCESS if the PHY was written to successfully. Since there is no error status from the MAC on a write, the user should read the PHY to verify the write was successful.
- XST_NO_FEATURE if the device is not configured with MII support
- XST_EMAC_MII_BUSY if there is another PHY operation in progress

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that the write is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PhyWrite thread.

emac/v1_00_c/src/xemac_stats.c File Reference

Detailed Description

Contains functions to get and clear the **XEmac** driver statistics.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	07/31/01	First release
1.00b	rpm	02/20/02	Repartitioned files and functions
1.00c	rpm	12/05/02	New version includes support for simple DMA

```
#include "xbasic_types.h"  
#include "xemac_i.h"
```

Functions

```
void XEmac_GetStats (XEmac *InstancePtr, XEmac_Stats *StatsPtr)  
void XEmac_ClearStats (XEmac *InstancePtr)
```

Function Documentation

```
void XEmac_ClearStats( XEmac * InstancePtr)
```

Clear the XEmacStats structure for this driver.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

Returns:

None.

Note:

None.

```
void XEmac_GetStats( XEmac * InstancePtr,  
                    XEmac_Stats * StatsPtr  
                    )
```

Get a copy of the XEmacStats structure, which contains the current statistics for this driver. The statistics are only cleared at initialization or on demand using the **XEmac_ClearStats()** function.

The DmaErrors and FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the upper layer software.

Parameters:

InstancePtr is a pointer to the **XEmac** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

emac_lite/v1_00_a/src/xemac_lite_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions and macros that can be used to access the device.

The Xilinx Ethernet Lite driver component. This component supports the Xilinx Lite Ethernet 10/100 MAC (EMAC Lite).

The Xilinx Ethernet Lite 10/100 MAC supports the following features:

- Media Independent Interface (MII) for connection to external 10/100 Mbps PHY transceivers.
- Independent internal transmit and receive buffers
- CSMA/CD compliant operations for half-duplex modes
- Unicast and broadcast
- Automatic FCS insertion
- Automatic pad insertion on transmit

The Xilinx Ethernet Lite 10/100 MAC does not support the following features:

- interrupts
- multi-frame buffering only 1 transmit frame is allowed into the transmit buffer only 1 receive frame is allowed into the receive buffer. the hardware blocks reception until buffer is emptied
- Pause frame (flow control) detection in full-duplex mode
- Programmable interframe gap
- Multicast and promiscuous address filtering
- Internal loopback
- Automatic source address insertion or overwrite (programmable)

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to the EMAC Lite. The driver handles transmission and reception of Ethernet frames, as well as configuration of the controller. It does not handle protocol stack functionality such as Link Layer Control (LLC) or the Address Resolution Protocol (ARP). The protocol stack that makes use of the driver handles this functionality. This implies that the driver is simply a pass-through mechanism between a protocol stack and the EMAC Lite.

Since the driver is a simple pass-through mechanism between a protocol stack and the EMAC Lite, no assembly or disassembly of Ethernet frames is done at the driver-level. This assumes that the protocol stack passes a correctly formatted Ethernet frame to the driver for transmission, and that the driver does not validate the contents of an incoming frame.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	06/01/02	First release

```
#include "xbasic_types.h"
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XEmacLite_mIsTxDone(BaseAddress)
#define XEmacLite_mIsRxEmpty(BaseAddress)
```

Functions

```
void XEmacLite_SetMacAddress (Xuint32 BaseAddress, Xuint8 *AddressPtr)
void XEmacLite_SendFrame (Xuint32 BaseAddress, Xuint8 *FramePtr, unsigned Size)
Xuint16 XEmacLite_RecvFrame (Xuint32 BaseAddress, Xuint8 *FramePtr)
```

Define Documentation

```
#define XEmaLite_mIsRxEmpty( BaseAddress )
```

Check to see if the receive is empty.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is empty, or XFALSE if it is not.

Note:

Xboolean [XEmaLite_mIsRxEmpty](#)(Xuint32 BaseAddress)

```
#define XEmaLite_mIsTxDone( BaseAddress )
```

Check to see if the transmission is complete.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is done, or XFALSE if it is not.

Note:

Xboolean [XEmaLite_mIsTxDone](#)(Xuint32 BaseAddress)

Function Documentation

```
Xuint16 XEmaLite_RecvFrame( Xuint32 BaseAddress,  
                           Xuint8 * FramePtr  
                           )
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a buffer where the frame will be stored.

Returns:

The type/length field of the frame received. When the type/length field contains the type , XEL_RPLR_MAX_LENGTH bytes will be copied out of the buffer and it is up to the higher layers to sort out the frame.

Note:

This function call is blocking in nature, i.e. it will wait until a frame arrives.

The hardware is actually a 32-bit access of which the least significant 8 bits are actually used. This forces the strange 32-bit read of the receive buffer in the buffer copy and the increment by 4 in the source address.

```
void XEmacLite_SendFrame( Xuint32 BaseAddress,  
                          Xuint8 * FramePtr,  
                          unsigned Size  
                          )
```

Send an Ethernet frame. The size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to frame

Size is the size, in bytes, of the frame

Returns:

None.

Note:

This function call is blocking in nature, i.e. it will wait until the frame is transmitted.

The hardware is actually a 32-bit access of which the least significant 8 bits are actually used. This forces the strange 32-bit write of the byte array in the buffer copy and the increment by 4 in the destination address.

```
void XEmacLite_SetMacAddress( Xuint32 BaseAddress,  
                             Xuint8 * AddressPtr  
                             )
```

Set the MAC address for this device. The address is a 48-bit value.

Parameters:

BaseAddress is register base address of the XEmacLite device.

AddressPtr is a pointer to a 6-byte MAC address. the format of the MAC address is major octet to minor octet

Returns:

None.

Note:

TX must be idle and RX should be idle for deterministic results.

emac_lite/v1_00_a/src/xemac_lite_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xemac_lite_l.h,v 1.3 2002/07/25 14:21:57 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file emac_lite/v1_00_a/src/xemac_lite_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions and
00028 * macros that can be used to access the device.
00029 *
00030 * The Xilinx Ethernet Lite driver component. This component supports the Xilinx
00031 * Lite Ethernet 10/100 MAC (EMAC Lite).
00032 *
00033 * The Xilinx Ethernet Lite 10/100 MAC supports the following features:
00034 * - Media Independent Interface (MII) for connection to external
00035 *   10/100 Mbps PHY transceivers.
00036 * - Independent internal transmit and receive buffers
00037 * - CSMA/CD compliant operations for half-duplex modes
00038 * - Unicast and broadcast
00039 * - Automatic FCS insertion
00040 * - Automatic pad insertion on transmit
00041 *
00042 * The Xilinx Ethernet Lite 10/100 MAC does not support the following features:
```

```

00043 * - interrupts
00044 * - multi-frame buffering
00045 *     only 1 transmit frame is allowed into the transmit buffer
00046 *     only 1 receive frame is allowed into the receive buffer.
00047 *     the hardware blocks reception until buffer is emptied
00048 * - Pause frame (flow control) detection in full-duplex mode
00049 * - Programmable interframe gap
00050 * - Multicast and promiscuous address filtering
00051 * - Internal loopback
00052 * - Automatic source address insertion or overwrite (programmable)
00053 *
00054 * <b>Driver Description</b>
00055 *
00056 * The device driver enables higher layer software (e.g., an application) to
00057 * communicate to the EMAC Lite. The driver handles transmission and reception
00058 * of Ethernet frames, as well as configuration of the controller. It does not
00059 * handle protocol stack functionality such as Link Layer Control (LLC) or the
00060 * Address Resolution Protocol (ARP). The protocol stack that makes use of the
00061 * driver handles this functionality. This implies that the driver is simply a
00062 * pass-through mechanism between a protocol stack and the EMAC Lite.
00063 *
00064 * Since the driver is a simple pass-through mechanism between a protocol stack
00065 * and the EMAC Lite, no assembly or disassembly of Ethernet frames is done at
00066 * the driver-level. This assumes that the protocol stack passes a correctly
00067 * formatted Ethernet frame to the driver for transmission, and that the driver
00068 * does not validate the contents of an incoming frame.
00069 *
00070 * @note
00071 *
00072 * None
00073 *
00074 * <pre>
00075 * MODIFICATION HISTORY:
00076 *
00077 * Ver    Who    Date      Changes
00078 * -----
00079 * 1.00a  ecm    06/01/02  First release
00080 * </pre>
00081 *
00082 * *****/
00083 *
00084 #ifndef XEMAC_LITE_L_H /* prevent circular inclusions */
00085 #define XEMAC_LITE_L_H /* by using protection macros */
00086 *
00087 /***** Include Files *****/
00088 *
00089 #include "xbasic_types.h"
00090 #include "xio.h"
00091 *
00092 /***** Constant Definitions *****/
00093 *
00094 /* Offset of the MAC registers from the IPIF base address */

```

```

00095 #define XEL_REG_OFFSET      0x0UL
00096
00097 /*
00098  * Register offsets for the Ethernet MAC.
00099  */
00100 #define XEL_TXBUFF_OFFSET (XEL_REG_OFFSET) /*
Transmit Buffer */
00101 #define XEL_TSR_OFFSET      (XEL_TXBUFF_OFFSET + 0x1FFC) /* Tx status */
00102 #define XEL_TPLR_OFFSET    (XEL_TXBUFF_OFFSET + 0x1FF4) /* Tx packet length */
00103
00104 #define XEL_RXBUFF_OFFSET    (XEL_REG_OFFSET + 0x2000) /* Receive Buffer */
00105 #define XEL_RSR_OFFSET      (XEL_RXBUFF_OFFSET + 0x1FFC) /* Rx status */
00106 #define XEL_RPLR_OFFSET    (XEL_RXBUFF_OFFSET + 0x30) /* Rx packet length */
00107
00108 #define XEM_MAC_HI_OFFSET   (XEL_TXBUFF_OFFSET + 0x14) /* MAC address hi
offset */
00109 #define XEM_MAC_LO_OFFSET   (XEL_TXBUFF_OFFSET) /* MAC address
lo offset */
00110 /*
00111  * Transmit Status Register (TSR)
00112  */
00113 #define XEL_TSR_XMIT_BUSY_MASK      0x01UL /* Xmit complete */
00114 #define XEL_TSR_PROGRAM_MASK      0x02UL /* Program the MAC address */
00115 /*
00116  * define for programming the MAC address into the EMAC Lite
00117  */
00118
00119 #define XEL_TSR_PROG_MAC_ADDR      (XEL_TSR_XMIT_BUSY_MASK | XEL_TSR_PROGRAM_MASK)
00120
00121 /*
00122  * Receive Status Register (RSR)
00123  */
00124 #define XEL_RSR_RECV_DONE_MASK      0x01UL /* Recv complete */
00125
00126 /*
00127  * Transmit Packet Length Register (TPLR)
00128  */
00129 #define XEL_TPLR_LENGTH_MASK_HI      0x0000FF00UL /* Transmit packet length
upper byte */
00130 #define XEL_TPLR_LENGTH_MASK_LO      0x000000FFUL /* Transmit packet length
lower byte */
00131
00132 /*
00133  * Receive Packet Length Register (RPLR)
00134  */
00135 #define XEL_RPLR_LENGTH_MASK_HI      0x0000FF00UL /* Receive packet length
upper byte */
00136 #define XEL_RPLR_LENGTH_MASK_LO      0x000000FFUL /* Receive packet length
lower byte */
00137
00138 #define XEL_HEADER_SIZE      14 /*
size of header */
00139 #define XEL_MTU_SIZE      1500 /* max size of

```

```

data in frame */
00140 #define XEL_CRC_SIZE 4 /*
size of CRC */
00141
00142
00143 #define XEL_RPLR_MAX_LENGTH
(XEL_HEADER_SIZE+XEL_MTU_SIZE+XEL_CRC_SIZE)
00144
/* maximum length of rx frame */
00145
/* used if length/type field */
00146
/* contains the type (> 1500) */
00147
00148
00149 #define XEL_MAC_ADDR_SIZE 6 /*
length of MAC address */
00150
00151
00152 /***** Macros (Inline Functions) Definitions *****/
00153
00154 /*****/
00155 /**
00156 *
00157 * Check to see if the transmission is complete.
00158 *
00159 * @param BaseAddress is the base address of the device
00160 *
00161 * @return XTRUE if it is done, or XFALSE if it is not.
00162 *
00163 * @note
00164 * Xboolean XEmacLite_mIsTxDone(Xuint32 BaseAddress)
00165 *
00166 *****/
00167 #define XEmacLite_mIsTxDone(BaseAddress) \
00168 (! (XIo_In32((BaseAddress) + XEL_TSR_OFFSET) &
XEL_TSR_XMIT_BUSY_MASK))
00169
00170
00171 /*****/
00172 /**
00173 *
00174 * Check to see if the receive is empty.
00175 *
00176 * @param BaseAddress is the base address of the device
00177 *
00178 * @return XTRUE if it is empty, or XFALSE if it is not.
00179 *
00180 * @note
00181 * Xboolean XEmacLite_mIsRxEmpty(Xuint32 BaseAddress)
00182 *
00183 *****/

```

```
00184 #define XEmaLite_mIsRxEmpty(BaseAddress) \  
00185         (!(XIo_In32((BaseAddress) + XEL_RSR_OFFSET) &  
XEmaLite_RSR_RECV_DONE_MASK))  
00186  
00187 /***** Function Prototypes *****/  
00188  
00189 void XEmaLite_SetMacAddress(Xuint32 BaseAddress, Xuint8 *AddressPtr);  
00190 void XEmaLite_SendFrame(Xuint32 BaseAddress, Xuint8 *FramePtr, unsigned Size);  
00191 Xuint16 XEmaLite_RecvFrame(Xuint32 BaseAddress, Xuint8 *FramePtr);  
00192  
00193  
00194 #endif /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

emacLite/v1_00_a/src/xemacLite_1.c File Reference

Detailed Description

This file contains the minimal, polled functions to send and receive Ethernet frames.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	06/01/02	First release

```
#include "xbasic_types.h"  
#include "xemacLite_1.h"
```

Functions

```
void XEmaCLite_SendFrame (Xuint32 BaseAddress, Xuint8 *FramePtr, unsigned Size)  
Xuint16 XEmaCLite_RecvFrame (Xuint32 BaseAddress, Xuint8 *FramePtr)  
void XEmaCLite_SetMacAddress (Xuint32 BaseAddress, Xuint8 *AddressPtr)
```

Function Documentation

```
Xuint16 XEmaLite_RecvFrame( Xuint32 BaseAddress,  
                             Xuint8 * FramePtr  
                             )
```

Receive a frame. Wait for a frame to arrive.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to a buffer where the frame will be stored.

Returns:

The type/length field of the frame received. When the type/length field contains the type , XEL_RPLR_MAX_LENGTH bytes will be copied out of the buffer and it is up to the higher layers to sort out the frame.

Note:

This function call is blocking in nature, i.e. it will wait until a frame arrives.

The hardware is actually a 32-bit access of which the least significant 8 bits are actually used. This forces the strange 32-bit read of the receive buffer in the buffer copy and the increment by 4 in the source address.

```
void XEmaLite_SendFrame( Xuint32 BaseAddress,  
                          Xuint8 * FramePtr,  
                          unsigned Size  
                          )
```

Send an Ethernet frame. The size is the total frame size, including header. This function blocks waiting for the frame to be transmitted.

Parameters:

BaseAddress is the base address of the device

FramePtr is a pointer to frame

Size is the size, in bytes, of the frame

Returns:

None.

Note:

This function call is blocking in nature, i.e. it will wait until the frame is transmitted.

The hardware is actually a 32-bit access of which the least significant 8 bits are actually used. This forces the strange 32-bit write of the byte array in the buffer copy and the increment by 4 in the destination address.

```
void XEmacLite_SetMacAddress( Xuint32 BaseAddress,  
                             Xuint8 * AddressPtr  
                             )
```

Set the MAC address for this device. The address is a 48-bit value.

Parameters:

BaseAddress is register base address of the XEmacLite device.

AddressPtr is a pointer to a 6-byte MAC address. the format of the MAC address is major octet to minor octet

Returns:

None.

Note:

TX must be idle and RX should be idle for deterministic results.

emc/v1_00_a/src/xemc.h File Reference

Detailed Description

This file contains the software API definition of the Xilinx External Memory Controller (**XEmc**) component. This controller can be attached to host OPB or PLB buses to control multiple banks of supported memory devices. The type of host bus is transparent to software.

This driver allows the user to access the device's registers to support fast/slow access to the memory devices as well as enabling/disabling paged mode access.

The Xilinx OPB/PLB External memory controller is a soft IP core designed for Xilinx FPGAs and contains the following general features:

- Support for 128, 64, 32, 16, and 8 bit bus interfaces.
- Controls up to 8 banks of supported memory devices.
- Separate control register for each bank of memory.
- Selectable wait state control (fast or slow). (See note 1)
- Supports page mode accesses. Page size is 8 bytes.
- System clock frequency of up to 133 MHz.

OPB features:

- OPB V2.0 bus interface with byte-enable support.
- Memory width of connected devices is the same as or smaller than OPB bus width.

Note:

(1) The number of wait states inserted for fast and slow mode is determined by the HW designer and is hard-coded into the IP. Each bank has its own settings.

(2) For read accesses, fast/slow access mode is ignored when page mode is enabled. For write accesses, page mode does not apply. (3) This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads. MODIFICATION HISTORY: Ver Who Date Changes -----
----- 1.00a rmm 01/29/02 First release 1.00a rpm 05/14/02 Made configuration
table/lookup public

```
#include "xbasic_types.h"
#include "xstatus.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XEmc
struct XEmc_Config
```

Functions

```
XStatus XEmc_Initialize (XEmc *InstancePtr, Xuint16 DeviceId)
XEmc_Config * XEmc_LookupConfig (Xuint16 DeviceId)
XStatus XEmc_SetPageMode (XEmc *InstancePtr, unsigned Bank, unsigned Mode)
XStatus XEmc_SetAccessSpeed (XEmc *InstancePtr, unsigned Bank, unsigned Speed)
unsigned XEmc_GetPageMode (XEmc *InstancePtr, unsigned Bank)
unsigned XEmc_GetAccessSpeed (XEmc *InstancePtr, unsigned Bank)
XStatus XEmc_SelfTest (XEmc *InstancePtr)
```

Function Documentation

```
unsigned XEmc_GetAccessSpeed( XEmc * InstancePtr,
                             unsigned Bank
                             )
```

Gets current access speed setting for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to retrieve the setting for. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in **xparameters.h** (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Returns:

Current access speed of bank. XEMC_ACCESS_SPEED_FAST or XEMC_ACCESS_SPEED_SLOW.

Note:

none

```
unsigned XEmc_GetPageMode( XEmc * InstancePtr,  
                           unsigned Bank  
                           )
```

Gets the current page mode setting for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to retrieve the setting for. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in [xparameters.h](#) (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Returns:

Current mode of bank. XEMC_PAGE_MODE_ENABLE or XEMC_PAGE_MODE_DISABLE.

Note:

none

```
XStatus XEmc_Initialize( XEmc * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initializes the **XEmc** instance provided by the caller based on the given DeviceID.

Parameters:

InstancePtr is a pointer to an **XEmc** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XEmc** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XEmc** component. Passing in a device id associates the generic **XEmc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- o XST_SUCCESS Initialization was successful.
- o XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

The control registers for each bank are not modified because it is possible that they have been setup during bootstrap processing prior to "C" runtime support.

XEmc_Config* XEmc_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table XEmc_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceID is the device identifier to lookup.

Returns:

XEmc configuration structure pointer if DeviceID is found.

XNULL if DeviceID is not found.

XStatus XEmc_SelfTest(XEmc * InstancePtr)

Runs a self-test on the driver/device. This includes the following tests:

- Control register read/write access for each bank.

Memory devices controlled by this component are not changed. However access speeds are toggled which could possibly have undesirable effects.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on. This parameter must have been previously initialized with **XEmc_Initialize()**.

Returns:

XST_SUCCESS If test passed

XST_FAILURE If test failed

Note:

- Control register contents are restored to their original state when the test completes.
- This test does not abort if an error is detected.

XStatus XEmc_SetAccessSpeed(XEmc * InstancePtr, unsigned Bank, unsigned Speed)

Sets the access speed for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to change. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in **xparameters.h** (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Speed is the new access speed. Valid speeds are XEMC_ACCESS_SPEED_SLOW and XEMC_ACCESS_SPEED_FAST.

Returns:

- XST_SUCCESS Access speed successfully set.
- XST_INVALID_PARAM Speed parameter is invalid.

Note:

none

```
XStatus XEmc_SetPageMode( XEmc * InstancePtr,  
                          unsigned Bank,  
                          unsigned Mode  
                          )
```

Sets the page mode for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to change. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in **xparameters.h** (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Mode is the new mode to set. Valid modes are XEMC_PAGE_MODE_ENABLE and XEMC_PAGE_MODE_DISABLE.

Returns:

- XST_SUCCESS Mode successfully set.
- XST_INVALID_PARAM Mode parameter is invalid.

Note:

none

XEmc Struct Reference

```
#include <xemc.h>
```

Detailed Description

The XEmc driver instance data. The user is required to allocate a variable of this type for every EMC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `emc/v1_00_a/src/xemc.h`
-

emc/v1_00_a/src/xemc.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xemc.h,v 1.5 2002/05/16 17:21:35 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 * @file emc/v1_00_a/src/xemc.h
00025 *
00026 * This file contains the software API definition of the Xilinx External Memory
00027 * Controller (XEmc) component. This controller can be attached to host OPB or
00028 * PLB buses to control multiple banks of supported memory devices. The type
00029 * of host bus is transparent to software.
00030 *
00031 * This driver allows the user to access the device's registers to support
00032 * fast/slow access to the memory devices as well as enabling/disabling paged
00033 * mode access.
00034 *
00035 * The Xilinx OPB/PLB External memory controller is a soft IP core designed for
00036 * Xilinx FPGAs and contains the following general features:
00037 * - Support for 128, 64, 32, 16, and 8 bit bus interfaces.
00038 * - Controls up to 8 banks of supported memory devices.
00039 * - Separate control register for each bank of memory.
00040 * - Selectable wait state control (fast or slow). (See note 1)
00041 * - Supports page mode accesses. Page size is 8 bytes.
00042 * - System clock frequency of up to 133 MHz.
```

```

00043 *
00044 * OPB features:
00045 *   - OPB V2.0 bus interface with byte-enable support.
00046 *   - Memory width of connected devices is the same as or smaller than OPB
00047 *     bus width.
00048 *
00049 * @note
00050 * <pre>
00051 * (1) The number of wait states inserted for fast and slow mode is determined
00052 *     by the HW designer and is hard-coded into the IP. Each bank has its
00053 *     own settings.
00054 *
00055 * (2) For read accesses, fast/slow access mode is ignored when page mode
00056 *     is enabled. For write accesses, page mode does not apply.
00057 *
00058 * (3) This driver is not thread-safe. Thread safety must be guaranteed by
00059 *     the layer above this driver if there is a need to access the device
00060 *     from multiple threads.
00061 *
00062 * MODIFICATION HISTORY:
00063 *
00064 * Ver   Who   Date     Changes
00065 * -----
00066 * 1.00a rmm   01/29/02 First release
00067 * 1.00a rpm   05/14/02 Made configuration table/lookup public
00068 * </pre>
00069 *
00070 *****/
00071 #ifndef XEMC_H /* prevent circular inclusions */
00072 #define XEMC_H /* by using protection macros */
00073
00074 /***** Include Files *****/
00075 #include "xbasic_types.h"
00076 #include "xstatus.h"
00077
00078 /***** Constant Definitions *****/
00079
00080 /** @name Page mode options
00081  *
00082  * Page modes used in XEMc_SetPageMode().
00083  * @{
00084  */
00085 #define XEMC_PAGE_MODE_ENABLE 0
00086 #define XEMC_PAGE_MODE_DISABLE 1
00087 /* @} */
00088
00089 /** @name Access speed options
00090  *
00091  * Access speeds used in XEMc_SetAccessSpeed(). What constitutes fast and
00092  * slow are defined by the IP design.
00093  * @{
00094  */

```



```

00095 #define XEMC_ACCESS_SPEED_FAST 2
00096 #define XEMC_ACCESS_SPEED_SLOW 3
00097 /*@}*/
00098
00099 /***** Type Definitions *****/
00100
00101 /**
00102  * This typedef contains configuration information for the device.
00103  */
00104 typedef struct
00105 {
00106     Xuint16 DeviceId;      /**< Unique ID of device */
00107     Xuint32 RegBaseAddr;  /**< Register base address */
00108     Xuint8  NumBanks;     /**< Number of devices controlled by this component
00109 } XEmc_Config;
00110
00111
00112 /**
00113  * The XEmc driver instance data. The user is required to allocate a
00114  * variable of this type for every EMC device in the system. A pointer
00115  * to a variable of this type is then passed to the driver API functions.
00116  */
00117 typedef struct
00118 {
00119     Xuint32 RegBaseAddr;  /* Base address of registers */
00120     Xuint32 IsReady;      /* Device is initialized and ready */
00121     Xuint8  NumBanks;     /* Number of device banks under control */
00122 } XEmc;
00123
00124
00125 /***** Macros (Inline Functions) Definitions *****/
00126
00127
00128 /***** Function Prototypes *****/
00129
00130 /*
00131  * Initialization and configuration functions. These functions are mandatory.
00132  * This API is implemented in xemc.c
00133  */
00134 XStatus XEmc_Initialize(XEmc *InstancePtr, Xuint16 DeviceId);
00135 XEmc_Config *XEmc_LookupConfig(Xuint16 DeviceId);
00136
00137 XStatus XEmc_SetPageMode(XEmc *InstancePtr, unsigned Bank, unsigned Mode);
00138 XStatus XEmc_SetAccessSpeed(XEmc *InstancePtr, unsigned Bank, unsigned Speed);
00139 unsigned XEmc_GetPageMode(XEmc *InstancePtr, unsigned Bank);
00140 unsigned XEmc_GetAccessSpeed(XEmc *InstancePtr, unsigned Bank);
00141
00142 /*
00143  * Selftest is optional
00144  * This API is implemented in xemc_selftest.c

```

```
00145 */
00146 XStatus XEmc_SelfTest(XEmc *InstancePtr);
00147
00148 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

XEmc_Config Struct Reference

```
#include <xemc.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 RegBaseAddr

Xuint8 NumBanks

Field Documentation

Xuint16 XEmc_Config::DeviceId

Unique ID of device

Xuint8 XEmc_Config::NumBanks

Number of devices controlled by this component

Xuint32 XEmc_Config::RegBaseAddr

Register base address

The documentation for this struct was generated from the following file:

- [emc/v1_00_a/src/xemc.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

emc/v1_00_a/src/xemc.c File Reference

Detailed Description

The implementation of the [XEmc](#) component. See [xemc.h](#) for more information about the component.

Note:

None

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	01/29/02	First release
1.00a	rpm	05/14/02	Made configuration table/lookup public

```
#include "xemc.h"  
#include "xemc_i.h"  
#include "xparameters.h"
```

Functions

[XStatus XEmc_Initialize](#) ([XEmc](#) *InstancePtr, [Xuint16](#) DeviceId)
[XEmc_Config](#) * [XEmc_LookupConfig](#) ([Xuint16](#) DeviceId)
[XStatus XEmc_SetPageMode](#) ([XEmc](#) *InstancePtr, unsigned Bank, unsigned Mode)
[XStatus XEmc_SetAccessSpeed](#) ([XEmc](#) *InstancePtr, unsigned Bank, unsigned Speed)
unsigned [XEmc_GetPageMode](#) ([XEmc](#) *InstancePtr, unsigned Bank)
unsigned [XEmc_GetAccessSpeed](#) ([XEmc](#) *InstancePtr, unsigned Bank)

Function Documentation

```
unsigned XEmc_GetAccessSpeed( XEmc * InstancePtr,  
                             unsigned Bank  
                             )
```

Gets current access speed setting for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to retrieve the setting for. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in [xparameters.h](#) (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Returns:

Current access speed of bank. XEMC_ACCESS_SPEED_FAST or XEMC_ACCESS_SPEED_SLOW.

Note:

none

```
unsigned XEmc_GetPageMode( XEmc * InstancePtr,  
                           unsigned Bank  
                           )
```

Gets the current page mode setting for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to retrieve the setting for. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in [xparameters.h](#) (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Returns:

Current mode of bank. XEMC_PAGE_MODE_ENABLE or XEMC_PAGE_MODE_DISABLE.

Note:

none

```
XStatus XEmc_Initialize( XEmc * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initializes the **XEmc** instance provided by the caller based on the given DeviceID.

Parameters:

InstancePtr is a pointer to an **XEmc** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XEmc** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XEmc** component. Passing in a device id associates the generic **XEmc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS Initialization was successful.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

The control registers for each bank are not modified because it is possible that they have been setup during bootstrap processing prior to "C" runtime support.

```
XEmc_Config* XEmc_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID. The table XEmc_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceID is the device identifier to lookup.

Returns:

XEmc configuration structure pointer if DeviceID is found.

XNULL if DeviceID is not found.

```
XStatus XEmc_SetAccessSpeed( XEmc * InstancePtr,  
                             unsigned Bank,  
                             unsigned Speed  
                             )
```

Sets the access speed for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to change. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in [xparameters.h](#) (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Speed is the new access speed. Valid speeds are XEMC_ACCESS_SPEED_SLOW and XEMC_ACCESS_SPEED_FAST.

Returns:

- XST_SUCCESS Access speed successfully set.
- XST_INVALID_PARAM Speed parameter is invalid.

Note:

none

```
XStatus XEmc_SetPageMode( XEmc * InstancePtr,  
                          unsigned Bank,  
                          unsigned Mode  
                          )
```


Sets the page mode for the given bank of memory devices.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on.

Bank is the set of devices to change. Valid range is 0 to the number of banks minus one. The number of banks is defined as a constant in **xparameters.h** (XPAR_EMC_<n>_NUM_BANKS) or it can be found in the NumBanks attribute of the **XEmc** instance.

Mode is the new mode to set. Valid modes are XEMC_PAGE_MODE_ENABLE and XEMC_PAGE_MODE_DISABLE.

Returns:

- XST_SUCCESS Mode successfully set.
- XST_INVALID_PARAM Mode parameter is invalid.

Note:

none

emc/v1_00_a/src/xemc_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xemc_i.h,v 1.4 2002/05/14 18:14:39 moleres Exp $: */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file emc/v1_00_a/src/xemc_i.h
00026 *
00027 * This header file contains register offsets and bit definitions for the
00028 * external memory controller (EMC). The definitions here are meant to be
00029 * used for internal xemc driver purposes.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date      Changes
00035 * -----
00036 * 1.00a  rmm    02/04/02  First release
00037 * 1.00a  rpm    05/14/02  Moved identifiers to xemc_l.h
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XEMC_I_H /* prevent circular inclusions */
```

```
00043 #define XEMC_I_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046 #include "xemc_1.h"
00047
00048 /***** Constant Definitions *****/
00049
00050 /***** Type Definitions *****/
00051
00052 /***** Macros (Inline Functions) Definitions *****/
00053
00054 /***** Function Prototypes *****/
00055
00056 /***** Variables *****/
00057
00058 extern XEmc_Config XEmc_ConfigTable[];
00059
00060
00061 #endif /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

emc/v1_00_a/src/xemc_i.h File Reference

Detailed Description

This header file contains register offsets and bit definitions for the external memory controller (EMC). The definitions here are meant to be used for internal xemc driver purposes.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	02/04/02	First release
1.00a	rpm	05/14/02	Moved identifiers to xemc_1.h

```
#include "xemc\_1.h"
```

[Go to the source code of this file.](#)

emc/v1_00_a/src/xemc_l.h File Reference

Detailed Description

Contains identifiers and low-level macros that can be used to access the device directly. High-level functions are defined in [xemc.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	05/14/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XEmc_mGetOffset(Bank)  
#define XEmc_mGetControlReg(Base, Bank)  
#define XEmc_mSetControlReg(Base, Bank, Data)  
#define XEmc_mEnablePageMode(BaseAddress, Bank)  
#define XEmc_mDisablePageMode(BaseAddress, Bank)  
#define XEmc_mEnableFastAccess(BaseAddress, Bank)  
#define XEmc_mDisableFastAccess(BaseAddress, Bank)
```

Define Documentation

```
#define XEmc_mDisableFastAccess( BaseAddress,  
                                Bank          )
```

Disable fast access in the given memory bank.

Parameters:

BaseAddress is the base address of the device

Bank is the memory bank to set.

Returns:

None.

Note:

None.

```
#define XEmc_mDisablePageMode( BaseAddress,  
                               Bank          )
```

Disable page mode in the given memory bank.

Parameters:

BaseAddress is the base address of the device

Bank is the memory bank to set.

Returns:

None.

Note:

None.

```
#define XEmc_mEnableFastAccess( BaseAddress,  
                                Bank          )
```

Enable fast access in the given memory bank.

Parameters:

BaseAddress is the base address of the device
Bank is the memory bank to set.

Returns:

None.

Note:

None.

```
#define XEmc_mEnablePageMode( BaseAddress,  
                             Bank )
```

Enable page mode in the given memory bank.

Parameters:

BaseAddress is the base address of the device
Bank is the memory bank to set.

Returns:

None.

Note:

None.

```
#define XEmc_mGetControlReg( Base,  
                             Bank )
```

Reads the contents of a bank's control register.

Parameters:

Base is the base address of the EMC component.

Bank identifies the control register to read.

Returns:

Value of the Bank's control register

Note:

- Macro signature: Xuint32 **XEmc_mGetControlReg**(Xuint32 Base, unsigned Bank)

```
#define XEmc_mGetOffset( Bank )
```

Calculate the offset of a control register based on its bank. This macro is used internally.

Parameters:

Bank is the bank number of the control register offset to calculate

Returns:

Offset to control register associated with Bank parameter.

Note:

- To compute the physical address of the register add the base address of the component to the result of this macro.
- Does not test for validity of Bank.
- Macro signature: unsigned **XEmc_mGetOffset**(unsigned Bank)

```
#define XEmc_mSetControlReg( Base,  
                             Bank,  
                             Data )
```


Writes to a bank's control register.

Parameters:

Base is the base address of the EMC component.

Bank identifies the control register to modify.

Data is the data to write to the control register.

Returns:

None.

Note:

- Macro signature: void **XEmc_mSetControlReg**(Xuint32 Base, unsigned Bank, Xuint32 Data)

emc/v1_00_a/src/xemc_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xemc_l.h,v 1.1 2002/05/14 18:13:56 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file emc/v1_00_a/src/xemc_l.h
00026 *
00027 * Contains identifiers and low-level macros that can be used to access the
00028 * device directly. High-level functions are defined in xemc.h.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver   Who   Date       Changes
00034 * -----
00035 * 1.00a rpm   05/14/02 First release
00036 * </pre>
00037 *
00038 *****/
00039 #ifndef XEMC_L_H /* prevent circular inclusions */
00040 #define XEMC_L_H /* by using protection macros */
00041
00042 /***** Include Files *****/
```

```

00043 #include "xbasic_types.h"
00044 #include "xio.h"
00045
00046 /***** Constant Definitions *****/
00047
00048 /*
00049  * Register offsets.
00050  *
00051  * There is a single control register for each bank.
00052  *
00053  * The EMC can be configured with between 1 to 8 banks of memory devices.
00054  * See xparameters.h for the definition for the number of banks present.
00055  */
00056 #define XEMC_CR0_OFFSET    0x00000000    /* Control register for bank 0 */
00057 #define XEMC_CR1_OFFSET    0x00000004    /* Control register for bank 1 */
00058 #define XEMC_CR2_OFFSET    0x00000008    /* Control register for bank 2 */
00059 #define XEMC_CR3_OFFSET    0x0000000C    /* Control register for bank 3 */
00060 #define XEMC_CR4_OFFSET    0x00000010    /* Control register for bank 4 */
00061 #define XEMC_CR5_OFFSET    0x00000014    /* Control register for bank 5 */
00062 #define XEMC_CR6_OFFSET    0x00000018    /* Control register for bank 6 */
00063 #define XEMC_CR7_OFFSET    0x0000001C    /* Control register for bank 7 */
00064
00065 /*
00066  * Masks and bit definitions for the CR register
00067  */
00068 #define XEMC_CR_FS_ENABLE  0x00000001    /* Fast/slow mode enable.
00069                                         0-slow,
00070                                         1-fast */
00071 #define XEMC_CR_PM_ENABLE  0x00000002    /* Page mode enable.
00072                                         0-page mode disable
00073                                         1-page mode enable */
00074
00075 /***** Type Definitions *****/
00076
00077 /***** Macros (Inline Functions) Definitions *****/
00078
00079 /*****
00080  *
00081  * Low-level driver macros and functions. The list below provides signatures
00082  * to help the user use the macros.
00083  *
00084  * Xuint32 XEmc_mGetControlReg(Xuint32 BaseAddress, unsigned Bank)
00085  * void XEmc_mSetControlReg(Xuint32 BaseAddress, unsigned Bank, Xuint32 Data)
00086  *
00087  * void XEmc_mEnablePageMode(Xuint32 BaseAddress, unsigned Bank)
00088  * void XEmc_mDisablePageMode(Xuint32 BaseAddress, unsigned Bank)
00089  * void XEmc_mEnableFastAccess(Xuint32 BaseAddress, unsigned Bank)
00090  * void XEmc_mDisableFastAccess(Xuint32 BaseAddress, unsigned Bank)
00091  *
00092  *****/
00093
00094 /*****

```

```

00095 /**
00096 *
00097 * Calculate the offset of a control register based on its bank. This macro is
00098 * used internally.
00099 *
00100 * @param      Bank is the bank number of the control register offset to calculate
00101 *
00102 * @return     Offset to control register associated with Bank parameter.
00103 *
00104 * @note
00105 *
00106 * - To compute the physical address of the register add the
00107 *   base address of the component to the result of this macro.
00108 * - Does not test for validity of Bank.
00109 * - Macro signature: unsigned XEmc_mGetOffset(unsigned Bank)
00110 *
00111 *****/
00112 #define XEmc_mGetOffset(Bank) (XEMC_CR0_OFFSET + (4 * Bank))
00113
00114
00115 /**
00116 /**
00117 *
00118 * Reads the contents of a bank's control register.
00119 *
00120 * @param      Base is the base address of the EMC component.
00121 * @param      Bank identifies the control register to read.
00122 *
00123 * @return     Value of the Bank's control register
00124 *
00125 * @note
00126 *
00127 * - Macro signature:
00128 *   Xuint32 XEmc_mGetControlReg(Xuint32 Base, unsigned Bank)
00129 *
00130 *****/
00131 #define XEmc_mGetControlReg(Base, Bank) \
00132     XIo_In32((XIo_Address)((Base) + XEmc_mGetOffset(Bank)))
00133
00134
00135 /**
00136 /**
00137 *
00138 * Writes to a bank's control register.
00139 *
00140 * @param      Base is the base address of the EMC component.
00141 * @param      Bank identifies the control register to modify.
00142 * @param      Data is the data to write to the control register.
00143 *
00144 * @return     None.
00145 *
00146 * @note

```

```
00147 *
00148 * - Macro signature:
00149 *     void XEmc_mSetControlReg(Xuint32 Base, unsigned Bank, Xuint32 Data)
00150 *
00151 *****/
00152 #define XEmc_mSetControlReg(Base, Bank, Data) \
00153     XIo_Out32((XIo_Address)((Base) + XEmc_mGetOffset(Bank)), (Data))
00154
00155
00156 /*****/
00157 /**
00158 *
00159 * Enable page mode in the given memory bank.
00160 *
00161 * @param     BaseAddress is the base address of the device
00162 * @param     Bank is the memory bank to set.
00163 *
00164 * @return    None.
00165 *
00166 * @note     None.
00167 *
00168 *****/
00169 #define XEmc_mEnablePageMode(BaseAddress, Bank) \
00170     XEmc_mSetControlReg(BaseAddress, Bank, \
00171         XEmc_mGetControlReg(BaseAddress, Bank) | XEMC_CR_PM_ENABLE)
00172
00173
00174 /*****/
00175 /**
00176 *
00177 * Disable page mode in the given memory bank.
00178 *
00179 * @param     BaseAddress is the base address of the device
00180 * @param     Bank is the memory bank to set.
00181 *
00182 * @return    None.
00183 *
00184 * @note     None.
00185 *
00186 *****/
00187 #define XEmc_mDisablePageMode(BaseAddress, Bank) \
00188     XEmc_mSetControlReg(BaseAddress, Bank, \
00189         XEmc_mGetControlReg(BaseAddress, Bank) & ~XEMC_CR_PM_ENABLE)
00190
00191
00192 /*****/
00193 /**
00194 *
00195 * Enable fast access in the given memory bank.
00196 *
00197 * @param     BaseAddress is the base address of the device
00198 * @param     Bank is the memory bank to set.
```

```

00199 *
00200 * @return    None.
00201 *
00202 * @note      None.
00203 *
00204 *****/
00205 #define XEmc_mEnableFastAccess(BaseAddress, Bank) \
00206         XEmc_mSetControlReg(BaseAddress, Bank, \
00207         XEmc_mGetControlReg(BaseAddress, Bank) | XEMC_CR_FS_ENABLE)
00208
00209
00210 *****/
00211 /**
00212 *
00213 * Disable fast access in the given memory bank.
00214 *
00215 * @param    BaseAddress is the base address of the device
00216 * @param    Bank is the memory bank to set.
00217 *
00218 * @return    None.
00219 *
00220 * @note      None.
00221 *
00222 *****/
00223 #define XEmc_mDisableFastAccess(BaseAddress, Bank) \
00224         XEmc_mSetControlReg(BaseAddress, Bank, \
00225         XEmc_mGetControlReg(BaseAddress, Bank) & ~XEMC_CR_FS_ENABLE)
00226
00227
00228 *****/ Function Prototypes *****/
00229
00230 #endif /* end of protection macro */

```

emc/v1_00_a/src/xemc_selftest.c File Reference

Detailed Description

The implementation of the **XEmc** component for the self-test functions.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	02/08/02	First release

```
#include "xemc.h"  
#include "xemc_i.h"
```

Functions

XStatus **XEmc_SelfTest** (**XEmc** *InstancePtr)

Function Documentation

XStatus **XEmc_SelfTest**(**XEmc** * *InstancePtr*)

Runs a self-test on the driver/device. This includes the following tests:

- Control register read/write access for each bank.

Memory devices controlled by this component are not changed. However access speeds are toggled which could possibly have undesirable effects.

Parameters:

InstancePtr is a pointer to the **XEmc** instance to be worked on. This parameter must have been previously initialized with **XEmc_Initialize()**.

Returns:

XST_SUCCESS If test passed

XST_FAILURE If test failed

Note:

- Control register contents are restored to their original state when the test completes.
- This test does not abort if an error is detected.

common/v1_00_a/src/xenv_vxworks.h

File Reference

Detailed Description

Defines common services that are typically found in a VxWorks target environment.

Note:

This file is not intended to be included directly by driver code. Instead, the generic [xenv.h](#) file is intended to be included by driver code.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	rmm	10/24/02	Added XENV_USLEEP macro
1.00a	rmm	07/16/01	First release

```
#include "xbasic\_types.h"  
#include "vxWorks.h"  
#include "vxLib.h"  
#include <string.h>
```

[Go to the source code of this file.](#)

Data Structures

struct [XENV_TIME_STAMP](#)

Defines

```
#define XENV\_MEM\_COPY(DestPtr, SrcPtr, Bytes)
#define XENV\_MEM\_FILL(DestPtr, Data, Bytes)
#define XENV\_TIME\_STAMP\_GET(StampPtr)
#define XENV\_TIME\_STAMP\_DELTA\_US(Stamp1Ptr, Stamp2Ptr)
#define XENV\_TIME\_STAMP\_DELTA\_MS(Stamp1Ptr, Stamp2Ptr)
#define XENV\_USLEEP(delay)
```

Define Documentation

```
#define XENV\_MEM\_COPY( DestPtr,  
                    SrcPtr,  
                    Bytes  )
```

Copies a non-overlapping block of memory.

Parameters:

DestPtr is the destination address to copy data to.

SrcPtr is the source address to copy data from.

Bytes is the number of bytes to copy.

Returns:

None.

Note:

Signature: void [XENV_MEM_COPY](#)(void *DestPtr, void *SrcPtr, unsigned Bytes)

```
#define XENV\_MEM\_FILL( DestPtr,  
                    Data,  
                    Bytes  )
```

Fills an area of memory with constant data.

Parameters:

DestPtr is the destination address to set.

Data contains the value to set.

Bytes is the number of bytes to set.

Returns:

None.

Note:

Signature: void **XENV_MEM_FILL**(void *DestPtr, char Data, unsigned Bytes)

```
#define XENV_TIME_STAMP_DELTA_MS( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

None.

```
#define XENV_TIME_STAMP_DELTA_US( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

None.

```
#define XENV_TIME_STAMP_GET( StampPtr )
```

Time is derived from the 64 bit PPC timebase register

Parameters:

StampPtr is the storage for the retrieved time stamp.

Returns:

None.

Note:

Signature: void [XENV_TIME_STAMP_GET](#)(XTIME_STAMP *StampPtr)

```
#define XENV_USLEEP( delay )
```

```
XENV\_USLEEP(unsigned delay)
```

Delay the specified number of microseconds.

Parameters:

delay is the number of microseconds to delay.

Returns:

None

common/v1_00_a/src/xenv.h File Reference

Detailed Description

Defines common services that are typically found in a host operating. environment. This include file simply includes an OS specific file based on the compile-time constant BUILD_ENV_*, where * is the name of the target environment.

All services are defined as macros.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	ch	10/24/02	Added XENV_LINUX
1.00a	rmm	04/17/02	First release

```
#include "xenv\_none.h"
```

[Go to the source code of this file.](#)

common/v1_00_a/src/xenv_none.h

[Go to the documentation of this file.](#)

```
00001 /*****
00002 *
00003 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00004 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00005 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00006 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00007 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00008 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00009 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00010 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00011 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00012 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00013 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00014 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00015 *     FOR A PARTICULAR PURPOSE.
00016 *
00017 *     (c) Copyright 2002 Xilinx Inc.
00018 *     All rights reserved.
00019 *
00020 *****/
00021 /*****
00022 /**
00023 *
00024 * @file common/v1_00_a/src/xenv_none.h
00025 *
00026 * Defines common services specified by xenv.h. Some of these services are
00027 * defined as not performing any action. The implementation of these services
00028 * are left to the user.
00029 *
00030 * @note
00031 *
00032 * This file is not intended to be included directly by driver code. Instead,
00033 * the generic xenv.h file is intended to be included by driver code.
00034 *
00035 * <pre>
00036 * MODIFICATION HISTORY:
00037 *
00038 * Ver   Who   Date     Changes
00039 * ----  ---  -
00040 * 1.00a rmm   03/21/02 First release
00041 * </pre>
00042 *
```

```

00043 *
00044 *****/
00045
00046 #ifndef XENV_NONE_H /* prevent circular inclusions */
00047 #define XENV_NONE_H /* by using protection macros */
00048
00049 /***** Include Files *****/
00050
00051 /*****/
00052 /**
00053 *
00054 * Copies a non-overlapping block of memory.
00055 *
00056 * @param DestPtr is the destination address to copy data to.
00057 * @param SrcPtr is the source address to copy data from.
00058 * @param Bytes is the number of bytes to copy.
00059 *
00060 * @return
00061 *
00062 * None.
00063 *
00064 * @note
00065 *
00066 * Signature: void XENV_MEM_COPY(void *DestPtr, void *SrcPtr, unsigned Bytes)
00067 *
00068 *****/
00069 #define XENV_MEM_COPY(DestPtr, SrcPtr, Bytes) \
00070 { \
00071     char *Dest = (char*)(DestPtr); \
00072     char *Src = (char*)(SrcPtr); \
00073     unsigned BytesLeft = (Bytes); \
00074 \
00075     while (BytesLeft--) *Dest++ = *Src++; \
00076 }
00077
00078 /*****/
00079 /**
00080 *
00081 * Fills an area of memory with constant data.
00082 *
00083 * @param DestPtr is the destination address to set.
00084 * @param Data contains the value to set.
00085 * @param Bytes is the number of bytes to set.
00086 *
00087 * @return
00088 *
00089 * None.
00090 *
00091 * @note
00092 *
00093 * Signature: void XENV_MEM_FILL(void *DestPtr, char Data, unsigned Bytes)
00094 *

```



```

00095  *****/
00096 #define XENV_MEM_FILL(DestPtr, Data, Bytes) \
00097 { \
00098     char *Dest = (char*)(DestPtr); \
00099     char c = (Data); \
00100     unsigned BytesLeft = (Bytes); \
00101     \
00102     while (BytesLeft--) *Dest++ = c; \
00103 }
00104
00105 /**
00106  * A structure that contains a time stamp used by other time stamp macros
00107  * defined below. This structure is processor dependent.
00108  */
00109 typedef int XENV_TIME_STAMP;
00110
00111 *****/
00112 /**
00113  *
00114  * Time is derived from the 64 bit PPC timebase register
00115  *
00116  * @param StampPtr is the storage for the retrieved time stamp.
00117  *
00118  * @return
00119  *
00120  * None.
00121  *
00122  * @note
00123  *
00124  * Signature: void XENV_TIME_STAMP_GET(XTIME_STAMP *StampPtr)
00125  *
00126  * @note
00127  *
00128  * This macro must be implemented by the user
00129  *****/
00130 #define XENV_TIME_STAMP_GET(StampPtr)
00131
00132 *****/
00133 /**
00134  *
00135  * This macro is not yet implemented and always returns 0.
00136  *
00137  * @param Stamp1Ptr is the first sampled time stamp.
00138  * @param Stamp2Ptr is the second sampled time stamp.
00139  *
00140  * @return 0
00141  *
00142  * @note
00143  *
00144  * This macro must be implemented by the user
00145  *
00146  *****/

```

```

00147 #define XENV_TIME_STAMP_DELTA_US(Stamp1Ptr, Stamp2Ptr)      (0)
00148
00149 /*****
00150 **
00151 *
00152 * This macro is not yet implemented and always returns 0.
00153 *
00154 * @param   Stamp1Ptr is the first sampled time stamp.
00155 * @param   Stamp2Ptr is the second sampled time stamp.
00156 *
00157 * @return  0
00158 *
00159 * @note
00160 *
00161 * This macro must be implemented by the user
00162 *
00163 *****/
00164 #define XENV_TIME_STAMP_DELTA_MS(Stamp1Ptr, Stamp2Ptr)      (0)
00165
00166 /*****
00167 **
00168 *
00169 * XENV_USLEEP(unsigned delay)
00170 *
00171 * Delay the specified number of microseconds. Not implemented without OS
00172 * support.
00173 *
00174 * @param   delay is the number of microseconds to delay.
00175 *
00176 * @return  None
00177 */
00178 /*****
00179 **
00180 *
00181 *
00182 #endif          /* end of protection macro */

```

Generated on 29 May 2003 for Xilinx Device Drivers

common/v1_00_a/src/xenv_none.h File Reference

Detailed Description

Defines common services specified by [xenv.h](#). Some of these services are defined as not performing any action. The implementation of these services are left to the user.

Note:

This file is not intended to be included directly by driver code. Instead, the generic [xenv.h](#) file is intended to be included by driver code.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	03/21/02	First release

[Go to the source code of this file.](#)

Defines

```
#define XENV\_MEM\_COPY(DestPtr, SrcPtr, Bytes)
#define XENV\_MEM\_FILL(DestPtr, Data, Bytes)
#define XENV\_TIME\_STAMP\_GET(StampPtr)
#define XENV\_TIME\_STAMP\_DELTA\_US(Stamp1Ptr, Stamp2Ptr)
#define XENV\_TIME\_STAMP\_DELTA\_MS(Stamp1Ptr, Stamp2Ptr)
```

```
#define XENV_USLEEP(delay)
```

Typedefs

```
typedef int XENV_TIME_STAMP
```

Define Documentation

```
#define XENV_MEM_COPY( DestPtr,  
                      SrcPtr,  
                      Bytes  )
```

Copies a non-overlapping block of memory.

Parameters:

DestPtr is the destination address to copy data to.

SrcPtr is the source address to copy data from.

Bytes is the number of bytes to copy.

Returns:

None.

Note:

Signature: void **XENV_MEM_COPY**(void *DestPtr, void *SrcPtr, unsigned Bytes)

```
#define XENV_MEM_FILL( DestPtr,  
                      Data,  
                      Bytes  )
```

Fills an area of memory with constant data.

Parameters:

DestPtr is the destination address to set.

Data contains the value to set.

Bytes is the number of bytes to set.

Returns:

None.

Note:

Signature: void **XENV_MEM_FILL**(void *DestPtr, char Data, unsigned Bytes)

```
#define XENV_TIME_STAMP_DELTA_MS( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

This macro must be implemented by the user

```
#define XENV_TIME_STAMP_DELTA_US( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

This macro must be implemented by the user

```
#define XENV_TIME_STAMP_GET( StampPtr )
```

Time is derived from the 64 bit PPC timebase register

Parameters:

StampPtr is the storage for the retrieved time stamp.

Returns:

None.

Note:

Signature: void **XENV_TIME_STAMP_GET**(XTIME_STAMP *StampPtr)

Note:

This macro must be implemented by the user

```
#define XENV_USLEEP( delay )
```

XENV_USLEEP(unsigned delay)

Delay the specified number of microseconds. Not implemented without OS support.

Parameters:

delay is the number of microseconds to delay.

Returns:

None

Typedef Documentation

typedef int XENV_TIME_STAMP

A structure that contains a time stamp used by other time stamp macros defined below. This structure is processor dependent.

Generated on 29 May 2003 for Xilinx Device Drivers

common/v1_00_a/src/xenv.h

[Go to the documentation of this file.](#)

```
00001 /*****
00002 *
00003 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00004 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00005 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00006 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00007 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00008 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00009 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00010 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00011 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00012 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00013 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00014 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00015 *     FOR A PARTICULAR PURPOSE.
00016 *
00017 *     (c) Copyright 2002 Xilinx Inc.
00018 *     All rights reserved.
00019 *
00020 *****/
00021 /*****
00022 /**
00023 *
00024 * @file common/v1_00_a/src/xenv.h
00025 *
00026 * Defines common services that are typically found in a host operating.
00027 * environment. This include file simply includes an OS specific file based
00028 * on the compile-time constant BUILD_ENV_*, where * is the name of the target
00029 * environment.
00030 *
00031 * All services are defined as macros.
00032 *
00033 * <pre>
00034 * MODIFICATION HISTORY:
00035 *
00036 * Ver    Who    Date      Changes
00037 * -----
00038 * 1.00b  ch    10/24/02  Added XENV_LINUX
00039 * 1.00a  rmm    04/17/02  First release
00040 * </pre>
00041 *
00042 *****/
```



```
00043
00044 #ifndef XENV_H /* prevent circular inclusions */
00045 #define XENV_H /* by using protection macros */
00046
00047 /*
00048  * Select which target environment we are operating under
00049  */
00050
00051 /* VxWorks target environment */
00052 #if defined XENV_VXWORKS
00053 #include "xenv_vxworks.h"
00054
00055 /* Linux target environment */
00056 #elif defined XENV_LINUX
00057 #include "xenv_linux.h"
00058
00059 /* Unit test environment */
00060 #elif defined XENV_UNITTEST
00061 #include "ut_xenv.h"
00062
00063 /* Integration test environment */
00064 #elif defined XENV_INTTEST
00065 #include "int_xenv.h"
00066
00067 /* No environment selected */
00068 #else
00069 #include "xenv_none.h"
00070 #endif
00071
00072
00073 /*
00074  * The following comments specify the types and macro wrappers that are
00075  * expected to be defined by the target specific header files
00076  */
00077
00078 /***** Type Definitions *****/
00079
00080 /*****
00081  **
00082  *
00083  * XENV_TIME_STAMP
00084  *
00085  * A structure that contains a time stamp used by other time stamp macros
00086  * defined below. This structure is processor dependent.
00087  */
00088
00089
00090 /***** Macros (Inline Functions) Definitions *****/
00091
00092 /*****
00093  **
00094  *
```

```
00095 * XENV_MEM_COPY(void *DestPtr, void *SrcPtr, unsigned Bytes)
00096 *
00097 * Copies a non-overlapping block of memory.
00098 *
00099 * @param DestPtr is the destination address to copy data to.
00100 * @param SrcPtr is the source address to copy data from.
00101 * @param Bytes is the number of bytes to copy.
00102 *
00103 * @return None
00104 */
00105
00106 /*****
00107 **
00108 *
00109 * XENV_MEM_FILL(void *DestPtr, char Data, unsigned Bytes)
00110 *
00111 * Fills an area of memory with constant data.
00112 *
00113 * @param DestPtr is the destination address to set.
00114 * @param Data contains the value to set.
00115 * @param Bytes is the number of bytes to set.
00116 *
00117 * @return None
00118 */
00119 /*****
00120 **
00121 *
00122 * XENV_TIME_STAMP_GET(XTIME_STAMP *StampPtr)
00123 *
00124 * Samples the processor's or external timer's time base counter.
00125 *
00126 * @param StampPtr is the storage for the retrieved time stamp.
00127 *
00128 * @return None
00129 */
00130
00131 /*****
00132 **
00133 *
00134 * XENV_TIME_STAMP_DELTA_US(XTIME_STAMP *Stamp1Ptr, XTIME_STAMP* Stamp2Ptr)
00135 *
00136 * Computes the delta between the two time stamps.
00137 *
00138 * @param Stamp1Ptr - First sampled time stamp.
00139 * @param Stamp1Ptr - Sedond sampled time stamp.
00140 *
00141 * @return An unsigned int value with units of microseconds.
00142 */
00143
00144 /*****
00145 **
00146 *
00147 * XENV_TIME_STAMP_DELTA_MS(XTIME_STAMP *Stamp1Ptr, XTIME_STAMP* Stamp2Ptr)
```

```
00148 *
00149 * Computes the delta between the two time stamps.
00150 *
00151 * @param Stamp1Ptr - First sampled time stamp.
00152 * @param Stamp1Ptr - Sedond sampled time stamp.
00153 *
00154 * @return An unsigned int value with units of milliseconds.
00155 */
00156
00157
00158 *
00159 * XENV_USLEEP(unsigned delay)
00160 *
00161 * Delay the specified number of microseconds.
00162 *
00163 * @param delay is the number of microseconds to delay.
00164 *
00165 * @return None
00166 */
00167
00168 #endif /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

common/v1_00_a/src/xenv_vxworks.h

[Go to the documentation of this file.](#)

```
00001 /*****
00002 *
00003 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00004 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00005 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00006 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00007 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00008 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00009 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00010 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00011 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00012 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00013 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00014 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00015 *     FOR A PARTICULAR PURPOSE.
00016 *
00017 *     (c) Copyright 2002 Xilinx Inc.
00018 *     All rights reserved.
00019 *
00020 *****/
00021 /*****
00022 /**
00023 *
00024 * @file common/v1_00_a/src/xenv_vxworks.h
00025 *
00026 * Defines common services that are typically found in a VxWorks target
00027 * environment.
00028 *
00029 * @note
00030 *
00031 * This file is not intended to be included directly by driver code. Instead,
00032 * the generic xenv.h file is intended to be included by driver code.
00033 *
00034 * <pre>
00035 * MODIFICATION HISTORY:
00036 *
00037 * Ver   Who   Date     Changes
00038 * -----
00039 * 1.00b rmm   10/24/02 Added XENV_USLEEP macro
00040 * 1.00a rmm   07/16/01 First release
00041 * </pre>
00042 *
```

```

00043 *
00044 *****/
00045
00046 #ifndef XENV_VXWORKS_H /* prevent circular inclusions */
00047 #define XENV_VXWORKS_H /* by using protection macros */
00048
00049 /***** Include Files *****/
00050 #include "xbasic_types.h"
00051 #include "vxWorks.h"
00052 #include "vxLib.h"
00053 #include <string.h>
00054
00055 /*****
00056 /**
00057 *
00058 * Copies a non-overlapping block of memory.
00059 *
00060 * @param DestPtr is the destination address to copy data to.
00061 * @param SrcPtr is the source address to copy data from.
00062 * @param Bytes is the number of bytes to copy.
00063 *
00064 * @return
00065 *
00066 * None.
00067 *
00068 * @note
00069 *
00070 * Signature: void XENV_MEM_COPY(void *DestPtr, void *SrcPtr, unsigned Bytes)
00071 *
00072 *****/
00073 #define XENV_MEM_COPY(DestPtr, SrcPtr, Bytes) \
00074 { \
00075     void *Dest = (DestPtr); \
00076     void *Src = (SrcPtr); \
00077     unsigned LengthBytes = (Bytes); \
00078     \
00079     memcpy(Dest, Src, (size_t)LengthBytes); \
00080 }
00081
00082 /*****
00083 /**
00084 *
00085 * Fills an area of memory with constant data.
00086 *
00087 * @param DestPtr is the destination address to set.
00088 * @param Data contains the value to set.
00089 * @param Bytes is the number of bytes to set.
00090 *
00091 * @return
00092 *
00093 * None.
00094 *

```

```

00095 * @note
00096 *
00097 * Signature: void XENV_MEM_FILL(void *DestPtr, char Data, unsigned Bytes)
00098 *
00099 *****/
00100 #define XENV_MEM_FILL(DestPtr, Data, Bytes) \
00101 { \
00102     void *Dest = (DestPtr); \
00103     char c = (Data); \
00104     unsigned LengthBytes = (Bytes); \
00105 \
00106     memset(Dest, (int)Data, (size_t)LengthBytes); \
00107 }
00108
00109 #if (CPU_FAMILY==PPC)
00110 /**
00111 * A structure that contains a time stamp used by other time stamp macros
00112 * defined below. This structure is processor dependent.
00113 */
00114 typedef struct
00115 {
00116     Xuint32 TimeBaseUpper;
00117     Xuint32 TimeBaseLower;
00118 } XENV_TIME_STAMP;
00119 #endif
00120
00121 *****/
00122 /**
00123 *
00124 * Time is derived from the 64 bit PPC timebase register
00125 *
00126 * @param StampPtr is the storage for the retrieved time stamp.
00127 *
00128 * @return
00129 *
00130 * None.
00131 *
00132 * @note
00133 *
00134 * Signature: void XENV_TIME_STAMP_GET(XTIME_STAMP *StampPtr)
00135 *
00136 *****/
00137 #define XENV_TIME_STAMP_GET(StampPtr) \
00138 { \
00139     vxTimeBaseGet((UINT32*)&(StampPtr)->TimeBaseUpper, \
00140                 (UINT32*)&(StampPtr)->TimeBaseLower); \
00141 }
00142
00143 *****/
00144 /**
00145 *
00146 * This macro is not yet implemented and always returns 0.

```

```

00147 *
00148 * @param Stamp1Ptr is the first sampled time stamp.
00149 * @param Stamp2Ptr is the second sampled time stamp.
00150 *
00151 * @return 0
00152 *
00153 * @note
00154 *
00155 * None.
00156 *
00157 *****/
00158 #define XENV_TIME_STAMP_DELTA_US(Stamp1Ptr, Stamp2Ptr) (0)
00159
00160 *****/
00161 /**
00162 *
00163 * This macro is not yet implemented and always returns 0.
00164 *
00165 * @param Stamp1Ptr is the first sampled time stamp.
00166 * @param Stamp2Ptr is the second sampled time stamp.
00167 *
00168 * @return 0
00169 *
00170 * @note
00171 *
00172 * None.
00173 *
00174 *****/
00175 #define XENV_TIME_STAMP_DELTA_MS(Stamp1Ptr, Stamp2Ptr) (0)
00176
00177 *****/
00178 /**
00179 *
00180 * XENV_USLEEP(unsigned delay)
00181 *
00182 * Delay the specified number of microseconds.
00183 *
00184 * @param delay is the number of microseconds to delay.
00185 *
00186 * @return None
00187 *
00188 *****/
00189 #define XENV_USLEEP(delay) sysUsDelay(delay)
00190
00191
00192
00193 #endif /* end of protection macro */

```

XENV_TIME_STAMP Struct Reference

```
#include <xenv\_vxworks.h>
```

Detailed Description

A structure that contains a time stamp used by other time stamp macros defined below. This structure is processor dependent.

The documentation for this struct was generated from the following file:

- `common/v1_00_a/src/xenv_vxworks.h`
-

common/v1_00_a/src/xenv_linux.h File Reference

Detailed Description

Defines common services specified by [xenv.h](#). Some of these services are defined as not performing any action. The implementation of these services are left to the user.

Note:

This file is not intended to be included directly by driver code. Instead, the generic [xenv.h](#) file is intended to be included by driver code.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ch	10/24/02	First release

```
#include "sleep.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XENV\_MEM\_COPY(DestPtr, SrcPtr, Bytes)  
#define XENV\_MEM\_FILL(DestPtr, Data, Bytes)  
#define XENV\_TIME\_STAMP\_GET(StampPtr)
```


Fills an area of memory with constant data.

Parameters:

DestPtr is the destination address to set.

Data contains the value to set.

Bytes is the number of bytes to set.

Returns:

None.

Note:

Signature: void **XENV_MEM_FILL**(void *DestPtr, char Data, unsigned Bytes)

```
#define XENV_TIME_STAMP_DELTA_MS( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

This macro must be implemented by the user

```
#define XENV_TIME_STAMP_DELTA_US( Stamp1Ptr,  
                                Stamp2Ptr )
```

This macro is not yet implemented and always returns 0.

Parameters:

Stamp1Ptr is the first sampled time stamp.

Stamp2Ptr is the second sampled time stamp.

Returns:

0

Note:

This macro must be implemented by the user

```
#define XENV_TIME_STAMP_GET( StampPtr )
```

Time is derived from the 64 bit PPC timebase register

Parameters:

StampPtr is the storage for the retrieved time stamp.

Returns:

None.

Note:

Signature: void [XENV_TIME_STAMP_GET](#)(XTIME_STAMP *StampPtr)

Note:

This macro must be implemented by the user

```
#define XENV_USLEEP( delay )
```

XENV_USLEEP(unsigned delay)

Delay the specified number of microseconds.

Parameters:

delay is the number of microseconds to delay.

Returns:

None

Typedef Documentation

typedef int XENV_TIME_STAMP

A structure that contains a time stamp used by other time stamp macros defined below. This structure is processor dependent.

Generated on 29 May 2003 for Xilinx Device Drivers

common/v1_00_a/src/xenv_linux.h

[Go to the documentation of this file.](#)

```
00001 /*****
00002 *
00003 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00004 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00005 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00006 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00007 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00008 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00009 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00010 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00011 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00012 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00013 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00014 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00015 *     FOR A PARTICULAR PURPOSE.
00016 *
00017 *     (c) Copyright 2002 Xilinx Inc.
00018 *     All rights reserved.
00019 *
00020 *****/
00021 /*****
00022 /**
00023 *
00024 * @file common/v1_00_a/src/xenv_linux.h
00025 *
00026 * Defines common services specified by xenv.h. Some of these services are
00027 * defined as not performing any action. The implementation of these services
00028 * are left to the user.
00029 *
00030 * @note
00031 *
00032 * This file is not intended to be included directly by driver code. Instead,
00033 * the generic xenv.h file is intended to be included by driver code.
00034 *
00035 * <pre>
00036 * MODIFICATION HISTORY:
00037 *
00038 * Ver    Who    Date      Changes
00039 * -----
00040 * 1.00a  ch     10/24/02  First release
00041 * </pre>
00042 *
```

```

00043 *
00044 *****/
00045
00046 #ifndef XENV_LINUX_H /* prevent circular inclusions */
00047 #define XENV_LINUX_H /* by using protection macros */
00048
00049 /*** Include Files *****/
00050 #include "sleep.h"
00051
00052 /***
00053 /**
00054 *
00055 * Copies a non-overlapping block of memory.
00056 *
00057 * @param DestPtr is the destination address to copy data to.
00058 * @param SrcPtr is the source address to copy data from.
00059 * @param Bytes is the number of bytes to copy.
00060 *
00061 * @return
00062 *
00063 * None.
00064 *
00065 * @note
00066 *
00067 * Signature: void XENV_MEM_COPY(void *DestPtr, void *SrcPtr, unsigned Bytes)
00068 *
00069 *****/
00070 #define XENV_MEM_COPY(DestPtr, SrcPtr, Bytes) \
00071 { \
00072     char *Dest = (char*)(DestPtr); \
00073     char *Src = (char*)(SrcPtr); \
00074     unsigned BytesLeft = (Bytes); \
00075 \
00076     while (BytesLeft--) *Dest++ = *Src++; \
00077 }
00078
00079 /***
00080 /**
00081 *
00082 * Fills an area of memory with constant data.
00083 *
00084 * @param DestPtr is the destination address to set.
00085 * @param Data contains the value to set.
00086 * @param Bytes is the number of bytes to set.
00087 *
00088 * @return
00089 *
00090 * None.
00091 *
00092 * @note
00093 *
00094 * Signature: void XENV_MEM_FILL(void *DestPtr, char Data, unsigned Bytes)

```

```

00095  *
00096  *****/
00097 #define XENV_MEM_FILL(DestPtr, Data, Bytes) \
00098 { \
00099     char *Dest = (char*)(DestPtr); \
00100     char c = (Data); \
00101     unsigned BytesLeft = (Bytes); \
00102     \
00103     while (BytesLeft--) *Dest++ = c; \
00104 }
00105
00106 /**
00107  * A structure that contains a time stamp used by other time stamp macros
00108  * defined below. This structure is processor dependent.
00109  */
00110 typedef int XENV_TIME_STAMP;
00111
00112 *****/
00113 /**
00114  *
00115  * Time is derived from the 64 bit PPC timebase register
00116  *
00117  * @param   StampPtr is the storage for the retrieved time stamp.
00118  *
00119  * @return
00120  *
00121  * None.
00122  *
00123  * @note
00124  *
00125  * Signature: void XENV_TIME_STAMP_GET(XTIME_STAMP *StampPtr)
00126  *
00127  * @note
00128  *
00129  * This macro must be implemented by the user
00130  *****/
00131 #define XENV_TIME_STAMP_GET(StampPtr)
00132
00133 *****/
00134 /**
00135  *
00136  * This macro is not yet implemented and always returns 0.
00137  *
00138  * @param   Stamp1Ptr is the first sampled time stamp.
00139  * @param   Stamp2Ptr is the second sampled time stamp.
00140  *
00141  * @return  0
00142  *
00143  * @note
00144  *
00145  * This macro must be implemented by the user
00146  *

```



```

00147  *****/
00148 #define XENV_TIME_STAMP_DELTA_US(Stamp1Ptr, Stamp2Ptr)      (0)
00149
00150  *****/
00151  /**
00152  *
00153  * This macro is not yet implemented and always returns 0.
00154  *
00155  * @param   Stamp1Ptr is the first sampled time stamp.
00156  * @param   Stamp2Ptr is the second sampled time stamp.
00157  *
00158  * @return  0
00159  *
00160  * @note
00161  *
00162  * This macro must be implemented by the user
00163  *
00164  *****/
00165 #define XENV_TIME_STAMP_DELTA_MS(Stamp1Ptr, Stamp2Ptr)      (0)
00166
00167  *****/
00168  /**
00169  *
00170  * XENV_USLEEP(unsigned delay)
00171  *
00172  * Delay the specified number of microseconds.
00173  *
00174  * @param   delay is the number of microseconds to delay.
00175  *
00176  * @return  None
00177  *
00178  *****/
00179 #define XENV_USLEEP(delay)                                     \
00180 {                                                             \
00181     usleep(delay);                                          \
00182 }
00183
00184
00185
00186 #endif                /* end of protection macro */

```

flash/v1_00_a/src/xflash_cfi.h File Reference

Detailed Description

This is a helper component for XFlash. It contains methods used to extract and interpret Common Flash Interface (CFI) from a flash memory part that supports the CFI query command.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xflash.h"  
#include "xflash_geometry.h"  
#include "xflash_properties.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XFL_CFI_POSITION_PTR(Ptr, BaseAddr, Interleave, ByteAddr)  
#define XFL_CFI_READ8(Ptr, Interleave)  
#define XFL_CFI_READ16(Ptr, Interleave, Data)
```

```
#define XFL_CFI_ADVANCE_PTR8(Ptr, Interleave)
#define XFL_CFI_ADVANCE_PTR16(Ptr, Interleave)
```

Functions

XStatus XFlashCFI_ReadCommon (**XFlashGeometry** *GeometryPtr, **XFlashProperties** *PropertiesPtr)

Define Documentation

```
#define XFL_CFI_ADVANCE_PTR16( Ptr,
                               Interleave )
```

Advances the CFI pointer to the next 16-bit quantity.

Parameters:

Ptr is the pointer to advance. Can be a pointer to any type.
Interleave is the byte interleaving (based on part layout)

Returns:

Adjusted Ptr.

```
#define XFL_CFI_ADVANCE_PTR8( Ptr,
                               Interleave )
```

Advances the CFI pointer to the next byte

Parameters:

Ptr is the pointer to advance. Can be a pointer to any type.
Interleave is the byte interleaving (based on part layout)

Returns:

Adjusted Ptr.

```
#define XFL_CFI_POSITION_PTR( Ptr,  
                             BaseAddr,  
                             Interleave,  
                             ByteAddr )
```

Moves the CFI data pointer to a physical address that corresponds to a specific CFI byte offset.

Parameters:

Ptr is the pointer to modify. Can be of any type
BaseAddr is the base address of flash part
Interleave is the byte interleaving (based on part layout)
ByteAddr is the byte offset within CFI data to read

Returns:

The *Ptr* argument is set to point at the the CFI byte specified by the *ByteAddr* parameter.

```
#define XFL_CFI_READ16( Ptr,  
                      Interleave,  
                      Data )
```

Reads 16-bits of data from the CFI data location into a local variable.

Parameters:

Ptr is the pointer to read. Can be a pointer to any type.
Interleave is the byte interleaving (based on part layout)
Data is the 16-bit storage location for the data to be read.

Returns:

The 16-bit value at *Ptr* adjusted for the interleave factor.

```
#define XFL_CFI_READ8( Ptr,  
                    Interleave )
```

Reads 8-bits of data from the CFI data location into a local variable.

Parameters:

Ptr is the pointer to read. Can be a pointer to any type.

Interleave is the byte interleaving (based on part layout)

Returns:

The byte at *Ptr* adjusted for the interleave factor.

Function Documentation

```
XStatus XFlashCFI_ReadCommon( XFlashGeometry * GeometryPtr,  
                              XFlashProperties * PropertiesPtr  
                              )
```

Retrieves the standard CFI data from the part(s), interpret the data, and update the provided geometry and properties structures.

Extended CFI data is part specific and ignored here. This data must be read by the specific part component driver.

Parameters:

GeometryPtr is an input/output parameter. This function expects the BaseAddress and MemoryLayout attributes to be correctly initialized. All other attributes of this structure will be setup using translated CFI data read from the part.

PropertiesPtr is an output parameter. Timing, identification, and programming CFI data will be translated and written to this structure.

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_CFI_QUERY_ERROR if an error occurred interpreting the data.
- o XST_FLASH_PART_NOT_SUPPORTED if invalid Layout parameter

Note:

None.

flash/v1_00_a/src/xflash.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xflash.h,v 1.8 2002/05/13 19:57:06 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file flash/v1_00_a/src/xflash.h
00026 *
00027 * This is the base component for XFlash. It provides the public
00028 * interface which upper layers use to communicate with specific flash
00029 * hardware.
00030 *
00031 * This driver supports "Common Flash Interface" (CFI) enabled flash hardware.
00032 * CFI allows entire families of flash parts to be supported with a single
00033 * driver sub-component.
00034 *
00035 * This driver is designed for devices external to the FPGA. As a result
00036 * interfaces such as IPIF and versioning are not incorporated into its
00037 * design. A more detailed description of the driver operation can be found
00038 * in XFlash.c
00039 *
00040 * @note
00041 *
00042 * This driver is intended to be RTOS and processor independent. It
```

```

00043 * works with physical addresses only. Any needs for dynamic memory
00044 * management, threads, mutual exclusion, virtual memory, cache
00045 * control, or HW write protection management must be satisfied by the
00046 * layer above this driver.
00047 * <br><br>
00048 * All writes to flash occur in units of bus-width bytes. If more than one
00049 * part exists on the data bus, then the parts are written in parallel.
00050 * Reads from flash are performed in any width up to the width of the data
00051 * bus. It is assumed that the flash bus controller or local bus supports
00052 * these types of accesses.
00053 *
00054 * <pre>
00055 * MODIFICATION HISTORY:
00056 *
00057 * Ver    Who    Date      Changes
00058 * -----
00059 * 1.00a rmm    07/16/01 First release
00060 * </pre>
00061 *
00062 *****/
00063
00064 #ifndef XFLASH_H /* prevent circular inclusions */
00065 #define XFLASH_H /* by using protection macros */
00066
00067 /***** Include Files *****/
00068 #include "xbasic_types.h"
00069 #include "xstatus.h"
00070 #include "xflash_geometry.h"
00071 #include "xflash_properties.h"
00072
00073 /***** Constant Definitions *****/
00074
00075 /**
00076  * Supported manufacturer IDs. Note, that not all parts from these listed
00077  * vendors are supported.
00078  */
00079 #define XFL_MANUFACTURER_ID_INTEL 0x89
00080
00081 /** @name Configuration options
00082  * @{
00083  */
00084 /**
00085  * <pre>
00086  * XFL_NON_BLOCKING_ERASE_OPTION    Controls whether the interface blocks on
00087  *                                  device erase until the operation is
00088  *                                  completed 1=noblock, 0=block
00089  * XFL_NON_BLOCKING_WRITE_OPTION   Controls whether the interface blocks on
00090  *                                  device program until the operation is
00091  *                                  completed 1=noblock, 0=block
00092  * </pre>
00093  */
00094 #define XFL_NON_BLOCKING_ERASE_OPTION 0x00000001UL

```

```

00095 #define XFL_NON_BLOCKING_WRITE_OPTION 0x00000002UL
00096 /*@}*/
00097
00098 #define XFL_MAX_VENDOR_DATA_LENGTH 20          /* Number of 32-bit integers
00099                                             reserved for vendor data as part
00100                                             of the XFlash sub-components */
00101
00102 /***** Type Definitions *****/
00103
00104
00105 /*
00106  * Define a type to contain part specific data to be maintained by
00107  * the part component. Within the component, a new type is defined that
00108  * overlays this type.
00109  */
00110 typedef Xuint32 XFlashVendorData[XFL_MAX_VENDOR_DATA_LENGTH];
00111
00112 /**
00113  * This typedef contains configuration information for the device.
00114  */
00115 typedef struct
00116 {
00117     Xuint16 DeviceId;          /**< Unique ID of device */
00118     Xuint32 BaseAddr;         /**< Base address of array */
00119     Xuint8  NumParts;         /**< Number of parts in the array */
00120     Xuint8  PartWidth;       /**< Width of each part in bytes */
00121     Xuint8  PartMode;        /**< Operation mode of each part in bytes */
00122 } XFlash_Config;
00123
00124 /**
00125  * The XFlash driver instance data. The user is required to allocate a
00126  * variable of this type for every flash device in the system. A pointer
00127  * to a variable of this type is then passed to the driver API functions.
00128  */
00129 typedef struct XFlashTag
00130 {
00131     Xuint32  Options;          /* Current device options */
00132     Xboolean IsReady;         /* Device is initialized and ready */
00133     XFlashGeometry Geometry;  /* Part geometry */
00134     XFlashProperties Properties; /* Part timing, programming, &
00135                                     identification properties */
00136     XFlashVendorData VendorData; /* Part specific data */
00137
00138     struct
00139     {
00140         XStatus (*ReadBlock)(struct XFlashTag *InstancePtr, Xuint16 Region,
00141                               Xuint16 Block, Xuint32 Offset, Xuint32 Bytes,
00142                               void* DestPtr);
00143
00144         XStatus (*WriteBlock)(struct XFlashTag *InstancePtr, Xuint16 Region,
00145                                Xuint16 Block, Xuint32 Offset, Xuint32 Bytes,

```



```
00146         void* SrcPtr);
00147
00148     XStatus (*WriteBlockSuspend)(struct XFlashTag *InstancePtr,
00149                                 Xuint16 Region, Xuint16 Block,
00150                                 Xuint32 Offset);
00151
00152     XStatus (*WriteBlockResume)(struct XFlashTag *InstancePtr,
00153                                 Xuint16 Region, Xuint16 Block,
00154                                 Xuint32 Offset);
00155
00156     XStatus (*EraseBlock)(struct XFlashTag *InstancePtr, Xuint16 Region,
00157                           Xuint16 Block, Xuint16 NumBlocks);
00158
00159     XStatus (*EraseBlockSuspend)(struct XFlashTag *InstancePtr,
00160                                  Xuint16 Region, Xuint16 Block);
00161
00162     XStatus (*EraseBlockResume)(struct XFlashTag *InstancePtr,
00163                                  Xuint16 Region, Xuint16 Block);
00164
00165     XStatus (*LockBlock)(struct XFlashTag *InstancePtr, Xuint16 Region,
00166                          Xuint16 Block, Xuint16 NumBlocks);
00167
00168     XStatus (*UnlockBlock)(struct XFlashTag *InstancePtr, Xuint16 Region,
00169                            Xuint16 Block, Xuint16 NumBlocks);
00170
00171     XStatus (*GetBlockStatus)(struct XFlashTag *InstancePtr, Xuint16
Region,
00172                               Xuint16 Block);
00173
00174     XStatus (*Read)(struct XFlashTag *InstancePtr, Xuint32 Offset,
00175                    Xuint32 Bytes, void* DestPtr);
00176
00177     XStatus (*Write)(struct XFlashTag *InstancePtr, Xuint32 Offset,
00178                     Xuint32 Bytes, void* SrcPtr);
00179
00180     XStatus (*WriteSuspend)(struct XFlashTag *InstancePtr, Xuint32 Offset);
00181     XStatus (*WriteResume)(struct XFlashTag *InstancePtr, Xuint32 Offset);
00182
00183     XStatus (*Erase)(struct XFlashTag *InstancePtr, Xuint32 Offset,
00184                     Xuint32 Bytes);
00185
00186     XStatus (*EraseSuspend)(struct XFlashTag *InstancePtr, Xuint32 Offset);
00187     XStatus (*EraseResume)(struct XFlashTag *InstancePtr, Xuint32 Offset);
00188
00189     XStatus (*Lock)(struct XFlashTag *InstancePtr, Xuint32 Offset,
00190                    Xuint32 Bytes);
00191
00192     XStatus (*Unlock)(struct XFlashTag *InstancePtr, Xuint32 Offset,
00193                    Xuint32 Bytes);
00194     XStatus (*GetStatus)(struct XFlashTag *InstancePtr, Xuint32 Offset);
```

```

00195
00196     XStatus (*EraseChip)(struct XFlashTag *InstancePtr);
00197     XStatus (*Initialize)(struct XFlashTag *Initialize);
00198     XStatus (*SelfTest)(struct XFlashTag *InstancePtr);
00199     XStatus (*Reset)(struct XFlashTag *InstancePtr);
00200     XStatus (*SetOptions)(struct XFlashTag *InstancePtr,
00201                          Xuint32 OptionsFlag);
00202
00203     Xuint32 (*GetOptions)(struct XFlashTag *InstancePtr);
00204     XFlashProperties* (*GetProperties)(struct XFlashTag *InstancePtr);
00205     XFlashGeometry* (*GetGeometry)(struct XFlashTag *InstancePtr);
00206     XStatus (*DeviceControl)(struct XFlashTag *InstancePtr, Xuint32
Command,
00207                             Xuint32 Param, Xuint32 *ReturnPtr);
00208     } VTable;
00209 } XFlash;
00210
00211
00212 /***** Macros (Inline Functions) Definitions *****/
00213
00214 /*
00215  * The following macros implement flash I/O primitives. All flash components
00216  * use these macros to access the flash devices. The only exceptions are read
00217  * functions which simply copy data using memcpy or equivalent utility funcs.
00218  *
00219  * The 8, 16, 32, 64 signify the access width.
00220  */
00221 #define READ_FLASH_8(Address) \
00222     (*(volatile Xuint8*)(Address))
00223
00224 #define READ_FLASH_16(Address) \
00225     (*(volatile Xuint16*)(Address))
00226
00227 #define READ_FLASH_32(Address) \
00228     (*(volatile Xuint32*)(Address))
00229
00230 #define READ_FLASH_64(Address, Data) \
00231     (XUINT64_MSW(Data) = *(volatile Xuint32*)(Address)); \
00232     (XUINT64_LSW(Data) = *(volatile Xuint32*)((Xuint32)(Address) + 4))
00233
00234 #define WRITE_FLASH_8(Address, Data) \
00235     (*(volatile Xuint8*)(Address) = (Xuint8)(Data))
00236
00237 #define WRITE_FLASH_16(Address, Data) \
00238     (*(volatile Xuint16*)(Address) = (Xuint16)(Data))
00239
00240 #define WRITE_FLASH_32(Address, Data) \
00241     (*(volatile Xuint32*)(Address) = (Xuint32)(Data))
00242
00243 #define WRITE_FLASH_64(Address, Data) \
00244     (*(volatile Xuint32*)(Address) = XUINT64_MSW(Data)); \

```

```

00245     (*(volatile Xuint32*)((Xuint32)(Address) + 4) = XUINT64_LSW(Data))
00246
00247 #define WRITE_FLASH_64x2(Address, Data1, Data2) \
00248     (*(volatile Xuint32*)(Address) = Data1); \
00249     (*(volatile Xuint32*)((Xuint32)(Address) + 4) = Data2)
00250
00251 /***** Function Prototypes *****/
00252
00253 /*
00254  * Initialization, configuration, & control Functions
00255  */
00256 XStatus XFlash_Initialize(XFlash *InstancePtr, Xuint16 DeviceId);
00257 XStatus XFlash_SelfTest(XFlash *InstancePtr);
00258 XStatus XFlash_Reset(XFlash *InstancePtr);
00259 XFlash_Config *XFlash_LookupConfig(Xuint16 DeviceId);
00260 XStatus XFlash_SetOptions(XFlash *InstancePtr, Xuint32 OptionsFlag);
00261 Xuint32 XFlash_GetOptions(XFlash *InstancePtr);
00262 XFlashProperties* XFlash_GetProperties(XFlash *InstancePtr);
00263 XFlashGeometry* XFlash_GetGeometry(XFlash *InstancePtr);
00264
00265 XStatus XFlash_DeviceControl(XFlash *InstancePtr, Xuint32 Command,
00266                             Xuint32 Param, Xuint32 *ReturnPtr);
00267
00268 /*
00269  * Non-geometry aware API
00270  */
00271 XStatus XFlash_Read(XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes,
00272                   void* DestPtr);
00273
00274 XStatus XFlash_Write(XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes,
00275                    void* SrcPtr);
00276
00277 XStatus XFlash_WriteSuspend(XFlash *InstancePtr, Xuint32 Offset);
00278 XStatus XFlash_WriteResume(XFlash *InstancePtr, Xuint32 Offset);
00279
00280 XStatus XFlash_Erase(XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes);
00281 XStatus XFlash_EraseSuspend(XFlash *InstancePtr, Xuint32 Offset);
00282 XStatus XFlash_EraseResume(XFlash *InstancePtr, Xuint32 Offset);
00283
00284 XStatus XFlash_Lock(XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes);
00285 XStatus XFlash_Unlock(XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes);
00286 XStatus XFlash_GetStatus(XFlash *InstancePtr, Xuint32 Offset);
00287
00288 /*
00289  * Geometry aware API
00290  */
00291 XStatus XFlash_ReadBlock(XFlash *InstancePtr, Xuint16 Region, Xuint16 Block,
00292                        Xuint32 Offset, Xuint32 Bytes, void* DestPtr);
00293
00294 XStatus XFlash_WriteBlock(XFlash *InstancePtr, Xuint16 Region, Xuint16 Block,

```

```
00295             Xuint32 Offset, Xuint32 Bytes, void* SrcPtr);
00296
00297 XStatus XFlash_WriteBlockSuspend(XFlash *InstancePtr, Xuint16 Region,
00298             Xuint16 Block, Xuint32 Offset);
00299
00300 XStatus XFlash_WriteBlockResume(XFlash *InstancePtr, Xuint16 Region,
00301             Xuint16 Block, Xuint32 Offset);
00302
00303 XStatus XFlash_EraseBlock(XFlash *InstancePtr, Xuint16 Region, Xuint16 Block,
00304             Xuint16 NumBlocks);
00305
00306 XStatus XFlash_EraseBlockSuspend(XFlash *InstancePtr, Xuint16 Region,
00307             Xuint16 Block);
00308
00309 XStatus XFlash_EraseBlockResume(XFlash *InstancePtr, Xuint16 Region,
00310             Xuint16 Block);
00311
00312 XStatus XFlash_LockBlock(XFlash *InstancePtr, Xuint16 Region, Xuint16 Block,
00313             Xuint16 NumBlocks);
00314
00315 XStatus XFlash_UnlockBlock(XFlash *InstancePtr, Xuint16 Region, Xuint16 Block,
00316             Xuint16 NumBlocks);
00317
00318 XStatus XFlash_GetBlockStatus(XFlash *InstancePtr, Xuint16 Region,
00319             Xuint16 Block);
00320
00321 /*
00322  * Other commands
00323  */
00324 XStatus XFlash_EraseChip(XFlash *InstancePtr);
00325
00326 #endif             /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

flash/v1_00_a/src/xflash.h File Reference

Detailed Description

This is the base component for XFlash. It provides the public interface which upper layers use to communicate with specific flash hardware.

This driver supports "Common Flash Interface" (CFI) enabled flash hardware. CFI allows entire families of flash parts to be supported with a single driver sub-component.

This driver is designed for devices external to the FPGA. As a result interfaces such as IPIF and versioning are not incorporated into its design. A more detailed description of the driver operation can be found in [XFlash.c](#)

Note:

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, cache control, or HW write protection management must be satisfied by the layer above this driver.

All writes to flash occur in units of bus-width bytes. If more than one part exists on the data bus, then the parts are written in parallel. Reads from flash are performed in any width up to the width of the data bus. It is assumed that the flash bus controller or local bus supports these types of accesses.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xflash_geometry.h"  
#include "xflash_properties.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XFlash_Config  
struct XFlashTag
```

Configuration options

```
#define XFL_NON_BLOCKING_ERASE_OPTION  
#define XFL_NON_BLOCKING_WRITE_OPTION
```

Defines

```
#define XFL_MANUFACTURER_ID_INTEL
```

Typedefs

```
typedef XFlashTag XFlash
```

Functions

```
XStatus XFlash_Initialize (XFlash *InstancePtr, Xuint16 DeviceId)
```

```
XStatus XFlash_SelfTest (XFlash *InstancePtr)
```

```
XStatus XFlash_Reset (XFlash *InstancePtr)
```

```
XFlash_Config * XFlash_LookupConfig (Xuint16 DeviceId)
```

```
XStatus XFlash_SetOptions (XFlash *InstancePtr, Xuint32 OptionsFlag)
```

```
Xuint32 XFlash_GetOptions (XFlash *InstancePtr)
```

```
XFlashProperties * XFlash_GetProperties (XFlash *InstancePtr)
```

```
XFlashGeometry * XFlash_GetGeometry (XFlash *InstancePtr)
```

```
XStatus XFlash_DeviceControl (XFlash *InstancePtr, Xuint32 Command, Xuint32 Param,  
    Xuint32 *ReturnPtr)
```

```
XStatus XFlash_Read (XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes, void *DestPtr)
```

```
XStatus XFlash_Write (XFlash *InstancePtr, Xuint32 Offset, Xuint32 Bytes, void *SrcPtr)
```

```
XStatus XFlash_WriteSuspend (XFlash *InstancePtr, Xuint32 Offset)
```

```
XStatus XFlash_WriteResume (XFlash *InstancePtr, Xuint32 Offset)
```

XStatus XFlash_Erase (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlash_EraseSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlash_EraseResume (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlash_Lock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlash_Unlock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlash_GetStatus (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlash_ReadBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)
XStatus XFlash_WriteBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)
XStatus XFlash_WriteBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)
XStatus XFlash_WriteBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)
XStatus XFlash_EraseBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)
XStatus XFlash_EraseBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)
XStatus XFlash_EraseBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)
XStatus XFlash_LockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)
XStatus XFlash_UnlockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)
XStatus XFlash_GetBlockStatus (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)
XStatus XFlash_EraseChip (**XFlash** *InstancePtr)

Define Documentation

#define XFL_MANUFACTURER_ID_INTEL

Supported manufacturer IDs. Note, that not all parts from these listed vendors are supported.

#define XFL_NON_BLOCKING_ERASE_OPTION

XFL_NON_BLOCKING_ERASE_OPTION	Controls whether the interface blocks on device erase until the operation is completed 1=noblock, 0=block
XFL_NON_BLOCKING_WRITE_OPTION	Controls whether the interface blocks on device program until the operation is completed 1=noblock, 0=block

#define XFL_NON_BLOCKING_WRITE_OPTION

```
XFL_NON_BLOCKING_ERASE_OPTION    Controls whether the interface blocks on
                                  device erase until the operation is
                                  completed 1=noblock, 0=block
XFL_NON_BLOCKING_WRITE_OPTION    Controls whether the interface blocks on
                                  device program until the operation is
                                  completed 1=noblock, 0=block
```

Typedef Documentation

typedef struct **XFlashTag** XFlash

The XFlash driver instance data. The user is required to allocate a variable of this type for every flash device in the system. A pointer to a variable of this type is then passed to the driver API functions.

Function Documentation

```
XStatus XFlash_DeviceControl( XFlash * InstancePtr,
                               Xuint32   Command,
                               Xuint32   Param,
                               Xuint32 * ReturnPtr
                               )
```

Accesses device specific data or commands. For a list of commands, see derived component documentation.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.
Command is the device specific command to issue
Param is the command parameter
ReturnPtr is the result of command (if any)

Returns:

- XST_SUCCESS if successful
- XST_FLASH_NOT_SUPPORTED if the command is not recognized/supported by the device(s).

Note:

None.


```
XStatus XFlash_Erase( XFlash * InstancePtr,  
                    Xuint32 Offset,  
                    Xuint32 Bytes  
                    )
```

Erases the specified address range.

Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

Erase the specified range of the device(s). Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_ERASE_OPTION option has an impact on the number of bytes that can be erased in a single call to this function:

- If clear, then the number of bytes to erase can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the the block size of the device at the provided offset and whether the device(s) contains a block erase queue.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin erasure.

Bytes is the number of bytes to erase.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is one of the following:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_BLOCKING_CALL_ERROR if the amount of data to be erased exceeds the erase queue capacity of the device(s).

If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_ERROR if an erase error occurred. This error is usually device specific. Use **XFlash_DeviceControl()** to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially erased.

Note:

Due to flash memory design, the range actually erased may be larger than what was specified by the

Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```
XStatus XFlash_EraseBlock( XFlash * InstancePtr,  
                          Xuint16 Region,  
                          Xuint16 Block,  
                          Xuint16 NumBlocks  
                          )
```

Erases the specified block.

Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_ERASE_OPTION option has an impact on the number of bytes that can be erased in a single call to this function:

- If clear, then the number of bytes to erase can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the the block size of the device at the provided offset and whether the device(s) contains a block erase queue.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is one of the below.

- XST_SUCCESS if successfull.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.

If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional codes can be returned

- XST_FLASH_ERROR if an erase error occured. This error is usually device specific. Use **XFlash_DeviceControl()** to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially erased.

Note:

The arguments point to a starting Region and Block. The NumBlocks parameter may cross over Region boundaries as long as the entire range lies within the part(s) address range and the XFL_NON_BLOCKING_ERASE_OPTION option is not set.

```
XStatus XFlash_EraseBlockResume( XFlash * InstancePtr,  
                                Xuint16 Region,  
                                Xuint16 Block  
                                )
```

Resumes an erase operation that was suspended with XFlash_EraseBlockSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.
Region is the region containing block
Block is the block that is being erased

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```
XStatus XFlash_EraseBlockSuspend( XFlash * InstancePtr,  
                                 Xuint16 Region,  
                                 Xuint16 Block  
                                 )
```

Suspends a currently in progress erase operation and place the device(s) in read mode. When suspended, any block not being erased can be read.

This function should be used only when the XFL_NON_BLOCKING_ERASE_OPTION option is set and a previous call to XFlash_EraseBlock() has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.
Region is the region containing block
Block is the block that is being erased

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region &

Block parameters are ignored.

XStatus XFlash_EraseChip(XFlash * InstancePtr)

Erases the entire device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is always XST_SUCCESS. If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_NOT_SUPPORTED if the chip erase is not supported by the device(s).
- XST_FLASH_ERROR if the device(s) have experienced an internal error during the operation. **XFlash_DeviceControl()** must be used to access the cause of the device specific error condition.

Note:

None.

XStatus XFlash_EraseResume(XFlash * InstancePtr, Xuint32 Offset)

Resumes an erase operation that was suspended with XFlash_EraseSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where erase resumption should be Resumed.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```
XStatus XFlash_EraseSuspend( XFlash * InstancePtr,  
                             Xuint32 Offset  
                             )
```

Suspends a currently in progress erase operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the XFL_NON_BLOCKING_ERASE_OPTION option is set and a previous call to [XFlash_Erase\(\)](#) has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where suspension should occur.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```
XStatus XFlash_GetBlockStatus( XFlash * InstancePtr,  
                              Xuint16 Region,  
                              Xuint16 Block  
                              )
```

Returns the status of the device. This function is intended to be used to poll the device in the following circumstances:

- After calling XFlash_WriteBlock with XFL_NON_BLOCKING_WRITE_OPTION option set.
- After calling XFlash_EraseBlock with XFL_NON_BLOCKING_ERASE_OPTION option set.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

Returns:

- XST_FLASH_READY if the device(s) have completed the previous operation without error.
- XST_FLASH_BUSY if the device(s) are currently performing an erase or write operation.
- XST_FLASH_ERROR if the device(s) have experienced an internal error occurring during another operation such as write, erase, or block locking. [XFlash_DeviceControl\(\)](#) must be used to access the cause of the device specific error condition.

Note:

With some types of flash devices, there may be no difference between using XFlash_GetBlockStatus or XFlash_GetStatus. See your part data sheet for more information.

XFlashGeometry* XFlash_GetGeometry(**XFlash *** *InstancePtr*)

Gets the instance's geometry data

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Instance's Geometry structure

Note:

None.

Xuint32 XFlash_GetOptions(**XFlash *** *InstancePtr*)

Gets interface options for this device instance.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Current options flag

Note:

None.

XFlashProperties* XFlash_GetProperties(**XFlash *** *InstancePtr*)

Gets the instance's property data

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Instance's Properties structure

Note:

None.

```
XStatus XFlash_GetStatus( XFlash * InstancePtr,  
                          Xuint32 Offset  
                          )
```

Returns the status of the device. This function is intended to be used to poll the device in the following circumstances:

- After calling XFlash_Write with XFL_NON_BLOCKING_WRITE_OPTION option set.
- After calling XFlash_Erase or XFlash_EraseChip with XFL_NON_BLOCKING_ERASE_OPTION option set.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the part.

Returns:

- XST_FLASH_READY if the device(s) have completed the previous operation without error.
- XST_FLASH_BUSY if the device(s) are currently performing an erase or write operation.
- XST_FLASH_ERROR if the device(s) have experienced an internal error occurring during another operation such as write, erase, or block locking. [XFlash_DeviceControl\(\)](#) must be used to access the cause of the device specific error condition.

Note:

With some types of flash devices, there may be no difference between using XFlash_GetBlockStatus or XFlash_GetStatus. See your part data sheet for more information.

```
XStatus XFlash_Initialize( XFlash * InstancePtr,  
                          Xuint16 DeviceId  
                          )
```

Initializes a specific XFlash instance. The initialization entails:

- Issuing the CFI query command
- Get and translate relevant CFI query information.
- Set default options for the instance.
- Setup the VTable.
- Call the initialize function of the instance, which does the following:
 - Get and translate extended vendor CFI query information.
 - Some VTable functions may be replaced with more efficient ones based on data extracted from the extended CFI query. A replacement example would be a buffered XFlash_WriteBlock replacing a non-buffered XFlash_WriteBlock.
 - Reset the device by clearing any status information and placing the device in read mode.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

DeviceId is the unique id of the device controlled by this component. Passing in a device id associates the generic component to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful. On error, a code indicating the specific error is returned. Possible error codes are:

- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.
- XST_FLASH_PART_NOT_SUPPORTED if the command set algorithm or Layout is not supported by any derived component compiled into the system.
- XST_FLASH_TOO_MANY_REGIONS if the part contains too many erase regions. This can be fixed by increasing the value of XFL_MAX_ERASE_REGIONS then re-compiling the driver.
- XST_FLASH_CFI_QUERY_ERROR if the device would not enter CFI query mode. Either the device(s) do not support CFI, the wrong BaseAddress param was used, an unsupported part layout exists, or a hardware problem exists with the part.

Note:

None.

```
XStatus XFlash_Lock( XFlash * InstancePtr,  
                    Xuint32  Offset,  
                    Xuint32  Bytes  
                    )
```


Locks the blocks in the specified range of the device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin block locking.

Bytes is the number of bytes to lock.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

Due to flash memory design, the range actually locked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```
XStatus XFlash_LockBlock( XFlash * InstancePtr,  
                          Xuint16 Region,  
                          Xuint16 Block,  
                          Xuint16 NumBlocks  
                          )
```

Locks the specified block. Prevents it from being erased or written.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase. The number may extend into a different region.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

None.

```
XFlash_Config* XFlash_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID.

Parameters:

DeviceId is the unique device ID to be searched for in the table

Returns:

Returns a pointer to the configuration data for the device, or XNULL if not device is found.

```
XStatus XFlash_Read( XFlash * InstancePtr,  
                    Xuint32 Offset,  
                    Xuint32 Bytes,  
                    void * DestPtr  
                    )
```

Copies data from the device(s) memory space to a user buffer. The source and destination addresses can be on any alignment supported by the processor.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to read.

Bytes is the number of bytes to copy.

DestPtr is the destination address to copy data to.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the source address does not start within the addressable areas of the device(s).

Note:

This function allows the transfer of data past the end of the device's address space. If this occurs, then results are undefined.

```
XStatus XFlash_ReadBlock( XFlash * InstancePtr,  
                         Xuint16 Region,  
                         Xuint16 Block,  
                         Xuint32 Offset,  
                         Xuint32 Bytes,  
                         void * DestPtr  
                         )
```

Copy data from a specific device block to a user buffer. The source and destination addresses can be on any byte alignment supported by the target processor.

Parameters:

- InstancePtr* is a pointer to the XFlash instance to be worked on.
- Region* is the erase region the block appears in.
- Block* is the block number within the erase region.
- Offset* is the starting offset in the block where reading will begin.
- Bytes* is the number of bytes to copy.
- DestPtr* is the destination address to copy data to.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if Region, Block, and Offset parameters do not point to a valid block.

Note:

The arguments point to a starting Region, Block, and Offset within that block. The Bytes parameter may cross over Region and Block boundaries. If Bytes extends past the end of the device's address space, then results are undefined.

XStatus XFlash_Reset(XFlash * InstancePtr)

Clears the device(s) status register(s) and place the device(s) into read mode.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_BUSY if the flash devices were in the middle of an operation and could not be reset.
- XST_FLASH_ERROR if the device(s) have experienced an internal error during the operation. **XFlash_DeviceControl()** must be used to access the cause of the device specific error condition.

Note:

None.

XStatus XFlash_SelfTest(XFlash * InstancePtr)

Runs a self-test on the driver/device. This is a destructive test. Tests performed include:

- Address bus test
- Data bus test

When the tests are complete, the device is reset back into read mode.

Parameters:

InstancePtr is a pointer to the XComponent instance to be worked on.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ERROR if any test fails.

Note:

None.

```
XStatus XFlash_SetOptions( XFlash * InstancePtr,  
                          Xuint32 OptionsFlag  
                          )
```

Sets interface options for this device instance.

Here are the currently available options: <pre XFL_NON_BLOCKING_WRITE_OPTION Blocking write on or off XFL_NON_BLOCKING_ERASE_OPTION Blocking erase on or off To set multiple options, OR the option constants together.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

OptionsFlag is the options to set. 1=set option, 0=clear option.

Returns:

- XST_SUCCESS if options successfully set.
- XST_FLASH_NOT_SUPPORTED if option is not supported.

Note:

None.

```
XStatus XFlash_Unlock( XFlash * InstancePtr,  
                      Xuint32 Offset,  
                      Xuint32 Bytes  
                      )
```

Unlocks the blocks in the specified range of the device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin block unlocking.

Bytes is the number of bytes to unlock.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

Due to flash memory design, the range actually unlocked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```
XStatus XFlash_UnlockBlock( XFlash * InstancePtr,  
                           Xuint16 Region,  
                           Xuint16 Block,  
                           Xuint16 NumBlocks  
                           )
```

Unlocks the specified block.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase. The number may extend into a different region.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

None.

```
XStatus XFlash_Write( XFlash * InstancePtr,  
                    Xuint32 Offset,  
                    Xuint32 Bytes,  
                    void * SrcPtr  
                    )
```

Programs the devices with data stored in the user buffer. The source and destination address must be aligned to the width of the flash's data bus.

Returns immediately if the XFL_NON_BLOCKING_WRITE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_WRITE_OPTION option has an impact on the number of bytes that can be written:

- If clear, then the number of bytes to write can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the alignment and size of the device's write buffer. The rule is that the number of bytes being written cannot cross over an alignment boundary. Alignment information is obtained in the InstancePtr->Properties.ProgCap attribute.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin programming. Must be aligned to the width of the flash's data bus.

Bytes is the number of bytes to program.

SrcPtr is the source address containing data to be programmed. Must be aligned to the width of the flash's data bus.

Returns:

If XFL_NON_BLOCKING_WRITE_OPTION option is set, then the return value is one of the following:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_ALIGNMENT_ERROR if the Offset or SrcPtr is not aligned to the width of the flash's data bus.
- XST_BLOCKING_CALL_ERROR if the write would cross a write buffer boundary.

If XFL_NON_BLOCKING_WRITE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_ERROR if a write error occurred. This error is usually device specific. Use **XFlash_DeviceControl()** to retrieve specific error conditions. When this error is returned, it is

possible that the target address range was only partially programmed.

Note:

None.

```
XStatus XFlash_WriteBlock( XFlash * InstancePtr,  
                           Xuint16 Region,  
                           Xuint16 Block,  
                           Xuint32 Offset,  
                           Xuint32 Bytes,  
                           void * SrcPtr  
                           )
```

Programs the devices with data stored in the user buffer. The source and destination address can be on any alignment supported by the processor. This function will block until the operation completes or an error is detected.

Returns immediately if the XFL_NON_BLOCKING_WRITE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_WRITE_OPTION option has an impact on the number of bytes that can be written:

- If clear, then the number of bytes to write can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the alignment and size of the device's write buffer. The rule is that the number of bytes being written cannot cross over an alignment boundary. Alignment information is obtained in the InstancePtr->Properties.ProgCap attribute.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

Offset is the starting offset in the block where writing will begin.

Bytes is the number of bytes to write.

SrcPtr is the source address containing data to be programmed

Returns:

If XFL_NON_BLOCKING_WRITE_OPTION option is set, then the return value is XST_SUCCESS if successful. On error, a code indicating the specific error is returned. Possible error codes are:

- XST_FLASH_ADDRESS_ERROR if Region, Block, and Offset parameters do not point to a valid block. Or, the Bytes parameter causes the read to go past the last addressable byte in the device(s).
- XST_FLASH_ALIGNMENT_ERROR if the Offset or SrcPtr is not aligned to the width of the

flash's data bus.

- XST_BLOCKING_CALL_ERROR if the write would cross a write buffer boundary.

If XFL_NON_BLOCKING_WRITE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_ERROR if a write error occurred. This error is usually device specific. Use **XFlash_DeviceControl()** to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially programmed.

Note:

The arguments point to a starting Region, Block, and Offset within that block. The Bytes parameter may cross over Region and Block boundaries as long as the entire range lies within the part(s) address range and the XFL_NON_BLOCKING_WRITE_OPTION option is not set.

```
XStatus XFlash_WriteBlockResume( XFlash * InstancePtr,  
                                Xuint16 Region,  
                                Xuint16 Block,  
                                Xuint32 Offset  
                                )
```

Resumes a write operation that was suspended with XFlash_WriteBlockSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the region containing block

Block is the block that is being erased

Offset is the offset in the device where resumption should occur

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```
XStatus XFlash_WriteBlockSuspend( XFlash * InstancePtr,  
                                  Xuint16 Region,  
                                  Xuint16 Block,  
                                  Xuint32 Offset  
                                  )
```


Suspends a currently in progress write operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the `XFL_NON_BLOCKING_WRITE_OPTION` option is set and a previous call to `XFlash_WriteBlock()` has been made. Otherwise, undetermined results may occur.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the region containing block
- Block* is the block that is being written
- Offset* is the offset in the device where suspension should occur

Returns:

- `XST_SUCCESS` if successful.
- `XST_FLASH_NOT_SUPPORTED` if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored in those cases.

```
XStatus XFlash_WriteResume( XFlash * InstancePtr,  
                           Xuint32 Offset  
                           )
```

Resumes a write operation that was suspended with `XFlash_WriteSuspend`.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Offset* is the offset with the device where write resumption should occur.

Returns:

- `XST_SUCCESS` if successful.
- `XST_FLASH_NOT_SUPPORTED` if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```
XStatus XFlash_WriteSuspend( XFlash * InstancePtr,  
                             Xuint32 Offset  
                             )
```

Suspends a currently in progress write operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the `XFL_NON_BLOCKING_ERASE_OPTION` option is set and a previous call to `XFlash_Write()` has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where suspension should occur.

Returns:

- `XST_SUCCESS` if successful.
- `XST_FLASH_NOT_SUPPORTED` if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

flash/v1_00_a/src/xflash.c File Reference

Detailed Description

This module implements the base component for flash memory devices that conform to the "Common Flash Interface" (CFI) standard. CFI allows a single flash driver to be used for an entire family of parts.

This is not a driver for a specific device, but for a set of command read/write/erase algorithms. CFI allows us to determine which algorithm to utilize at runtime. It is this set of command algorithms that will be implemented as the derived component.

Flash memory space is segmented into areas called blocks. The size of each block is based on a power of 2. A region is defined as a contiguous set of blocks of the same size. Some parts have several regions while others have one. The arrangement of blocks and regions is referred to by this module as the part's geometry.

The cells within the part can be programmed from a logic 1 to a logic 0 and not the other way around. To change a cell back to a logic 1, the entire block containing that cell must be erased. When a block is erased all bytes contain the value 0xFF. The number of times a block can be erased is finite. Eventually the block will wear out and will no longer be capable of erasure. As of this writing, the typical flash block can be erased 100,000 or more times.

Features provided by this module:

- Part timing, geometry, features, and command algorithm determined by CFI query.
- Supported architectures include:
 - 16-bit data bus: Single x16 part in word mode
 - 32-bit data bus: Two x16 parts in word mode.
- Two read/write/erase APIs.
- Erase/write suspension.
- Self test diagnostics.
- Block locking (if supported by specific part).
- Chip erase (if supported by specific part).

- Non-blocking write & erase function calls.
- Part specific control. Supported features of individual parts are listed within the driver module for that part.

Features listed above not currently implemented include:

- Non-blocking write & erase function calls.
- Block locking.
- Support of more architectures.
- Self test diagnostics.

This component exports two differing types of read/write/erase APIs. The geometry un-aware API allows the user to ignore the geometry of the underlying flash device. The geometry aware API operates on specific blocks. The former API is designed for casual use while the latter may prove useful for designers wishing to use this driver under a flash file system. Both APIs can be used interchangeably.

Write and erase function calls can be set to return immediately (non-blocking) even though the intended operation is incomplete. This is useful for systems that utilize a watchdog timer. It also facilitates the use of an interrupt driven programming algorithm. This feature is dependent upon the capabilities of the flash devices.

If the geometry un-aware API is used, then the user requires no knowledge of the underlying hardware. Usage of this API along with non-blocking write/erase should be done carefully because non-blocking write/erase assumes some knowledge of the device(s) geometry.

If part specific advanced features are required, then the XFlash_DeviceControl function is available provided the feature has been implemented by the part driver module.

Note:

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, cache control, or HW write protection management must be satisfied by the layer above this driver.

Use of this driver by multiple threads must be carefully thought out taking into consideration the underlying flash devices in use. This driver does not use mutual exclusion or critical region control.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----

```
#include "xflash.h"  
#include "xflash_cfi.h"  
#include "xparameters.h"
```

Functions

XStatus XFlash_Initialize (**XFlash** *InstancePtr, **Xuint16** DeviceId)

XFlash_Config * **XFlash_LookupConfig** (**Xuint16** DeviceId)

XStatus XFlash_ReadBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)

XStatus XFlash_WriteBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)

XStatus XFlash_WriteBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)

XStatus XFlash_WriteBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)

XStatus XFlash_EraseBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)

XStatus XFlash_EraseBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)

XStatus XFlash_EraseBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)

XStatus XFlash_LockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)

XStatus XFlash_UnlockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)

XStatus XFlash_GetBlockStatus (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)

XStatus XFlash_Read (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)

XStatus XFlash_Write (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)

XStatus XFlash_WriteSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlash_WriteResume (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlash_Erase (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlash_EraseSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlash_EraseResume (**XFlash** *InstancePtr, **Xuint32** Offset)
XStatus XFlash_Lock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlash_Unlock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)
XStatus XFlash_GetStatus (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlash_EraseChip (**XFlash** *InstancePtr)

XStatus XFlash_SelfTest (**XFlash** *InstancePtr)

XStatus XFlash_Reset (**XFlash** *InstancePtr)

XStatus XFlash_SetOptions (**XFlash** *InstancePtr, **Xuint32** OptionsFlag)

Xuint32 XFlash_GetOptions (**XFlash** *InstancePtr)

XFlashGeometry * **XFlash_GetGeometry** (**XFlash** *InstancePtr)

XFlashProperties * **XFlash_GetProperties** (**XFlash** *InstancePtr)

XStatus XFlash_DeviceControl (**XFlash** *InstancePtr, **Xuint32** Command, **Xuint32** Param, **Xuint32** *ReturnPtr)

Xboolean XFlash_IsReady (**XFlash** *InstancePtr)

Function Documentation

```
XStatus XFlash_DeviceControl( XFlash * InstancePtr,  
                             Xuint32 Command,  
                             Xuint32 Param,  
                             Xuint32 * ReturnPtr  
                             )
```

Accesses device specific data or commands. For a list of commands, see derived component documentation.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Command is the device specific command to issue

Param is the command parameter

ReturnPtr is the result of command (if any)

Returns:

- XST_SUCCESS if successful
- XST_FLASH_NOT_SUPPORTED if the command is not recognized/supported by the device(s).

Note:

None.

```
XStatus XFlash_Erase( XFlash * InstancePtr,  
                     Xuint32 Offset,  
                     Xuint32 Bytes  
                     )
```

Erases the specified address range.

Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

Erase the specified range of the device(s). Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_ERASE_OPTION option has an impact on the number of bytes that can be erased in a single call to this function:

- If clear, then the number of bytes to erase can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the the block size of the device at the provided offset and whether the device(s) contains a block erase queue.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin erasure.

Bytes is the number of bytes to erase.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is one of the following:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_BLOCKING_CALL_ERROR if the amount of data to be erased exceeds the erase queue capacity of the device(s).

If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional

codes can be returned:

- XST_FLASH_ERROR if an erase error occurred. This error is usually device specific. Use **XFlash_DeviceControl()** to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially erased.

Note:

Due to flash memory design, the range actually erased may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```
XStatus XFlash_EraseBlock( XFlash * InstancePtr,  
                           Xuint16 Region,  
                           Xuint16 Block,  
                           Xuint16 NumBlocks  
                           )
```

Erases the specified block.

Returns immediately if the XFL_NON_BLOCKING_ERASE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_ERASE_OPTION option has an impact on the number of bytes that can be erased in a single call to this function:

- If clear, then the number of bytes to erase can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the the block size of the device at the provided offset and whether the device(s) contains a block erase queue.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase.

Returns:

If XFL_NON_BLOCKING_ERASE_OPTION option is set, then the return value is one of the below.

- XST_SUCCESS if successfull.

- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.

If XFL_NON_BLOCKING_ERASE_OPTION option is clear, then the following additional codes can be returned

- XST_FLASH_ERROR if an erase error occurred. This error is usually device specific. Use **XFlash_DeviceControl()** to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially erased.

Note:

The arguments point to a starting Region and Block. The NumBlocks parameter may cross over Region boundaries as long as the entire range lies within the part(s) address range and the XFL_NON_BLOCKING_ERASE_OPTION option is not set.

```
XStatus XFlash_EraseBlockResume( XFlash * InstancePtr,
                                Xuint16 Region,
                                Xuint16 Block
                                )
```

Resumes an erase operation that was suspended with XFlash_EraseBlockSuspend.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the region containing block
- Block* is the block that is being erased

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```
XStatus XFlash_EraseBlockSuspend( XFlash * InstancePtr,
                                  Xuint16 Region,
                                  Xuint16 Block
                                  )
```

Suspends a currently in progress erase operation and place the device(s) in read mode. When suspended, any block not being erased can be read.

This function should be used only when the `XFL_NON_BLOCKING_ERASE_OPTION` option is set and a previous call to `XFlash_EraseBlock()` has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.
Region is the region containing block
Block is the block that is being erased

Returns:

- `XST_SUCCESS` if successful.
- `XST_FLASH_NOT_SUPPORTED` if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

XStatus XFlash_EraseChip(XFlash * InstancePtr)

Erases the entire device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

If `XFL_NON_BLOCKING_ERASE_OPTION` option is set, then the return value is always `XST_SUCCESS`. If `XFL_NON_BLOCKING_ERASE_OPTION` option is clear, then the following additional codes can be returned:

- `XST_FLASH_NOT_SUPPORTED` if the chip erase is not supported by the device(s).
- `XST_FLASH_ERROR` if the device(s) have experienced an internal error during the operation. `XFlash_DeviceControl()` must be used to access the cause of the device specific error condition.

Note:

None.

```
XStatus XFlash_EraseResume( XFlash * InstancePtr,  
                             Xuint32 Offset  
                             )
```

Resumes an erase operation that was suspended with XFlash_EraseSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where erase resumption should be Resumed.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```
XStatus XFlash_EraseSuspend( XFlash * InstancePtr,  
                             Xuint32 Offset  
                             )
```

Suspends a currently in progress erase operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the XFL_NON_BLOCKING_ERASE_OPTION option is set and a previous call to **XFlash_Erase()** has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where suspension should occur.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```
XStatus XFlash_GetBlockStatus( XFlash * InstancePtr,  
                               Xuint16 Region,  
                               Xuint16 Block  
                               )
```

Returns the status of the device. This function is intended to be used to poll the device in the following circumstances:

- After calling XFlash_WriteBlock with XFL_NON_BLOCKING_WRITE_OPTION option set.
- After calling XFlash_EraseBlock with XFL_NON_BLOCKING_ERASE_OPTION option set.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

Returns:

- XST_FLASH_READY if the device(s) have completed the previous operation without error.
- XST_FLASH_BUSY if the device(s) are currently performing an erase or write operation.
- XST_FLASH_ERROR if the device(s) have experienced an internal error occurring during another operation such as write, erase, or block locking.
XFlash_DeviceControl() must be used to access the cause of the device specific error condition.

Note:

With some types of flash devices, there may be no difference between using XFlash_GetBlockStatus or XFlash_GetStatus. See your part data sheet for more information.

```
XFlashGeometry* XFlash_GetGeometry( XFlash * InstancePtr)
```

Gets the instance's geometry data

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Instance's Geometry structure

Note:

None.

Xuint32 XFlash_GetOptions(XFlash * *InstancePtr*)

Gets interface options for this device instance.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Current options flag

Note:

None.

XFlashProperties* XFlash_GetProperties(XFlash * *InstancePtr*)

Gets the instance's property data

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Returns:

Instance's Properties structure

Note:

None.

```
XStatus XFlash_GetStatus( XFlash * InstancePtr,  
                          Xuint32 Offset  
                          )
```

Returns the status of the device. This function is intended to be used to poll the device in the following circumstances:

- After calling XFlash_Write with XFL_NON_BLOCKING_WRITE_OPTION option set.
- After calling XFlash_Erase or XFlash_EraseChip with XFL_NON_BLOCKING_ERASE_OPTION option set.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the part.

Returns:

- XST_FLASH_READY if the device(s) have completed the previous operation without error.
- XST_FLASH_BUSY if the device(s) are currently performing an erase or write operation.
- XST_FLASH_ERROR if the device(s) have experienced an internal error occurring during another operation such as write, erase, or block locking.
XFlash_DeviceControl() must be used to access the cause of the device specific error condition.

Note:

With some types of flash devices, there may be no difference between using XFlash_GetBlockStatus or XFlash_GetStatus. See your part data sheet for more information.

```
XStatus XFlash_Initialize( XFlash * InstancePtr,  
                          Xuint16 DeviceId  
                          )
```

Initializes a specific XFlash instance. The initialization entails:

- Issuing the CFI query command
- Get and translate relevant CFI query information.
- Set default options for the instance.
- Setup the VTable.
- Call the initialize function of the instance, which does the following:
 - Get and translate extended vendor CFI query information.
 - Some VTable functions may be replaced with more efficient ones based on data extracted from the extended CFI query. A replacement example would be a buffered XFlash_WriteBlock replacing a non-buffered XFlash_WriteBlock.
 - Reset the device by clearing any status information and placing the device in read mode.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

DeviceId is the unique id of the device controlled by this component. Passing in a device id associates the generic component to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful. On error, a code indicating the specific error is returned. Possible error codes are:

- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.
- XST_FLASH_PART_NOT_SUPPORTED if the command set algorithm or Layout is not supported by any derived component compiled into the system.
- XST_FLASH_TOO_MANY_REGIONS if the part contains too many erase regions. This can be fixed by increasing the value of XFL_MAX_ERASE_REGIONS then re-compiling the driver.
- XST_FLASH_CFI_QUERY_ERROR if the device would not enter CFI query mode. Either the device(s) do not support CFI, the wrong BaseAddress param was used, an unsupported part layout exists, or a hardware problem exists with the part.

Note:

None.

Xboolean XFlash_IsReady(**XFlash** * *InstancePtr*)

Checks the readiness of the device, which means it has been successfully initialized.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

Returns:

XTRUE if the device has been initialized (but not necessarily started), and XFALSE otherwise.

Note:

This function only exists in the base component since it is common across all derived components. Asserts based on the IsReady flag exist only in the base component.

```
XStatus XFlash_Lock( XFlash * InstancePtr,  
                    Xuint32 Offset,  
                    Xuint32 Bytes  
                    )
```

Locks the blocks in the specified range of the device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin block locking.

Bytes is the number of bytes to lock.

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- o XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

Due to flash memory design, the range actually locked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.


```
XStatus XFlash_LockBlock( XFlash * InstancePtr,  
                          Xuint16 Region,  
                          Xuint16 Block,  
                          Xuint16 NumBlocks  
                          )
```

Locks the specified block. Prevents it from being erased or written.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase. The number may extend into a different region.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

None.

```
XFlash_Config* XFlash_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID.

Parameters:

DeviceId is the unique device ID to be searched for in the table

Returns:

Returns a pointer to the configuration data for the device, or XNULL if not device is found.

```
XStatus XFlash_Read( XFlash * InstancePtr,  
                    Xuint32 Offset,  
                    Xuint32 Bytes,  
                    void * DestPtr  
                    )
```

Copies data from the device(s) memory space to a user buffer. The source and destination addresses can be on any alignment supported by the processor.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Offset* is the offset into the device(s) address space from which to read.
- Bytes* is the number of bytes to copy.
- DestPtr* is the destination address to copy data to.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the source address does not start within the addressable areas of the device(s).

Note:

This function allows the transfer of data past the end of the device's address space. If this occurs, then results are undefined.

```
XStatus XFlash_ReadBlock( XFlash * InstancePtr,  
                         Xuint16 Region,  
                         Xuint16 Block,  
                         Xuint32 Offset,  
                         Xuint32 Bytes,  
                         void * DestPtr  
                         )
```

Copy data from a specific device block to a user buffer. The source and destination addresses can be on any byte alignment supported by the target processor.

Parameters:

- InstancePtr* is a pointer to the XFlash instance to be worked on.
- Region* is the erase region the block appears in.
- Block* is the block number within the erase region.
- Offset* is the starting offset in the block where reading will begin.
- Bytes* is the number of bytes to copy.
- DestPtr* is the destination address to copy data to.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if Region, Block, and Offset parameters do not point to a valid block.

Note:

The arguments point to a starting Region, Block, and Offset within that block. The Bytes parameter may cross over Region and Block boundaries. If Bytes extends past the end of the device's address space, then results are undefined.

XStatus XFlash_Reset(XFlash * InstancePtr)

Clears the device(s) status register(s) and place the device(s) into read mode.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_BUSY if the flash devices were in the middle of an operation and could not be reset.
- XST_FLASH_ERROR if the device(s) have experienced an internal error during the operation. **XFlash_DeviceControl()** must be used to access the cause of the device specific error condition.

Note:

None.

XStatus XFlash_SelfTest(**XFlash** * *InstancePtr*)

Runs a self-test on the driver/device. This is a destructive test. Tests performed include:

- Address bus test
- Data bus test

When the tests are complete, the device is reset back into read mode.

Parameters:

InstancePtr is a pointer to the XComponent instance to be worked on.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ERROR if any test fails.

Note:

None.

XStatus XFlash_SetOptions(**XFlash** * *InstancePtr*, **Xuint32** *OptionsFlag*)

Sets interface options for this device instance.

Here are the currently available options: <pre XFL_NON_BLOCKING_WRITE_OPTION Blocking write on or off XFL_NON_BLOCKING_ERASE_OPTION Blocking erase on or off To set multiple options, OR the option constants together.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

OptionsFlag is the options to set. 1=set option, 0=clear option.

Returns:

- XST_SUCCESS if options successfully set.
- XST_FLASH_NOT_SUPPORTED if option is not supported.

Note:

None.

```
XStatus XFlash_Unlock( XFlash * InstancePtr,  
                        Xuint32 Offset,  
                        Xuint32 Bytes  
                        )
```

Unlocks the blocks in the specified range of the device(s).

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin block unlocking.

Bytes is the number of bytes to unlock.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

Due to flash memory design, the range actually unlocked may be larger than what was specified by the Offset & Bytes parameters. This will occur if the parameters do not align to block boundaries.

```
XStatus XFlash_UnlockBlock( XFlash * InstancePtr,  
                            Xuint16 Region,  
                            Xuint16 Block,  
                            Xuint16 NumBlocks  
                            )
```

Unlocks the specified block.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

NumBlocks is the the number of blocks to erase. The number may extend into a different region.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if region and/or block do not specify a valid block within the device.
- XST_FLASH_NOT_SUPPORTED if the device(s) do not support block locking.

Note:

None.

```
XStatus XFlash_Write( XFlash * InstancePtr,  
                    Xuint32 Offset,  
                    Xuint32 Bytes,  
                    void * SrcPtr  
                    )
```

Programs the devices with data stored in the user buffer. The source and destination address must be aligned to the width of the flash's data bus.

Returns immediately if the XFL_NON_BLOCKING_WRITE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_WRITE_OPTION option has an impact on the number of bytes that can be written:

- If clear, then the number of bytes to write can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the alignment and size of the device's write buffer. The rule is that the number of bytes being written cannot cross over an alignment boundary. Alignment information is obtained in the InstancePtr->Properties.ProgCap attribute.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset into the device(s) address space from which to begin programming. Must be aligned to the width of the flash's data bus.

Bytes is the number of bytes to program.

SrcPtr is the source address containing data to be programmed. Must be aligned to the width of the flash's data bus.

Returns:

If XFL_NON_BLOCKING_WRITE_OPTION option is set, then the return value is one of the following:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the destination address range is not completely within the addressable areas of the device(s).
- XST_FLASH_ALIGNMENT_ERROR if the Offset or SrcPtr is not aligned to the width of the flash's data bus.
- XST_BLOCKING_CALL_ERROR if the write would cross a write buffer boundary.

If XFL_NON_BLOCKING_WRITE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_ERROR if a write error occurred. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially programmed.

Note:

None.

```
XStatus XFlash_WriteBlock( XFlash * InstancePtr,  
                           Xuint16 Region,  
                           Xuint16 Block,  
                           Xuint32 Offset,  
                           Xuint32 Bytes,  
                           void * SrcPtr  
                           )
```

Programs the devices with data stored in the user buffer. The source and destination address can be on any alignment supported by the processor. This function will block until the operation completes or an error is detected.

Returns immediately if the XFL_NON_BLOCKING_WRITE_OPTION option is set. If clear, the device(s) are polled until an error, timeout or the operation completes successfully.

The XFL_NON_BLOCKING_WRITE_OPTION option has an impact on the number of bytes that can be written:

- If clear, then the number of bytes to write can be any number as long as it is within the bounds of the device(s).
- If set, then the number of bytes depends on the alignment and size of the device's write buffer. The rule is that the number of bytes being written cannot cross over an alignment boundary. Alignment information is obtained in the InstancePtr->Properties.ProgCap attribute.

Parameters:

InstancePtr is a pointer to the XFlash instance to be worked on.

Region is the erase region the block appears in.

Block is the block number within the erase region.

Offset is the starting offset in the block where writing will begin.

Bytes is the number of bytes to write.

SrcPtr is the source address containing data to be programmed

Returns:

If XFL_NON_BLOCKING_WRITE_OPTION option is set, then the return value is XST_SUCCESS if successful. On error, a code indicating the specific error is returned.

Possible error codes are:

- XST_FLASH_ADDRESS_ERROR if Region, Block, and Offset parameters do not point to a valid block. Or, the Bytes parameter causes the read to go past the last addressable byte in the device(s).
- XST_FLASH_ALIGNMENT_ERROR if the Offset or SrcPtr is not aligned to the width of the flash's data bus.
- XST_BLOCKING_CALL_ERROR if the write would cross a write buffer boundary.

If XFL_NON_BLOCKING_WRITE_OPTION option is clear, then the following additional codes can be returned:

- XST_FLASH_ERROR if a write error occurred. This error is usually device specific. Use [XFlash_DeviceControl\(\)](#) to retrieve specific error conditions. When this error is returned, it is possible that the target address range was only partially programmed.

Note:

The arguments point to a starting Region, Block, and Offset within that block. The Bytes parameter may cross over Region and Block boundaries as long as the entire range lies within the part(s) address range and the XFL_NON_BLOCKING_WRITE_OPTION option is not set.

```
XStatus XFlash_WriteBlockResume( XFlash * InstancePtr,  
                                Xuint16 Region,  
                                Xuint16 Block,  
                                Xuint32 Offset  
                                )
```

Resumes a write operation that was suspended with XFlash_WriteBlockSuspend.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Region is the region containing block

Block is the block that is being erased

Offset is the offset in the device where resumption should occur

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored.

```
XStatus XFlash_WriteBlockSuspend( XFlash * InstancePtr,  
                                  Xuint16 Region,  
                                  Xuint16 Block,  
                                  Xuint32 Offset  
                                  )
```

Suspends a currently in progress write operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the `XFL_NON_BLOCKING_WRITE_OPTION` option is set and a previous call to `XFlash_WriteBlock()` has been made. Otherwise, undetermined results may occur.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the region containing block
- Block* is the block that is being written
- Offset* is the offset in the device where suspension should occur

Returns:

- o `XST_SUCCESS` if successful.
- o `XST_FLASH_NOT_SUPPORTED` if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore Region & Block parameters are ignored in those cases.

```
XStatus XFlash_WriteResume( XFlash * InstancePtr,  
                             Xuint32 Offset  
                             )
```

Resumes a write operation that was suspended with `XFlash_WriteSuspend`.

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Offset* is the offset with the device where write resumption should occur.

Returns:

- o `XST_SUCCESS` if successful.
- o `XST_FLASH_NOT_SUPPORTED` if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

```
XStatus XFlash_WriteSuspend( XFlash * InstancePtr,  
                             Xuint32 Offset  
                             )
```

Suspends a currently in progress write operation and place the device(s) in read mode. When suspended, any block not being programmed can be read.

This function should be used only when the XFL_NON_BLOCKING_ERASE_OPTION option is set and a previous call to **XFlash_Write()** has been made. Otherwise, undetermined results may occur.

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.

Offset is the offset with the device where suspension should occur.

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_NOT_SUPPORTED if write suspension is not supported by the device(s)

Note:

Some devices such as Intel do not require a block address to suspend erasure. Therefore the Offset parameter is ignored.

flash/v1_00_a/src/xflash_cfi.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xflash_cfi.h,v 1.6 2002/08/13 21:14:26 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file flash/v1_00_a/src/xflash_cfi.h
00026 *
00027 * This is a helper component for XFlash. It contains methods used to extract
00028 * and interpret Common Flash Interface (CFI) from a flash memory part that
00029 * supports the CFI query command.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver   Who   Date       Changes
00035 * ----  ---  -
00036 * 1.00a rmm   07/16/01 First release
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XFLASH_CFI_H /* prevent circular inclusions */
00042 #define XFLASH_CFI_H /* by using protection macros */
```

```

00043
00044 /***** Include Files *****/
00045 #include "xbasic_types.h"
00046 #include "xstatus.h"
00047 #include "xflash.h"
00048 #include "xflash_geometry.h"
00049 #include "xflash_properties.h"
00050
00051 /***** Constant Definitions *****/
00052
00053 /*
00054  * Vendor command set codes
00055  * Refer to industry document "CFI publication 100" for the latest list
00056  */
00057 #define XFL_CMDSET_INTEL_STANDARD      3
00058 #define XFL_CMDSET_INTEL_EXTENDED     1    /* Also includes Sharp */
00059 #define XFL_CMDSET_AMD_STANDARD       2    /* Also includes Fujitsu */
00060 #define XFL_CMDSET_AMD_EXTENDED       4    /* Also includes Fujitsu */
00061 #define XFL_CMDSET_MITSUBISHI_STANDARD 256
00062 #define XFL_CMDSET_MITSUBISHI_EXTENDED 257
00063
00064
00065 /***** Type Definitions *****/
00066
00067
00068 /***** Macros (Inline Functions) Definitions *****/
00069
00070 /*
00071  * XFL_CFI_ macros are used by the base and derived components to access data
00072  * from the flash device when it is in CFI query mode. The query data begins
00073  * at the base address of the part and is interleaved based on the part
00074  * layout:
00075  *           XFL_LAYOUT_X16_X16_X1    Interleave=2
00076  *           XFL_LAYOUT_X16_X16_X2    Interleave=4
00077  */
00078
00079 /*****
00080 /**
00081  *
00082  * Moves the CFI data pointer to a physical address that corresponds to a
00083  * specific CFI byte offset.
00084  *
00085  * @param   Ptr is the pointer to modify. Can be of any type
00086  * @param   BaseAddr is the base address of flash part
00087  * @param   Interleave is the byte interleaving (based on part layout)
00088  * @param   ByteAddr is the byte offset within CFI data to read
00089  *
00090  * @return
00091  *
00092  * The Ptr argument is set to point at the the CFI byte specified by the
00093  * ByteAddr parameter.
00094  */

```

```

00095 */
00096 #define XFL_CFI_POSITION_PTR(Ptr, BaseAddr, Interleave, ByteAddr) \
00097 ( \
00098     Ptr = (void*)((Xuint32)BaseAddr + ((Xuint32)Interleave * \
00099         (Xuint32)ByteAddr)) \
00100 )
00101
00102 /*****
00103 /**
00104 *
00105 * Reads 8-bits of data from the CFI data location into a local variable.
00106 *
00107 * @param   Ptr is the pointer to read. Can be a pointer to any type.
00108 * @param   Interleave is the byte interleaving (based on part layout)
00109 *
00110 * @return
00111 *
00112 * The byte at Ptr adjusted for the interleave factor.
00113 *
00114 */
00115 #define XFL_CFI_READ8(Ptr, Interleave) \
00116     READ_FLASH_8((Xuint32)Ptr + Interleave - 1)
00117
00118 /*****
00119 /**
00120 *
00121 * Reads 16-bits of data from the CFI data location into a local variable.
00122 *
00123 * @param   Ptr is the pointer to read. Can be a pointer to any type.
00124 * @param   Interleave is the byte interleaving (based on part layout)
00125 * @param   Data is the 16-bit storage location for the data to be read.
00126 *
00127 * @return
00128 *
00129 * The 16-bit value at Ptr adjusted for the interleave factor.
00130 *
00131 */
00132 #define XFL_CFI_READ16(Ptr, Interleave, Data) \
00133 { \
00134     (Data) = (Xuint16)READ_FLASH_8((Xuint8*)(Ptr) + ((Interleave) * 2) - 1); \
00135     (Data) <<= 8; \
00136     (Data) |= (Xuint16)READ_FLASH_8((Xuint8*)(Ptr) + (Interleave) - 1); \
00137 }
00138
00139 /*****
00140 /**
00141 *
00142 * Advances the CFI pointer to the next byte
00143 *
00144 * @param   Ptr is the pointer to advance. Can be a pointer to any type.
00145 * @param   Interleave is the byte interleaving (based on part layout)
00146 *

```

```

00147 * @return
00148 *
00149 * Adjusted Ptr.
00150 *
00151 */
00152 #define XFL_CFI_ADVANCE_PTR8(Ptr, Interleave) \
00153     (Ptr = (void*)((Xuint32)Ptr + (Interleave)))
00154
00155 /*****
00156 **
00157 *
00158 * Advances the CFI pointer to the next 16-bit quantity.
00159 *
00160 * @param   Ptr is the pointer to advance. Can be a pointer to any type.
00161 * @param   Interleave is the byte interleaving (based on part layout)
00162 *
00163 * @return
00164 *
00165 * Adjusted Ptr.
00166 *
00167 */
00168 #define XFL_CFI_ADVANCE_PTR16(Ptr, Interleave) \
00169     (Ptr = (void*)((Xuint32)Ptr + ((Interleave) << 1)))
00170
00171
00172 /***** Function Prototypes *****/
00173
00174 XStatus XFlashCFI_ReadCommon(XFlashGeometry *GeometryPtr,
00175                             XFlashProperties *PropertiesPtr);
00176
00177 #endif          /* end of protection macro */

```

flash/v1_00_a/src/xflash_geometry.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xflash_geometry.h,v 1.7 2002/04/19 16:19:42 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file flash/v1_00_a/src/xflash_geometry.h
00026 *
00027 * This is a helper component for XFlash. It contains the geometry information
00028 * for an XFlash instance with utilities to translate from absolute to block
00029 * coordinate systems.
00030 *
00031 * <b>Absolute coordinates</b>
00032 *
00033 * This coordinate system is simply an offset into the address space of the
00034 * flash instance.
00035 *
00036 * <b>Block coordinates</b>
00037 *
00038 * This coordinate system is dependent on the device's block geometry. All
00039 * flash devices are divisible by independent erase blocks which can be
00040 * addressed with this coordinate system. The coordinates are defined as:
00041 *
00042 * - BlockOffset - The offset within an erase block
```


00043 * - Block - An erase block
 00044 * - Region - An area containing erase blocks of the same size
 00045 *
 00046 *
 00047 * The picture below shows the differences between the coordinate systems.
 00048 * The sample part has two regions of three blocks each.

```
00049 * <pre>
```

	Absolute	Region	Block	Offset
	-----	-----	-----	-----
00052 * +-----+ 0 0 0 0				
00053 * 0 0 blocksizeA-1				
00054 * +-----+ 0 1 0				
00055 * 0 1 blocksizeA-1				
00056 * +-----+ 0 2 0				
00057 * 0 2 blocksizeA-1				
00058 * +=====+ 1 0 0				
00059 *				
00060 *				
00061 *				
00062 * 1 0 blocksizeB-1				
00063 * +-----+ 1 1 0				
00064 *				
00065 *				
00066 *				
00067 * 1 1 blocksizeB-1				
00068 * +-----+ 1 2 0				
00069 *				
00070 *				
00071 *				
00072 * 1 2 blocksizeB-1				
00073 * +-----+ 1 3 0				
00074 *				
00075 *				
00076 *				
00077 *				
00078 * +-----+ DevSize-1 1 3 blocksizeB-1				

```
00079 * </pre>
```

00080 *
 00081 * **Multiple devices**
 00082 *
 00083 * Some systems designs have more than one physical part wired in parallel
 00084 * on the data bus with each part appearing to be interleaved in the address
 00085 * space. The geometry takes these extra parts into consideration by
 00086 * doubling the sizes of blocks and regions.
 00087 *

```
00088 * <pre>
```

00089 * MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
00093 * 1.00a	rmm	03/15/02	Added XFL_LAYOUT_X16_X16_X4
00094 * 1.00a	rmm	07/16/01	First release

```
00095 * </pre>
```



```

00145                                     is 64-bits */
00146
00147 /*
00148  * LAYOUT constants are used to parse the XFL_LAYOUT_X* constants defined
00149  * above
00150  */
00151 #define XFL_LAYOUT_NUM_PARTS_MASK    0x000000FF
00152 #define XFL_LAYOUT_PART_MODE_MASK   0x0000FF00
00153 #define XFL_LAYOUT_PART_WIDTH_MASK  0x00FF0000
00154 #define XFL_LAYOUT_CFI_INTERL_MASK  0xFF000000
00155
00156 #define XFL_LAYOUT_NUM_PARTS_1      0x00000001
00157 #define XFL_LAYOUT_NUM_PARTS_2      0x00000002
00158 #define XFL_LAYOUT_NUM_PARTS_4      0x00000004
00159 #define XFL_LAYOUT_PART_MODE_8      0x00000100
00160 #define XFL_LAYOUT_PART_MODE_16     0x00000200
00161 #define XFL_LAYOUT_PART_WIDTH_8     0x00010000
00162 #define XFL_LAYOUT_PART_WIDTH_16    0x00020000
00163 #define XFL_LAYOUT_CFI_INTERL_1     0x01000000
00164 #define XFL_LAYOUT_CFI_INTERL_2     0x02000000
00165 #define XFL_LAYOUT_CFI_INTERL_4     0x04000000
00166 #define XFL_LAYOUT_CFI_INTERL_8     0x08000000
00167
00168 /***** Type Definitions *****/
00169
00170 /**
00171  * Flash geometry
00172  */
00173 typedef struct
00174 {
00175     Xuint32 BaseAddress;           /**< Base address of part(s) */
00176     Xuint32 MemoryLayout;         /**< How multiple parts are connected on
00177                                     the data bus. Choices are limited to
00178                                     XFL_LAYOUT_Xa_Xb_Xc constants */
00179     Xuint32 DeviceSize;           /**< Total device size in bytes */
00180     Xuint32 NumEraseRegions;      /**< Number of erase regions */
00181     Xuint16 NumBlocks;            /**< Total number of blocks in device */
00182
00183     struct
00184     {
00185         Xuint32 AbsoluteOffset;   /**< Offset within part where region begins
00186 */
00187         Xuint16 AbsoluteBlock;    /**< Block number where region begins */
00188         Xuint16 Number;           /**< Number of blocks in this region */
00189         Xuint32 Size;             /**< Size of the block in bytes */
00190     } EraseRegion[XFL_MAX_ERASE_REGIONS+1];
00191 } XFlashGeometry;
00192
00193 /***** Macros (Inline Functions) Definitions *****/
00194

```

```

00195 /*****
00196 /**
00197 *
00198 * Tests the given Region, Block, and Offset to verify they lie within the
00199 * address space defined by a geometry instance.
00200 *
00201 * @param GeometryPtr is the geometry instance that defines flash addressing
00202 * @param Region is the region to test
00203 * @param Block is the block to test
00204 * @param BlockOffset is the offset within block
00205 *
00206 * @return
00207 *
00208 * - 0 if Region, Block, & BlockOffset do not lie within the address space
00209 * described by GeometryPtr.
00210 * - 1 if Region, Block, & BlockOffset are within the address space
00211 *
00212 */
00213 #define XFL_GEOMETRY_IS_BLOCK_VALID(GeometryPtr, Region, Block, BlockOffset) \
00214 ((Region) < (GeometryPtr)->NumEraseRegions) && \
00215 ((Block) < (GeometryPtr)->EraseRegion[Region].Number) && \
00216 ((BlockOffset) < (GeometryPtr)->EraseRegion[Region].Size))
00217
00218
00219 /*****
00220 /**
00221 *
00222 * Tests the given absolute Offset to verify it lies within the bounds of the
00223 * address space defined by a geometry instance.
00224 *
00225 * @param GeometryPtr is the geometry instance that defines flash addressing
00226 * @param Offset is the offset to test
00227 *
00228 * @return
00229 *
00230 * - 0 if Offset do not lie within the address space described by GeometryPtr.
00231 * - 1 if Offset are within the address space
00232 *
00233 */
00234 #define XFL_GEOMETRY_IS_ABSOLUTE_VALID(GeometryPtr, Offset) \
00235 ((Offset) < (GeometryPtr)->DeviceSize)
00236
00237 /*****
00238 /**
00239 *
00240 * Calculates the number of blocks between the given coordinates and the end of
00241 * the device.
00242 *
00243 * @param GeometryPtr is the geometry instance that defines flash addressing
00244 * @param Region is the starting region
00245 * @param Block is the starting block.
00246 *

```

```

00247 * @return
00248 *
00249 * The number of blocks between Region/Block and the end of the device
00250 * (inclusive)
00251 *
00252 */
00253 #define XFL_GEOMETRY_BLOCKS_LEFT(GeometryPtr, Region, Block) \
00254 ((GeometryPtr)->NumBlocks - ((GeometryPtr)->EraseRegion[Region].AbsoluteBlock
+ Block))
00255
00256 /*****/
00257 /**
00258 *
00259 * Calculates the number of blocks between the given start and end coordinates.
00260 *
00261 * @param GeometryPtr is the geometry instance that defines flash addressing
00262 * @param StartRegion is the starting region
00263 * @param StartBlock is the starting block.
00264 * @param EndRegion is the ending region
00265 * @param EndBlock is the ending block.
00266 *
00267 * @return
00268 *
00269 * The number of blocks between start Region/Block and end Region/Block
00270 * (inclusive)
00271 *
00272 */
00273 #define XFL_GEOMETRY_BLOCK_DIFF(GeometryPtr, StartRegion, StartBlock,
EndRegion, EndBlock) \
00274 (((GeometryPtr)->EraseRegion[EndRegion].AbsoluteBlock + (EndBlock)) - \
00275 ((GeometryPtr)->EraseRegion[StartRegion].AbsoluteBlock + (StartBlock)) + 1)
00276
00277 /*****/
00278 /**
00279 *
00280 * Increments the given Region and Block to the next block address.
00281 *
00282 * @param GeometryPtr is the geometry instance that defines flash addressing
00283 * @param Region is the starting region.
00284 * @param Block is the starting block.
00285 *
00286 * @return
00287 *
00288 * Region parameter is incremented if the next block starts in a new region.
00289 * Block parameter is set to zero if the next block starts in a new region,
00290 * otherwise it is incremented by one.
00291 *
00292 */
00293 #define XFL_GEOMETRY_INCREMENT(GeometryPtr, Region, Block) \
00294 { \
00295     if ((GeometryPtr)->EraseRegion[Region].Number <= ++(Block)) \
00296     { \

```

```

00297         (Region)++;
00298         (Block) = 0;
00299     }
00300 }
00301
00302 /***** Function Prototypes *****/
00303 XStatus XFlashGeometry_ToBlock(XFlashGeometry *InstancePtr,
00304                               Xuint32 AbsoluteOffset,
00305                               Xuint16 *Region,
00306                               Xuint16 *Block,
00307                               Xuint32 *BlockOffset);
00308
00309 XStatus XFlashGeometry_ToAbsolute(XFlashGeometry *InstancePtr,
00310                                   Xuint16 Region,
00311                                   Xuint16 Block,
00312                                   Xuint32 BlockOffset,
00313                                   Xuint32 *AbsoluteOffsetPtr);
00314
00315 unsigned int XFlashGeometry_DiffBlocks(XFlashGeometry *GeometryPtr,
00316                                       Xuint16 StartRegion, Xuint16 StartBlock,
00317                                       Xuint16 EndRegion, Xuint16 EndBlock);
00318
00319 Xuint32 XFlashGeometry_ConvertLayout(Xuint8 NumParts, Xuint8 PartWidth,
00320                                     Xuint8 PartMode);
00321
00322 #endif          /* end of protection macro */

```

Generated on 29 May 2003 for Xilinx Device Drivers

flash/v1_00_a/src/xflash_geometry.h File Reference

Detailed Description

This is a helper component for XFlash. It contains the geometry information for an XFlash instance with utilities to translate from absolute to block coordinate systems.

Absolute coordinates

This coordinate system is simply an offset into the address space of the flash instance.

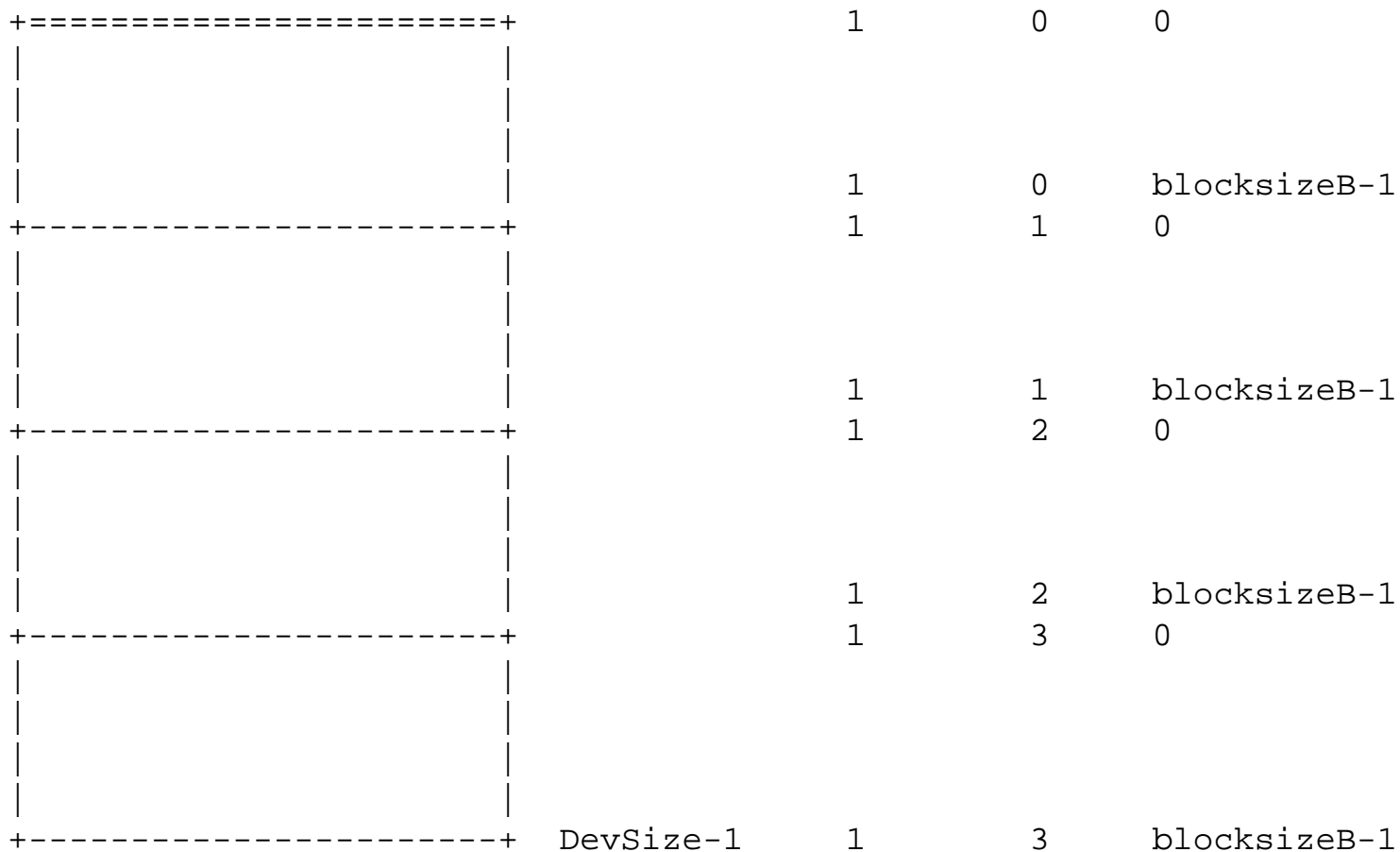
Block coordinates

This coordinate system is dependent on the device's block geometry. All flash devices are divisible by independent erase blocks which can be addressed with this coordinate system. The coordinates are defined as:

- BlockOffset - The offset within an erase block
- Block - An erase block
- Region - An area containing erase blocks of the same size

The picture below shows the differences between the coordinate systems. The sample part has two regions of three blocks each.

	Absolute	Region	Block	Offset
	-----	-----	-----	-----
+-----+	0	0	0	0
		0	0	blocksizeA-1
+-----+		0	1	0
		0	1	blocksizeA-1
+-----+		0	2	0
		0	2	blocksizeA-1



Multiple devices

Some systems designs have more than one physical part wired in parallel on the data bus with each part appearing to be interleaved in the address space. The geometry takes these extra parts into consideration by doubling the sizes of blocks and regions.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	03/15/02	Added XFL_LAYOUT_X16_X16_X4
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"
#include "xstatus.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XFlashGeometry
```

Defines

```
#define XFL_MAX_ERASE_REGIONS  
#define XFL_GEOMETRY_IS_BLOCK_VALID(GeometryPtr, Region, Block, BlockOffset)  
#define XFL_GEOMETRY_IS_ABSOLUTE_VALID(GeometryPtr, Offset)  
#define XFL_GEOMETRY_BLOCKS_LEFT(GeometryPtr, Region, Block)  
#define XFL_GEOMETRY_BLOCK_DIFF(GeometryPtr, StartRegion, StartBlock, EndRegion,  
    EndBlock)  
#define XFL_GEOMETRY_INCREMENT(GeometryPtr, Region, Block)
```

Functions

```
XStatus XFlashGeometry_ToBlock (XFlashGeometry *InstancePtr, Xuint32 AbsoluteOffset, Xuint16  
    *Region, Xuint16 *Block, Xuint32 *BlockOffset)  
XStatus XFlashGeometry_ToAbsolute (XFlashGeometry *InstancePtr, Xuint16 Region, Xuint16 Block,  
    Xuint32 BlockOffset, Xuint32 *AbsoluteOffsetPtr)  
Xuint32 XFlashGeometry_ConvertLayout (Xuint8 NumParts, Xuint8 PartWidth, Xuint8 PartMode)
```

Define Documentation

```
#define XFL_GEOMETRY_BLOCK_DIFF( GeometryPtr,  
    StartRegion,  
    StartBlock,  
    EndRegion,  
    EndBlock    )
```

Calculates the number of blocks between the given start and end coordinates.

Parameters:

- GeometryPtr* is the geometry instance that defines flash addressing
- StartRegion* is the starting region
- StartBlock* is the starting block.
- EndRegion* is the ending region
- EndBlock* is the ending block.

Returns:

The number of blocks between start Region/Block and end Region/Block (inclusive)

```
#define XFL_GEOMETRY_BLOCKS_LEFT( GeometryPtr,  
                                Region,  
                                Block      )
```

Calculates the number of blocks between the given coordinates and the end of the device.

Parameters:

- GeometryPtr* is the geometry instance that defines flash addressing
- Region* is the starting region
- Block* is the starting block.

Returns:

The number of blocks between Region/Block and the end of the device (inclusive)

```
#define XFL_GEOMETRY_INCREMENT( GeometryPtr,  
                                Region,  
                                Block      )
```

Increments the given Region and Block to the next block address.

Parameters:

GeometryPtr is the geometry instance that defines flash addressing

Region is the starting region.

Block is the starting block.

Returns:

Region parameter is incremented if the next block starts in a new region. Block parameter is set to zero if the next block starts in a new region, otherwise it is incremented by one.

```
#define XFL_GEOMETRY_IS_ABSOLUTE_VALID( GeometryPtr,  
                                         Offset      )
```

Tests the given absolute Offset to verify it lies within the bounds of the address space defined by a geometry instance.

Parameters:

GeometryPtr is the geometry instance that defines flash addressing

Offset is the offset to test

Returns:

- 0 if Offset do not lie within the address space described by GeometryPtr.
- 1 if Offset are within the address space

```
#define XFL_GEOMETRY_IS_BLOCK_VALID( GeometryPtr,  
                                     Region,  
                                     Block,  
                                     BlockOffset  )
```

Tests the given Region, Block, and Offset to verify they lie within the address space defined by a geometry instance.

Parameters:

GeometryPtr is the geometry instance that defines flash addressing
Region is the region to test
Block is the block to test
BlockOffset is the offset within block

Returns:

- 0 if Region, Block, & BlockOffset do not lie within the address space described by GeometryPtr.
- 1 if Region, Block, & BlockOffset are within the address space

#define XFL_MAX_ERASE_REGIONS

A block region is defined as a set of consecutive erase blocks of the same size. Most flash devices only have a handful of regions. If a part has more regions than defined by this constant, then the constant must be modified to accomodate the part. The minimum value of this constant is 1 and there is no maximum value. Note that increasing this value also increases the amount of memory used by the geometry structure approximately 12 bytes per increment.

Function Documentation

```
Xuint32 XFlashGeometry_ConvertLayout( Xuint8 NumParts,  
                                       Xuint8 PartWidth,  
                                       Xuint8 PartMode  
                                       )
```

Converts array layout into an XFL_LAYOUT_Xa_Xb_Xc constant. This function is typically called during initialization to convert ordinal values delivered by a system generator into the XFL constants which are optimized for use by the flash driver.

Parameters:

- NumParts* - Number of parts in the array.
- PartWidth* - Width of each part in bytes.
- PartMode* - Operation mode of each part in bytes.

Returns:

- XFL_LAYOUT_* - One of the supported layouts
- XNULL if a layout cannot be found that supports the given arguments

Note:

None.

```
XStatus XFlashGeometry_ToAbsolute( XFlashGeometry * InstancePtr,  
                                   Xuint16          Region,  
                                   Xuint16          Block,  
                                   Xuint32          BlockOffset,  
                                   Xuint32 *       AbsoluteOffsetPtr  
                                   )
```

Converts block coordinates to a part offset. Region, Block, & BlockOffset are converted to PartOffset

Parameters:

- InstancePtr* is the pointer to the XFlash instance to be worked on.
- Region* is the erase region the physical address appears in.
- Block* is the block within Region the physical address appears in.
- BlockOffset* is the offset within Block where the physical address appears.
- AbsoluteOffsetPtr* is the returned offset value

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the block coordinates are invalid.

Note:

None.

```
XStatus XFlashGeometry_ToBlock( XFlashGeometry * InstancePtr,  
                                Xuint32 AbsoluteOffset,  
                                Xuint16 * RegionPtr,  
                                Xuint16 * BlockPtr,  
                                Xuint32 * BlockOffsetPtr  
                                )
```

Converts part offset block coordinates. PartOffset is converted to Region, Block, & BlockOffset

Parameters:

- InstancePtr* is the pointer to the **XFlashGeometry** instance to be worked on.
- AbsoluteOffset* is the offset within part to find block coordinates for.
- RegionPtr* is the the region that corresponds to AbsoluteOffset. This is a return parameter.
- BlockPtr* is the the block within Region that corresponds to AbsoluteOffset. This is a return parameter.
- BlockOffsetPtr* is the the offset within Block that corresponds to AbsoluteOffset. This is a return parameter.

Returns:

- o XST_SUCCESS if successful.
- o XST_FLASH_ADDRESS_ERROR if the block coordinates are invalid.

Note:

None.

XFlashGeometry Struct Reference

```
#include <xflash_geometry.h>
```

Detailed Description

Flash geometry

Data Fields

[Xuint32 BaseAddress](#)
[Xuint32 MemoryLayout](#)
[Xuint32 DeviceSize](#)
[Xuint32 NumEraseRegions](#)
[Xuint16 NumBlocks](#)
[Xuint32 AbsoluteOffset](#)
[Xuint16 AbsoluteBlock](#)
[Xuint16 Number](#)
[Xuint32 Size](#)

Field Documentation

[Xuint16 XFlashGeometry::AbsoluteBlock](#)

Block number where region begins

[Xuint32 XFlashGeometry::AbsoluteOffset](#)

Offset within part where region begins

Xuint32 XFlashGeometry::BaseAddress

Base address of part(s)

Xuint32 XFlashGeometry::DeviceSize

Total device size in bytes

Xuint32 XFlashGeometry::MemoryLayout

How multiple parts are connected on the data bus. Choices are limited to XFL_LAYOUT_Xa_Xb_Xc constants

Xuint16 XFlashGeometry::Number

Number of blocks in this region

Xuint16 XFlashGeometry::NumBlocks

Total number of blocks in device

Xuint32 XFlashGeometry::NumEraseRegions

Number of erase regions

Xuint32 XFlashGeometry::Size

Size of the block in bytes

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_geometry.h](#)

flash/v1_00_a/src/xflash_geometry.c File Reference

Detailed Description

This module implements the helper component [XFlashGeometry](#).

The helper component is responsible for containing the geometry information for the flash part(s) and for converting from an absolute part offset to region/block/blockOffset coordinates.

[XFlashGeometry](#) describes the geometry of the entire instance, not the individual parts of that instance. For example, if the user's board architecture uses two 16-bit parts in parallel for a 32-bit data path, then the size of erase blocks and device sizes are multiplied by a factor of two.

Note:

This helper component is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, or cache control management must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "xflash.h"  
#include "xflash_geometry.h"
```

Functions

XStatus XFlashGeometry_ToAbsolute (**XFlashGeometry** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** BlockOffset, **Xuint32** *AbsoluteOffsetPtr)

XStatus XFlashGeometry_ToBlock (**XFlashGeometry** *InstancePtr, **Xuint32** AbsoluteOffset, **Xuint16** *RegionPtr, **Xuint16** *BlockPtr, **Xuint32** *BlockOffsetPtr)

Xuint32 XFlashGeometry_ConvertLayout (**Xuint8** NumParts, **Xuint8** PartWidth, **Xuint8** PartMode)

Function Documentation

```
Xuint32 XFlashGeometry_ConvertLayout( Xuint8 NumParts,  
                                     Xuint8 PartWidth,  
                                     Xuint8 PartMode  
                                     )
```

Converts array layout into an XFL_LAYOUT_Xa_Xb_Xc constant. This function is typically called during initialization to convert ordinal values delivered by a system generator into the XFL constants which are optimized for use by the flash driver.

Parameters:

- NumParts* - Number of parts in the array.
- PartWidth* - Width of each part in bytes.
- PartMode* - Operation mode of each part in bytes.

Returns:

- XFL_LAYOUT_* - One of the supported layouts
- XNULL if a layout cannot be found that supports the given arguments

Note:

None.

```

XStatus XFlashGeometry_ToAbsolute( XFlashGeometry * InstancePtr,
                                     Xuint16           Region,
                                     Xuint16           Block,
                                     Xuint32           BlockOffset,
                                     Xuint32 *         AbsoluteOffsetPtr
                                     )

```

Converts block coordinates to a part offset. Region, Block, & BlockOffset are converted to PartOffset

Parameters:

InstancePtr is the pointer to the XFlash instance to be worked on.
Region is the erase region the physical address appears in.
Block is the block within Region the physical address appears in.
BlockOffset is the offset within Block where the physical address appears.
AbsoluteOffsetPtr is the returned offset value

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the block coordinates are invalid.

Note:

None.

```

XStatus XFlashGeometry_ToBlock( XFlashGeometry * InstancePtr,
                                  Xuint32           AbsoluteOffset,
                                  Xuint16 *         RegionPtr,
                                  Xuint16 *         BlockPtr,
                                  Xuint32 *         BlockOffsetPtr
                                  )

```

Converts part offset block coordinates. PartOffset is converted to Region, Block, & BlockOffset

Parameters:

InstancePtr is the pointer to the **XFlashGeometry** instance to be worked on.

AbsoluteOffset is the offset within part to find block coordinates for.

RegionPtr is the the region that corresponds to AbsoluteOffset. This is a return parameter.

BlockPtr is the the block within Region that corresponds to AbsoluteOffset. This is a return parameter.

BlockOffsetPtr is the the offset within Block that corresponds to AbsoluteOffset. This is a return parameter.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_ADDRESS_ERROR if the block coordinates are invalid.

Note:

None.

flash/v1_00_a/src/xflash_properties.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xflash_properties.h,v 1.4 2002/03/09 00:14:46 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file flash/v1_00_a/src/xflash_properties.h
00026 *
00027 * This is a helper component for XFlash. It contains various datum
00028 * common to flash devices most of which can be derived from the CFI query.
00029 *
00030 * @note
00031 *
00032 * There is no implementation file with this component.
00033 *
00034 * <pre>
00035 * MODIFICATION HISTORY:
00036 *
00037 * Ver   Who   Date       Changes
00038 * ----  ---  -
00039 * 1.00a rmm   07/16/01 First release
00040 * </pre>
00041 *
00042 *****/
```

```

00043
00044 #ifndef XFLASH_PROPERTIES_H /* prevent circular inclusions */
00045 #define XFLASH_PROPERTIES_H /* by using protection macros */
00046
00047 /***** Include Files *****/
00048 #include "xbasic_types.h"
00049 #include "xstatus.h"
00050
00051 /***** Constant Definitions *****/
00052
00053
00054 /***** Type Definitions *****/
00055
00056 /**
00057  * Flash timing
00058  */
00059 typedef struct
00060 {
00061     Xuint16 WriteSingle_Us;           /**< Time to program a single word unit
00062                                     Units are in microseconds */
00063     Xuint16 WriteBuffer_Us;         /**< Time to program the contents of the
00064                                     write buffer. Units are in microseconds
00065                                     If the part does not support write
00066                                     buffers, then this value should be
00067                                     zero */
00068     Xuint16 EraseBlock_Ms;          /**< Time to erase a single block
00069                                     Units are in milliseconds */
00070     Xuint16 EraseChip_Ms;           /**< Time to perform a chip erase
00071                                     Units are in milliseconds */
00072 } XFlashTiming;
00073
00074 /**
00075  * Flash identification
00076  */
00077 typedef struct
00078 {
00079     Xuint8  ManufacturerID;          /**< Manufacturer of parts */
00080     Xuint8  DeviceID;                /**< Part number of manufacturer */
00081     Xuint16 CommandSet;              /**< Command algorithm used by part.
Choices
00082                                     are defined in XFL_CMDSET constants */
00083 } XFlashPartID;
00084
00085 /**
00086  * Programming parameters
00087  */
00088 typedef struct
00089 {
00090     Xuint32 WriteBufferSize;         /**< Number of bytes that can be
programmed
00091                                     at once */

```

```

00092     Xuint32 WriteBufferAlignmentMask;/**< Alignment of the write buffer */
00093     Xuint32 EraseQueueSize;           /**< Number of erase blocks that can be
00094                                         queued up at once */
00095 } XFlashProgCap;
00096
00097
00098 /**
00099  * Consolidated parameters
00100  */
00101 typedef struct
00102 {
00103     XFlashPartID PartID;                /**< Uniquely identifies the part */
00104     XFlashTiming TimeTypical;          /**< Typical timing data */
00105     XFlashTiming TimeMax;              /**< Worst case timing data */
00106     XFlashProgCap ProgCap;             /**< Programming capabilities */
00107 } XFlashProperties;
00108
00109 /***** Function Prototypes *****/
00110
00111 #endif          /* end of protection macro */

```

Generated on 29 May 2003 for Xilinx Device Drivers

flash/v1_00_a/src/xflash_properties.h

File Reference

Detailed Description

This is a helper component for XFlash. It contains various datum common to flash devices most of which can be derived from the CFI query.

Note:

There is no implementation file with this component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XFlashPartID  
struct XFlashProgCap  
struct XFlashProperties
```


struct **XFlashTiming**

Generated on 29 May 2003 for Xilinx Device Drivers

XFlashPartID Struct Reference

```
#include <xflash_properties.h>
```

Detailed Description

Flash identification

Data Fields

Xuint8 ManufacturerID

Xuint8 DeviceID

Xuint16 CommandSet

Field Documentation

Xuint16 XFlashPartID::CommandSet

Command algorithm used by part. Choices are defined in XFL_CMDSET constants

Xuint8 XFlashPartID::DeviceID

Part number of manufacturer

Xuint8 XFlashPartID::ManufacturerID

Manufacturer of parts

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_properties.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XFlashProgCap Struct Reference

```
#include <xflash_properties.h>
```

Detailed Description

Programming parameters

Data Fields

[Xuint32 WriteBufferSize](#)

[Xuint32 WriteBufferAlignmentMask](#)

[Xuint32 EraseQueueSize](#)

Field Documentation

[Xuint32 XFlashProgCap::EraseQueueSize](#)

Number of erase blocks that can be queued up at once

[Xuint32 XFlashProgCap::WriteBufferAlignmentMask](#)

Alignment of the write buffer

[Xuint32 XFlashProgCap::WriteBufferSize](#)

Number of bytes that can be programmed at once

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_properties.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XFlashProperties Struct Reference

```
#include <xflash_properties.h>
```

Detailed Description

Consolidated parameters

Data Fields

[XFlashPartID](#) PartID

[XFlashTiming](#) TimeTypical

[XFlashTiming](#) TimeMax

[XFlashProgCap](#) ProgCap

Field Documentation

[XFlashPartID](#) XFlashProperties::PartID

Uniquely identifies the part

[XFlashProgCap](#) XFlashProperties::ProgCap

Programming capabilities

[XFlashTiming](#) XFlashProperties::TimeMax

Worst case timing data

XFlashTiming XFlashProperties::TimeTypical

Typical timing data

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_properties.h](#)
-

Generated on 29 May 2003 for Xilinx Device Drivers

XFlashTiming Struct Reference

```
#include <xflash_properties.h>
```

Detailed Description

Flash timing

Data Fields

[Xuint16 WriteSingle_Us](#)

[Xuint16 WriteBuffer_Us](#)

[Xuint16 EraseBlock_Ms](#)

[Xuint16 EraseChip_Ms](#)

Field Documentation

[Xuint16 XFlashTiming::EraseBlock_Ms](#)

Time to erase a single block Units are in milliseconds

[Xuint16 XFlashTiming::EraseChip_Ms](#)

Time to perform a chip erase Units are in milliseconds

[Xuint16 XFlashTiming::WriteBuffer_Us](#)

Time to program the contents of the write buffer. Units are in microseconds If the part does not support write buffers, then this value should be zero

Xuint16 XFlashTiming::WriteSingle_Us

Time to program a single word unit Units are in microseconds

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash_properties.h](#)
-

Generated on 29 May 2003 for Xilinx Device Drivers

flash/v1_00_a/src/xflash_cfi.c File Reference

Detailed Description

This module implements the helper component XFlashCFI.

The helper component is responsible for retrieval and translation of CFI data from a compliant flash device. CFI contains data that defines part geometry, write/erase timing, and programming data.

Data is retrieved using macros defined in this component's header file. The macros simplify data extraction because they have been written to take into account the layout of parts on the data bus. To the driver, CFI data appears as if it were always being read from a single 8-bit part (XFL_LAYOUT_X8_X8_X1) Otherwise, the retrieval code would have to contend with all the formats illustrated below. The illustration shows how the first three bytes of the CFI query data "QRY" appear in flash memory space for various part layouts.

```
                Byte Offset (big-Endian)
                0123456789ABCDEF
                -----
XFL_LAYOUT_X16_X16_X1   Q R Y
XFL_LAYOUT_X16_X16_X2   Q Q R R Y Y
```

Where the holes between Q, R, and Y are NULL (all bits 0)

Note:

This helper component is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads, mutual exclusion, virtual memory, or cache control management must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	03/15/02	Added 64 bit access
1.00a	rmm	07/16/01	First release

```
#include "xbasic_types.h"  
#include "xflash.h"  
#include "xflash_cfi.h"
```

Functions

XStatus XFlashCFI_ReadCommon (**XFlashGeometry** *GeometryPtr, **XFlashProperties** *PropertiesPtr)

Function Documentation

```
XStatus XFlashCFI_ReadCommon( XFlashGeometry * GeometryPtr,  
                               XFlashProperties * PropertiesPtr  
                               )
```

Retrieves the standard CFI data from the part(s), interpret the data, and update the provided geometry and properties structures.

Extended CFI data is part specific and ignored here. This data must be read by the specific part component driver.

Parameters:

GeometryPtr is an input/output parameter. This function expects the BaseAddress and MemoryLayout attributes to be correctly initialized. All other attributes of this structure will be setup using translated CFI data read from the part.

PropertiesPtr is an output parameter. Timing, identification, and programming CFI data will be translated and written to this structure.

Returns:

- XST_SUCCESS if successful.
- XST_FLASH_CFI_QUERY_ERROR if an error occurred interpreting the data.
- XST_FLASH_PART_NOT_SUPPORTED if invalid Layout parameter

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

XFlash_Config Struct Reference

```
#include <xflash.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId
Xuint32 BaseAddr
 Xuint8 NumParts
 Xuint8 PartWidth
 Xuint8 PartMode

Field Documentation

Xuint32 XFlash_Config::BaseAddr

Base address of array

Xuint16 XFlash_Config::DeviceId

Unique ID of device

Xuint8 XFlash_Config::NumParts

Number of parts in the array

Xuint8 XFlash_Config::PartMode

Operation mode of each part in bytes

Xuint8 XFlash_Config::PartWidth

Width of each part in bytes

The documentation for this struct was generated from the following file:

- [flash/v1_00_a/src/xflash.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XFlashTag Struct Reference

```
#include <xflash.h>
```

Detailed Description

The XFlash driver instance data. The user is required to allocate a variable of this type for every flash device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `flash/v1_00_a/src/xflash.h`
-

flash/v1_00_a/src/xflash_g.c File Reference

Detailed Description

This file contains a table that specifies the configuration of devices in the system. In addition, there is a lookup function used by the driver to access its configuration information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	01/18/01	First release
1.00a	rpm	05/05/02	Added include of xparameters.h

```
#include "xflash.h"  
#include "xparameters.h"
```

Variables

[XFlash_Config](#) [XFlash_ConfigTable](#) [XPAR_XFLASH_NUM_INSTANCES]

Variable Documentation

[XFlash_Config](#) [XFlash_ConfigTable](#)[XPAR_XFLASH_NUM_INSTANCES]

This table contains configuration information for each flash device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

XStatus XFlashIntel_Read (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)

XStatus XFlashIntel_Write (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)

XStatus XFlashIntel_WriteSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_WriteResume (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_Erase (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlashIntel_EraseSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_EraseResume (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_Lock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlashIntel_Unlock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlashIntel_GetStatus (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_ReadBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)

XStatus XFlashIntel_WriteBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)

XStatus XFlashIntel_WriteBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)

XStatus XFlashIntel_WriteBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint32** Offset)

XStatus XFlashIntel_EraseBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)

XStatus XFlashIntel_EraseBlockSuspend (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)

XStatus XFlashIntel_EraseBlockResume (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)

XStatus XFlashIntel_LockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)

XStatus XFlashIntel_UnlockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)

XStatus XFlashIntel_GetBlockStatus (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)

XStatus XFlashIntel_EraseChip (**XFlash** *InstancePtr)

Function Documentation

XStatus XFlashIntel_DeviceControl(**XFlash** * *InstancePtr*,
Xuint32 *Command*,
Xuint32 *Param*,
Xuint32 * *ReturnPtr*
)

See the base component for a description of this function, its return values, and arguments.

Note:

Intel specific commands:

Command: XFL_INTEL_DEVCTL_SET_RYBY

Description:

Set the mode of the RYBY signal.

Param:

One of XFL_INTEL_RYBY_PULSE_OFF, XFL_INTEL_RYBY_PULSE_ON_ERASE, XFL_INTEL_RYBY_PULSE_ON_PROG, XFL_INTEL_RYBY_PULSE_ON_ERASE_PROG

Return:

None

Command: XFL_INTEL_DEVCTL_GET_LAST_ERROR

Description:

Retrieve the last error condition. The data is in the form of the status register(s) read from the device(s) at the time the error was detected. The registers are formatted verbatim as they are seen on the data bus.

Param:

None

Return:

The contents of the Status registers at the time the last error was detected.

```
XStatus XFlashIntel_Erase( XFlash * InstancePtr,  
                          Xuint32 Offset,  
                          Xuint32 Bytes  
                          )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_EraseBlock( XFlash * InstancePtr,  
                                Xuint16 Region,  
                                Xuint16 Block,  
                                Xuint16 NumBlocks  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_EraseBlockResume( XFlash * InstancePtr,  
                                       Xuint16 Region,  
                                       Xuint16 Block  
                                       )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region & Block parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_EraseBlockSuspend( XFlash * InstancePtr,  
                                        Xuint16 Region,  
                                        Xuint16 Block  
                                        )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region & Block parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_EraseChip( XFlash * InstancePtr)
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_EraseResume( XFlash * InstancePtr,  
                                Xuint32 Offset  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_EraseSuspend( XFlash * InstancePtr,  
                                  Xuint32 Offset  
                                  )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_GetBlockStatus( XFlash * InstancePtr,  
                                    Xuint16 Region,  
                                    Xuint16 Block  
                                    )
```

See the base component for a description of this function, its return values, and arguments.

Note:

The Region & Block parameters are not used because the device's status register appears at every addressible location.

XFlashGeometry* XFlashIntel_GetGeometry(**XFlash** * *InstancePtr*)

See the base component for a description of this function, its return values, and arguments.

Note:

None.

Xuint32 XFlashIntel_GetOptions(**XFlash** * *InstancePtr*)

See the base component for a description of this function, its return values, and arguments.

Note:

None.

XFlashProperties* XFlashIntel_GetProperties(**XFlash** * *InstancePtr*)

See the base component for a description of this function, its return values, and arguments.

Note:

None.

XStatus XFlashIntel_GetStatus(**XFlash** * *InstancePtr*, **Xuint32** *Offset*)

See the base component for a description of this function, its return values, and arguments.

Note:

The Offset parameter is not used because the device's status register appears at every addressible location.

XStatus XFlashIntel_Initialize(**XFlash** * *InstancePtr*)

See the base component for a description of this function, its return values, and arguments.

Parameters:

InstancePtr is a pointer to the flash instance to be worked on.

Returns:

- XST_SUCCESS if successful
- XST_FLASH_PART_NOT_SUPPORTED if the part is not supported

Note:

Two geometry attributes MUST be defined prior to invoking this function:

- BaseAddress
- MemoryLayout

```
XStatus XFlashIntel_Lock( XFlash * InstancePtr,  
                          Xuint32 Offset,  
                          Xuint32 Bytes  
                          )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_LockBlock( XFlash * InstancePtr,  
                               Xuint16 Region,  
                               Xuint16 Block,  
                               Xuint16 NumBlocks  
                               )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.


```
XStatus XFlashIntel_Read( XFlash * InstancePtr,  
                          Xuint32 Offset,  
                          Xuint32 Bytes,  
                          void * DestPtr  
                          )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_ReadBlock( XFlash * InstancePtr,  
                               Xuint16 Region,  
                               Xuint16 Block,  
                               Xuint32 Offset,  
                               Xuint32 Bytes,  
                               void * DestPtr  
                               )
```

See the base component for a description of this function, its return values, and arguments.

Note:

The part is assumed to be in read-array mode.

```
XStatus XFlashIntel_Reset( XFlash * InstancePtr)
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_SelfTest( XFlash * InstancePtr)
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_SetOptions( XFlash * InstancePtr,  
                                Xuint32 OptionsFlag  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_Unlock( XFlash * InstancePtr,  
                            Xuint32 Offset,  
                            Xuint32 Bytes  
                            )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_UnlockBlock( XFlash * InstancePtr,  
                                  Xuint16 Region,  
                                  Xuint16 Block,  
                                  Xuint16 NumBlocks  
                                  )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_Write( XFlash * InstancePtr,  
                           Xuint32 Offset,  
                           Xuint32 Bytes,  
                           void * SrcPtr  
                           )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_WriteBlock( XFlash * InstancePtr,  
                                Xuint16 Region,  
                                Xuint16 Block,  
                                Xuint32 Offset,  
                                Xuint32 Bytes,  
                                void * SrcPtr  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_WriteBlockResume( XFlash * InstancePtr,  
                                       Xuint16 Region,  
                                       Xuint16 Block,  
                                       Xuint32 Offset  
                                       )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region, Block, & Offset parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_WriteBlockSuspend( XFlash * InstancePtr,  
                                       Xuint16 Region,  
                                       Xuint16 Block,  
                                       Xuint32 Offset  
                                       )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region, Block, & Offset parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_WriteResume( XFlash * InstancePtr,  
                                Xuint32 Offset  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_WriteSuspend( XFlash * InstancePtr,  
                                  Xuint32 Offset  
                                  )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

flash/v1_00_a/src/xflash_intel.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xflash_intel.h,v 1.3 2002/03/06 15:25:08 whittle Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file flash/v1_00_a/src/xflash_intel.h
00026 *
00027 * This is an Intel specific Flash memory component driver for CFI enabled
00028 * parts.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date        Changes
00034 * -----
00035 * 1.00a rmm    07/16/01 First release
00036 * </pre>
00037 *
00038 *****/
00039
00040 #ifndef XFLASH_INTEL_H /* prevent circular inclusions */
00041 #define XFLASH_INTEL_H /* by using protection macros */
00042
```

```

00043 /***** Include Files *****/
00044
00045 #include "xbasic_types.h"
00046 #include "xflash.h"
00047
00048 /***** Constant Definitions *****/
00049
00050 /*
00051  * XFlash_DeviceControl command list
00052  */
00053 #define XFL_INTEL_DEVCTL_SET_RYBY          0 /* Set RYBY pin mode */
00054 #define XFL_INTEL_DEVCTL_GET_LAST_ERROR  1 /* Retrieve the last error data */
00055
00056 /*
00057  * RYBY Options
00058  * These options control the RYBY signal. They can be accessed with the
00059  * XFL_INTEL_DEVCTL_SET_RYBY command.
00060  */
00061 #define XFL_INTEL_RYBY_PULSE_OFF          0 /* Do not pulse */
00062 #define XFL_INTEL_RYBY_PULSE_ON_ERASE    1 /* Pulse on erase complete
00063                                           only */
00064 #define XFL_INTEL_RYBY_PULSE_ON_PROG     2 /* Pulse on program complete
00065                                           only */
00066 #define XFL_INTEL_RYBY_PULSE_ON_ERASE_PROG 3 /* Pulse on erase & program
00067                                           complete */
00068
00069 /*
00070  * Status register bit definitions. Use these bitmaps to decipher the
00071  * return value of the XFL_INTEL_DEVCTL_GET_LAST_ERROR command.
00072  */
00073 #define XFL_INTEL_SR_WSM_READY            0x80
00074 #define XFL_INTEL_SR_ERASE_SUSPENDED     0x40
00075 #define XFL_INTEL_SR_ERASE_OR_UNLOCK_ERROR 0x20
00076 #define XFL_INTEL_SR_PROG_OR_LOCK_ERROR  0x10
00077 #define XFL_INTEL_SR_VOLTAGE_ERROR       0x08
00078 #define XFL_INTEL_SR_WRITE_SUSPENDED    0x04
00079 #define XFL_INTEL_SR_BLOCK_LOCKED_ERROR  0x02
00080
00081 /*
00082  * Extended capabilities list
00083  * These bits can be read with the XFL_INTEL_DEVCTL_GET_SUPPORT1 command.
00084  *
00085  * XFL_INTEL_SUPPORT1_CHIP_ERASE - The part supports the chip erase command.
00086  *
00087  * XFL_INTEL_SUPPORT1_SUSPEND_ERASE - The part supports suspension and
00088  *   resumption of an erase operation.
00089  *
00090  * XFL_INTEL_SUPPORT1_SUSPEND_PROG - The part supports suspension and
00091  *   resumption of a programming operation.
00092  *
00093  * XFL_INTEL_SUPPORT1_LEGACY_LOCK - The part supports legacy lock/unlock.
00094  *

```

```

00095 * XFL_INTEL_SUPPORT1_QUEUED_ERASE - The part supports the queueing up of
00096 *   erase blocks.
00097 *
00098 * XFL_INTEL_SUPPORT1_INSTANT_LOCK - The part supports instant individual
00099 *   block locking.
00100 *
00101 * XFL_INTEL_SUPPORT1_PROTECTION_BIT - The part supports protection bits.
00102 *
00103 * XFL_INTEL_SUPPORT1_PAGE_MODE_READ - The part supports page-mode reads.
00104 *
00105 * XFL_INTEL_SUPPORT1_SYNC_READ - The part supports synchronous reads.
00106 *
00107 */
00108 #define XFL_INTEL_SUPPORT1_CHIP_ERASE          0x80000000UL
00109 #define XFL_INTEL_SUPPORT1_SUSPEND_ERASE      0x40000000UL
00110 #define XFL_INTEL_SUPPORT1_SUSPEND_PROG      0x20000000UL
00111 #define XFL_INTEL_SUPPORT1_LEGACY_LOCK       0x10000000UL
00112 #define XFL_INTEL_SUPPORT1_QUEUED_ERASE      0x08000000UL
00113 #define XFL_INTEL_SUPPORT1_INSTANT_LOCK      0x04000000UL
00114 #define XFL_INTEL_SUPPORT1_PROTECTION_BIT    0x02000000UL
00115 #define XFL_INTEL_SUPPORT1_PAGE_MODE_READ    0x01000000UL
00116 #define XFL_INTEL_SUPPORT1_SYNC_READ        0x00800000UL
00117
00118 /*
00119 * Suspension capabilities list
00120 *
00121 * XFL_INTEL_PROG_AFTER_ERASE_SUSPEND - The part supports programming
00122 *   after suspending an erase operation
00123 */
00124 #define XFL_INTEL_SUSPEND_SUPPORT_PROG_AFTER_ERASE 0x00000001UL
00125
00126 /***** Type Definitions *****/
00127
00128
00129 /***** Macros (Inline Functions) Definitions *****/
00130
00131
00132 /***** Function Prototypes *****/
00133
00134 XStatus XFlashIntel_Initialize(XFlash *InstancePtr);
00135 XStatus XFlashIntel_SelfTest(XFlash *InstancePtr);
00136 XStatus XFlashIntel_Reset(XFlash *InstancePtr);
00137 XStatus XFlashIntel_SetOptions(XFlash *InstancePtr, Xuint32 OptionsFlag);
00138 Xuint32 XFlashIntel_GetOptions(XFlash *InstancePtr);
00139 XFlashProperties* XFlashIntel_GetProperties(XFlash *InstancePtr);
00140 XFlashGeometry* XFlashIntel_GetGeometry(XFlash *InstancePtr);
00141 XStatus XFlashIntel_DeviceControl(XFlash *InstancePtr, Xuint32 Command,
00142                                   Xuint32 Param, Xuint32 *ReturnPtr);
00143
00144 /*
00145 * Non-geometry aware API

```



```
00191 XStatus XFlashIntel_UnlockBlock(XFlash *InstancePtr, Xuint16 Region, Xuint16
Block,
00192         Xuint16 NumBlocks);
00193
00194 XStatus XFlashIntel_GetBlockStatus(XFlash *InstancePtr, Xuint16 Region,
00195         Xuint16 Block);
00196
00197 /*
00198  * Other commands
00199  */
00200 XStatus XFlashIntel_EraseChip(XFlash *InstancePtr);
00201
00202 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

flash/v1_00_a/src/xflash_intel.c File Reference

Detailed Description

The implementation of the Intel CFI Version of the XFlash component.

This module utilizes the XFlash base component, whose attributes have been defined mostly from a CFI data query. This data is used to define the geometry of the part(s), timeout values for write & erase operations, and optional features.

Note:

- Special consideration has to be given to varying data bus widths. To boost performance, multiple devices in parallel on the data bus are accessed in parallel. Therefore to reduce complexity and increase performance, many local primitive functions are duplicated with the only difference being the width of writes to the devices.

Even with the performance boosting optimizations, the overhead associated with this component is rather high due to the general purpose nature of its design.

Flash block erasing is a time consuming operation with nearly all latency occurring due to the devices' themselves. It takes on the order of 1 second to erase each block.

Writes by comparison are much quicker so driver overhead becomes an issue. The write algorithm has been optimized for bulk data programming and should provide relatively better performance.

- The code/comments refers to WSM frequently. This stands for Write State Machine. WSM is the internal programming engine of the devices.
- This driver and the underlying Intel flash memory does not allow re- programming while code is executing from the same memory.
- If hardware is flakey or fails, then this driver could hang a thread of execution.
- This module has some dependencies on whether it is being unit tested. These areas are noted with conditional compilation based on whether XENV_UNITTEST is defined. This is required because unit testing occurs without real flash devices.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	07/16/01	First release
1.00a	rmm	03/14/02	Added 64 bit array support
1.00a	rmm	05/13/03	Fixed diab compiler warnings relating to asserts.

```
#include "xflash_intel.h"  
#include "xflash_intel_1.h"  
#include "xflash_cfi.h"  
#include "xflash_geometry.h"  
#include "xenv.h"
```

Data Structures

```
union StatReg  
struct XFlashVendorData_IntelTag
```

Functions

```
XStatus XFlashIntel_Initialize (XFlash *InstancePtr)  
XStatus XFlashIntel_ReadBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block,  
    Xuint32 Offset, Xuint32 Bytes, void *DestPtr)  
XStatus XFlashIntel_WriteBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block,  
    Xuint32 Offset, Xuint32 Bytes, void *SrcPtr)  
XStatus XFlashIntel_WriteBlockSuspend (XFlash *InstancePtr, Xuint16 Region, Xuint16  
    Block, Xuint32 Offset)  
XStatus XFlashIntel_WriteBlockResume (XFlash *InstancePtr, Xuint16 Region, Xuint16  
    Block, Xuint32 Offset)  
XStatus XFlashIntel_EraseBlock (XFlash *InstancePtr, Xuint16 Region, Xuint16 Block,  
    Xuint16 NumBlocks)  
XStatus XFlashIntel_EraseBlockSuspend (XFlash *InstancePtr, Xuint16 Region, Xuint16  
    Block)  
XStatus XFlashIntel_EraseBlockResume (XFlash *InstancePtr, Xuint16 Region, Xuint16  
    Block)
```

XStatus XFlashIntel_LockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)

XStatus XFlashIntel_UnlockBlock (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block, **Xuint16** NumBlocks)

XStatus XFlashIntel_GetBlockStatus (**XFlash** *InstancePtr, **Xuint16** Region, **Xuint16** Block)

XStatus XFlashIntel_Read (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *DestPtr)

XStatus XFlashIntel_Write (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes, void *SrcPtr)

XStatus XFlashIntel_WriteSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_WriteResume (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_Erase (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlashIntel_EraseSuspend (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_EraseResume (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_Lock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlashIntel_Unlock (**XFlash** *InstancePtr, **Xuint32** Offset, **Xuint32** Bytes)

XStatus XFlashIntel_GetStatus (**XFlash** *InstancePtr, **Xuint32** Offset)

XStatus XFlashIntel_EraseChip (**XFlash** *InstancePtr)

XStatus XFlashIntel_SelfTest (**XFlash** *InstancePtr)

XStatus XFlashIntel_Reset (**XFlash** *InstancePtr)

XStatus XFlashIntel_SetOptions (**XFlash** *InstancePtr, **Xuint32** OptionsFlag)

Xuint32 XFlashIntel_GetOptions (**XFlash** *InstancePtr)

XFlashGeometry * **XFlashIntel_GetGeometry** (**XFlash** *InstancePtr)

XFlashProperties * **XFlashIntel_GetProperties** (**XFlash** *InstancePtr)

XStatus XFlashIntel_DeviceControl (**XFlash** *InstancePtr, **Xuint32** Command, **Xuint32** Param, **Xuint32** *ReturnPtr)

Function Documentation

```
XStatus XFlashIntel_DeviceControl( XFlash * InstancePtr,  
                                   Xuint32 Command,  
                                   Xuint32 Param,  
                                   Xuint32 * ReturnPtr  
                                   )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Intel specific commands:

Command: XFL_INTEL_DEVCTL_SET_RYBY

Description:

Set the mode of the RYBY signal.

Param:

One of XFL_INTEL_RYBY_PULSE_OFF, XFL_INTEL_RYBY_PULSE_ON_ERASE, XFL_INTEL_RYBY_PULSE_ON_PROG, XFL_INTEL_RYBY_PULSE_ON_ERASE_PROG

Return:

None

Command: XFL_INTEL_DEVCTL_GET_LAST_ERROR

Description:

Retrieve the last error condition. The data is in the form of the status register(s) read from the device(s) at the time the error was detected. The registers are formatted verbatim as they are seen on the data bus.

Param:

None

Return:

The contents of the Status registers at the time the last error was detected.

```
XStatus XFlashIntel_Erase( XFlash * InstancePtr,  
                          Xuint32 Offset,  
                          Xuint32 Bytes  
                          )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_EraseBlock( XFlash * InstancePtr,  
                                Xuint16 Region,  
                                Xuint16 Block,  
                                Xuint16 NumBlocks  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_EraseBlockResume( XFlash * InstancePtr,  
                                       Xuint16 Region,  
                                       Xuint16 Block  
                                       )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region & Block parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_EraseBlockSuspend( XFlash * InstancePtr,  
                                        Xuint16 Region,  
                                        Xuint16 Block  
                                        )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region & Block parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_EraseChip( XFlash * InstancePtr)
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_EraseResume( XFlash * InstancePtr,  
                                Xuint32 Offset  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_EraseSuspend( XFlash * InstancePtr,  
                                  Xuint32 Offset  
                                  )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_GetBlockStatus( XFlash * InstancePtr,  
                                    Xuint16 Region,  
                                    Xuint16 Block  
                                    )
```

See the base component for a description of this function, its return values, and arguments.

Note:

The Region & Block parameters are not used because the device's status register appears at every addressible location.

XFlashGeometry* XFlashIntel_GetGeometry(**XFlash *** *InstancePtr*)

See the base component for a description of this function, its return values, and arguments.

Note:

None.

Xuint32 XFlashIntel_GetOptions(**XFlash *** *InstancePtr*)

See the base component for a description of this function, its return values, and arguments.

Note:

None.

XFlashProperties* XFlashIntel_GetProperties(**XFlash *** *InstancePtr*)

See the base component for a description of this function, its return values, and arguments.

Note:

None.

XStatus XFlashIntel_GetStatus(**XFlash *** *InstancePtr*, **Xuint32** *Offset*))

See the base component for a description of this function, its return values, and arguments.

Note:

The Offset parameter is not used because the device's status register appears at every addressible location.

XStatus XFlashIntel_Initialize(**XFlash *** *InstancePtr*)

See the base component for a description of this function, its return values, and arguments.

Parameters:

InstancePtr is a pointer to the flash instance to be worked on.

Returns:

- XST_SUCCESS if successful
- XST_FLASH_PART_NOT_SUPPORTED if the part is not supported

Note:

Two geometry attributes MUST be defined prior to invoking this function:

- BaseAddress
- MemoryLayout

```
XStatus XFlashIntel_Lock( XFlash * InstancePtr,  
                          Xuint32  Offset,  
                          Xuint32  Bytes  
                          )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_LockBlock( XFlash * InstancePtr,  
                               Xuint16  Region,  
                               Xuint16  Block,  
                               Xuint16  NumBlocks  
                               )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_Read( XFlash * InstancePtr,  
                          Xuint32 Offset,  
                          Xuint32 Bytes,  
                          void * DestPtr  
                          )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_ReadBlock( XFlash * InstancePtr,  
                               Xuint16 Region,  
                               Xuint16 Block,  
                               Xuint32 Offset,  
                               Xuint32 Bytes,  
                               void * DestPtr  
                               )
```

See the base component for a description of this function, its return values, and arguments.

Note:

The part is assumed to be in read-array mode.

```
XStatus XFlashIntel_Reset( XFlash * InstancePtr)
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_SelfTest( XFlash * InstancePtr)
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_SetOptions( XFlash * InstancePtr,  
                                Xuint32 OptionsFlag  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_Unlock( XFlash * InstancePtr,  
                            Xuint32 Offset,  
                            Xuint32 Bytes  
                            )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_UnlockBlock( XFlash * InstancePtr,  
                                 Xuint16 Region,  
                                 Xuint16 Block,  
                                 Xuint16 NumBlocks  
                                 )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_Write( XFlash * InstancePtr,  
                            Xuint32 Offset,  
                            Xuint32 Bytes,  
                            void * SrcPtr  
                            )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_WriteBlock( XFlash * InstancePtr,  
                                Xuint16 Region,  
                                Xuint16 Block,  
                                Xuint32 Offset,  
                                Xuint32 Bytes,  
                                void * SrcPtr  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

None.

```
XStatus XFlashIntel_WriteBlockResume( XFlash * InstancePtr,  
                                       Xuint16 Region,  
                                       Xuint16 Block,  
                                       Xuint32 Offset  
                                       )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region, Block, & Offset parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_WriteBlockSuspend( XFlash * InstancePtr,  
                                       Xuint16 Region,  
                                       Xuint16 Block,  
                                       Xuint32 Offset  
                                       )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Region, Block, & Offset parameters are ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_WriteResume( XFlash * InstancePtr,  
                                Xuint32 Offset  
                                )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

```
XStatus XFlashIntel_WriteSuspend( XFlash * InstancePtr,  
                                  Xuint32 Offset  
                                  )
```

See the base component for a description of this function, its return values, and arguments.

Note:

Offset parameter is ignored.

Intel flash does not differentiate RESUME between erase and write. It depends on what the flash was doing at the time the SUSPEND command was issued.

flash/v1_00_a/src/xflash_intel_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xflash_intel_l.h,v 1.1 2002/05/13 19:55:51 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file flash/v1_00_a/src/xflash_intel_l.h
00026 *
00027 * Contains identifiers and low-level macros/functions for the Intel 28FxxxJ3A
00028 * StrataFlash driver.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date        Changes
00034 * -----
00035 * 1.00a rpm  05/06/02  First release
00036 * </pre>
00037 *
00038 *****/
00039
00040 #ifndef XFLASH_INTEL_L_H /* prevent circular inclusions */
00041 #define XFLASH_INTEL_L_H /* by using protection macros */
00042
```

```

00043 /***** Include Files *****/
00044
00045 #include "xbasic_types.h"
00046 #include "xio.h"
00047
00048 /***** Constant Definitions *****/
00049
00050 /*
00051  * Commands written to the devices are defined by the CMD_* constants below.
00052  * Each Command contains 8-bits of significant data. For x16 or greater
devices,
00053  * the command data should appear on the LSB. Other bytes may be written as
00054  * don't cares. To eliminate the need to know the bus layout, the width of
00055  * individual parts, or potential byte-swapping requirements, the CMD_*
00056  * constants are defined with the command data in every nibble.
00057  */
00058
00059 /*
00060  * BCS/SCS command codes
00061  */
00062 #define XFL_INTEL_CMD_READ_ARRAY          0xFFFFFFFFUL
00063 #define XFL_INTEL_CMD_READ_ID_CODES      0x90909090UL
00064 #define XFL_INTEL_CMD_READ_STATUS_REG    0x70707070UL
00065 #define XFL_INTEL_CMD_CLEAR_STATUS_REG   0x50505050UL
00066 #define XFL_INTEL_CMD_WRITE_BUFFER       0xE8E8E8E8UL
00067 #define XFL_INTEL_CMD_PROGRAM            0x40404040UL
00068 #define XFL_INTEL_CMD_BLOCK_ERASE        0x20202020UL
00069 #define XFL_INTEL_CMD_CONFIRM            0xD0D0D0D0UL
00070 #define XFL_INTEL_CMD_SUSPEND            0xB0B0B0B0UL
00071 #define XFL_INTEL_CMD_RESUME             0xD0D0D0D0UL
00072
00073 #define XFL_INTEL_STATUS_READY            0x00800080UL
00074
00075 /*
00076  * SCS command codes
00077  */
00078 #define XFL_INTEL_CMD_READ_QUERY          0x98989898UL
00079 #define XFL_INTEL_CMD_CONFIG              0xB8B8B8B8UL
00080 #define XFL_INTEL_CMD_LOCK_BLOCK_SET      0x60606060UL
00081 #define XFL_INTEL_CMD_LOCK_BLOCK_SET_CONFIRM 0x01010101UL
00082 #define XFL_INTEL_CMD_LOCK_BLOCK_CLEAR    0x60606060UL
00083
00084 /*
00085  * Other command codes
00086  */
00087 #define XFL_INTEL_CMD_PROTECTION          0xC0C0C0C0UL
00088
00089 /*
00090  * Configuration command codes
00091  */
00092 #define XFL_INTEL_CONFIG_RYBY_LEVEL       0x00000000UL
00093 #define XFL_INTEL_CONFIG_RYBY_PULSE_ERASE 0x01010101UL

```

```

00094 #define XFL_INTEL_CONFIG_RYBY_PULSE_WRITE 0x02020202UL
00095 #define XFL_INTEL_CONFIG_RYBY_PULSE_ALL 0x03030303UL
00096
00097
00098 /***** Type Definitions *****/
00099
00100 /***** Macros (Inline Functions) Definitions *****/
00101
00102 /*****
00103 *
00104 * Low-level driver macros and functions. The list below provides signatures
00105 * to help the user use the macros.
00106 *
00107 * void XFlashIntel_mSendCmd(Xuint32 BaseAddress, Xuint32 Offset,
00108 * unsigned int Command)
00109 * int XFlashIntel_WaitReady(Xuint32 BaseAddress, Xuint32 Offset);
00110 *
00111 * int XFlashIntel_WriteAddr(Xuint32 BaseAddress, Xuint32 Offset,
00112 * Xuint8 *BufferPtr, unsigned int Length);
00113 * int XFlashIntel_ReadAddr(Xuint32 BaseAddress, Xuint32 Offset,
00114 * Xuint8 *BufferPtr, unsigned int Length);
00115 * int XFlashIntel_EraseAddr(Xuint32 BaseAddress, Xuint32 Offset);
00116 * int XFlashIntel_LockAddr(Xuint32 BaseAddress, Xuint32 Offset);
00117 * int XFlashIntel_UnlockAddr(Xuint32 BaseAddress, Xuint32 Offset);
00118 *
00119 *****/
00120
00121 /*****
00122 **
00123 *
00124 * Send the specified command to the flash device.
00125 *
00126 * @param BaseAddress is the base address of the device
00127 * @param Offset is the offset address from the base address.
00128 * @param Command is the command to send.
00129 *
00130 * @return None.
00131 *
00132 * @note None.
00133 *
00134 *****/
00135 #define XFlashIntel_mSendCmd(BaseAddress, Offset, Command) \
00136 XIo_Out32((BaseAddress) + (Offset), (Command))
00137
00138 /***** Variable Definitions *****/
00139
00140
00141 /***** Function Prototypes *****/
00142
00143 int XFlashIntel_WaitReady(Xuint32 BaseAddress, Xuint32 Offset);
00144 int XFlashIntel_WriteAddr(Xuint32 BaseAddress, Xuint32 Offset,
00145 Xuint8 *BufferPtr, unsigned int Length);

```



```
00146 int XFlashIntel_ReadAddr(Xuint32 BaseAddress, Xuint32 Offset,  
00147                             Xuint8 *BufferPtr, unsigned int Length);  
00148 int XFlashIntel_EraseAddr(Xuint32 BaseAddress, Xuint32 Offset);  
00149 int XFlashIntel_LockAddr(Xuint32 BaseAddress, Xuint32 Offset);  
00150 int XFlashIntel_UnlockAddr(Xuint32 BaseAddress, Xuint32 Offset);  
00151  
00152  
00153 #endif                /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

flash/v1_00_a/src/xflash_intel_l.h File Reference

Detailed Description

Contains identifiers and low-level macros/functions for the Intel 28FxxxJ3A StrataFlash driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	05/06/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XFlashIntel_mSendCmd(BaseAddress, Offset, Command)
```

Functions

```
int XFlashIntel_WaitReady (Xuint32 BaseAddress, Xuint32 Offset)
```

int **XFlashIntel_WriteAddr** (**Xuint32** BaseAddress, **Xuint32** Offset, **Xuint8** *BufferPtr, unsigned int Length)
int **XFlashIntel_ReadAddr** (**Xuint32** BaseAddress, **Xuint32** Offset, **Xuint8** *BufferPtr, unsigned int Length)
int **XFlashIntel_EraseAddr** (**Xuint32** BaseAddress, **Xuint32** Offset)
int **XFlashIntel_LockAddr** (**Xuint32** BaseAddress, **Xuint32** Offset)
int **XFlashIntel_UnlockAddr** (**Xuint32** BaseAddress, **Xuint32** Offset)

Define Documentation

```
#define XFlashIntel_mSendCmd( BaseAddress,  
                             Offset,  
                             Command  )
```

Send the specified command to the flash device.

Parameters:

BaseAddress is the base address of the device
Offset is the offset address from the base address.
Command is the command to send.

Returns:

None.

Note:

None.

Function Documentation

```
int XFlashIntel_EraseAddr( Xuint32 BaseAddress,  
                           Xuint32 Offset  
                           )
```

Erase the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin erasing. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int XFlashIntel_LockAddr( Xuint32 BaseAddress,  
                        Xuint32 Offset  
                        )
```

Lock the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to lock. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int XFlashIntel_ReadAddr( Xuint32 BaseAddress,  
                        Xuint32 Offset,  
                        Xuint8 * BufferPtr,  
                        unsigned int Length  
                        )
```

Read some number of bytes from the specified address.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin reading.

BufferPtr is the buffer used to store the bytes that are read.

Length is the number of bytes to read from flash.

Returns:

The number of bytes actually read.

Note:

This function assumes 32-bit access to the flash array.

```
int XFlashIntel_UnlockAddr( Xuint32 BaseAddress,  
                           Xuint32 Offset  
                           )
```

Unlock the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to unlock. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int XFlashIntel_WaitReady( Xuint32 BaseAddress,  
                          Xuint32 Offset  
                          )
```

Wait for the flash array to be in the ready state (ready for a command).

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int XFlashIntel_WriteAddr( Xuint32      BaseAddress,  
                           Xuint32      Offset,  
                           Xuint8 *    BufferPtr,  
                           unsigned int Length  
                           )
```

Write the specified address with some number of bytes.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin writing.

BufferPtr is the buffer that will be written to flash.

Length is the number of bytes in *BufferPtr* that will be written to flash.

Returns:

The number of bytes actually written.

Note:

This function assumes 32-bit access to the flash array.

flash/v1_00_a/src/xflash_intel_l.c File Reference

Detailed Description

Contains low-level functions for the XFlashIntel driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	05/06/02	First release

```
#include "xflash_intel_l.h"
```

Functions

```
int XFlashIntel_WriteAddr (Xuint32 BaseAddress, Xuint32 Offset, Xuint8 *BufferPtr, unsigned int Length)
int XFlashIntel_ReadAddr (Xuint32 BaseAddress, Xuint32 Offset, Xuint8 *BufferPtr, unsigned int Length)
int XFlashIntel_EraseAddr (Xuint32 BaseAddress, Xuint32 Offset)
int XFlashIntel_LockAddr (Xuint32 BaseAddress, Xuint32 Offset)
int XFlashIntel_UnlockAddr (Xuint32 BaseAddress, Xuint32 Offset)
int XFlashIntel_WaitReady (Xuint32 BaseAddress, Xuint32 Offset)
```

Function Documentation

```
int XFlashIntel_EraseAddr( Xuint32 BaseAddress,  
                           Xuint32 Offset  
                           )
```

Erase the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin erasing. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int XFlashIntel_LockAddr( Xuint32 BaseAddress,  
                          Xuint32 Offset  
                          )
```

Lock the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to lock. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.


```
int XFlashIntel_ReadAddr( Xuint32   BaseAddress,  
                          Xuint32   Offset,  
                          Xuint8 * BufferPtr,  
                          unsigned int Length  
                          )
```

Read some number of bytes from the specified address.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin reading.

BufferPtr is the buffer used to store the bytes that are read.

Length is the number of bytes to read from flash.

Returns:

The number of bytes actually read.

Note:

This function assumes 32-bit access to the flash array.

```
int XFlashIntel_UnlockAddr( Xuint32 BaseAddress,  
                            Xuint32 Offset  
                            )
```

Unlock the block beginning at the specified address. The user is assumed to know the block boundaries and pass in an address/offset that is block aligned.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to unlock. This offset is assumed to be a block boundary.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int XFlashIntel_WaitReady( Xuint32 BaseAddress,  
                           Xuint32 Offset  
                           )
```

Wait for the flash array to be in the ready state (ready for a command).

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address.

Returns:

0 if successful, or -1 if an error occurred.

Note:

This function assumes 32-bit access to the flash array.

```
int XFlashIntel_WriteAddr( Xuint32 BaseAddress,  
                           Xuint32 Offset,  
                           Xuint8 * BufferPtr,  
                           unsigned int Length  
                           )
```

Write the specified address with some number of bytes.

Parameters:

BaseAddress is the base address of the device.

Offset is the offset address from the base address to begin writing.

BufferPtr is the buffer that will be written to flash.

Length is the number of bytes in *BufferPtr* that will be written to flash.

Returns:

The number of bytes actually written.

Note:

This function assumes 32-bit access to the flash array.

gemac/v1_00_c/src/xgemac.h File Reference

Detailed Description

The Xilinx Ethernet driver component. This component supports the Xilinx Ethernet 1Gbit (GEMAC).

The Xilinx Ethernet 1Gbit MAC supports the following features:

- Scatter-gather & simple DMA operations, as well as simple memory mapped direct I/O interface (FIFOs)
- Gigabit Media Independent Interface (GMII) for connection to external 1Gbit Mbps PHY transceivers. Supports 125Mhz 10 bit interface (TBI) to external PHY and SerDes to external transceiver
- GMII management control reads and writes with GMII PHYs
- Independent internal transmit and receive FIFOs
- CSMA/CD compliant operations for half-duplex modes
- Internal loopback
- Automatic source address insertion or overwrite (programmable)
- Automatic FCS insertion and stripping (programmable)
- Automatic pad insertion and stripping (programmable)
- Pause frame (flow control) detection in full-duplex mode
- Programmable interframe gap
- VLAN frame support
- Jumbo frame support
- Pause frame support

Hardware limitations in this version

- Always in promiscuous mode
- Statistic counters not implemented
- Unicast, multicast, broadcast, and promiscuous address filtering not implemented
- Half-duplex mode not implemented
- Auto source address insertion not implemented

The device driver does not support the features listed below

- Programmable PHY reset signal

Device driver limitations in this version

- Simple DMA untested
- Polled fifo, interrupt driven fifo, and scatter-gather DMA work but have not been stress tested before release. Code that processes error conditions has not been well tested.

Driver Description

The device driver enables higher layer software (e.g., an application) to communicate to the GEMAC. The driver handles transmission and reception of Ethernet frames, as well as configuration of the controller. It does not handle protocol stack functionality such as Link Layer Control (LLC) or the Address Resolution Protocol (ARP). The protocol stack that makes use of the driver handles this functionality. This implies that the driver is simply a pass-through mechanism between a protocol stack and the EMAC. A single device driver can support multiple EMACs.

The driver is designed for a zero-copy buffer scheme. That is, the driver will not copy buffers. This avoids potential throughput bottlenecks within the driver.

Since the driver is a simple pass-through mechanism between a protocol stack and the GEMAC, no assembly or disassembly of Ethernet frames is done at the driver-level. This assumes that the protocol stack passes a correctly formatted Ethernet frame to the driver for transmission, and that the driver does not validate the contents of an incoming frame

PHY Communication

The driver currently does not provide an interface to read or write the PHY.

In the future, the driver provides rudimentary read and write functions to allow the higher layer software to access the PHY. The GEMAC provides MII registers for the driver to access. This management interface can be parameterized away in the FPGA implementation process. If this is the case, the PHY read and write functions of the driver return `XST_NO_FEATURE`.

External loopback is usually supported at the PHY. It is up to the user to turn external loopback on or off at the PHY. The driver simply provides pass-through functions for configuring the PHY. After the initial reset of the PHY during driver initialization, the driver does not read, write, or reset the PHY on its own. All control of the PHY must be done by the user.

Asynchronous Callbacks

The driver services interrupts and passes Ethernet frames to the higher layer software through asynchronous callback functions. When using the driver directly (i.e., not with the RTOS protocol stack), the higher layer software must register its callback functions during initialization. The driver requires callback functions for received frames, for confirmation of transmitted frames, and for asynchronous errors.

Interrupts

The driver has no dependencies on the interrupt controller. The driver provides two interrupt handlers. `XGemac_IntrHandlerDma()` handles interrupts when the GEMAC is configured with scatter-gather DMA. `XGemac_IntrHandlerFifo()` handles interrupts when the GEMAC is configured for direct FIFO I/O or simple DMA. Either of these routines can be connected to the system interrupt controller by the user.

Interrupt Frequency

When the GEMAC is configured with scatter-gather DMA, the frequency of interrupts can be controlled with the interrupt coalescing features of the scatter-gather DMA engine. The frequency of interrupts can be adjusted using the driver API functions for setting the packet count threshold and the packet wait bound values.

The scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

The packet wait bound is a timer value used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold.

Device Reset

Some errors that can occur in the device require a device reset. These errors are listed in the `XGemac_SetErrorHandler()` function header. The user's error handler is responsible for resetting the device and re-configuring it based on its needs (the driver does not save the current configuration). When integrating into an RTOS, these reset and re-configure obligations are taken care of by the Xilinx adapter software.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the `xgemac_g.c` files. A table is defined where each entry contains configuration information for an EMAC device. This information includes such things as the base address of the memory-mapped device, the base addresses of IPIF, DMA, and FIFO modules within the device, and whether the device has DMA, counter registers, multicast support, MII support, and flow control.

The driver tries to use the features built into the device. So if, for example, the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has set up the buffer descriptor lists already. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used. Modifying the configuration table is a workaround for now until we get some experience with how users are intending to use the hardware in its different configurations. For example, if the hardware is built with scatter-gather DMA but the user is intending to use only simple DMA, the user either needs to modify the config table as a workaround or rebuild the hardware with only simple DMA. The recommendation at this point is to build the hardware with the features you intend to use. If you're inclined to modify the table, do so before the call to `XGemac_Initialize()`. Here is a snippet of code that changes a device to simple DMA (the hardware needs to have DMA for this to work of course):

```
XGemac_Config *ConfigPtr;  
  
ConfigPtr = XGemac_LookupConfig(DeviceId);  
ConfigPtr->IpIfDmaConfig = XGE_CFG_SIMPLE_DMA;
```

Simple DMA

Simple DMA is supported through the FIFO functions, `FifoSend` and `FifoRecv`, of the driver (i.e., there is no separate interface for it). The driver makes use of the DMA engine for a simple DMA transfer if the device is configured with DMA, otherwise it uses the FIFOs directly. While the simple DMA interface is therefore transparent to the user, the caching of network buffers is not. If the device is configured with DMA and the FIFO interface is used, the user must ensure that the network buffers are not cached or are cache coherent, since DMA will be used to transfer to and from the Emac device. If the device is configured with DMA and the user really wants to use the FIFOs directly, the user should rebuild the hardware without DMA. If unable to do this, there is a workaround (described above in Device Configuration) to modify the configuration table of the driver to fake the driver into thinking the device has no DMA. A code snippet follows:

```
XGemac_Config *ConfigPtr;  
  
ConfigPtr = XGemac_LookupConfig(DeviceId);  
ConfigPtr->IpIfDmaConfig = XGE_CFG_NO_DMA;
```

Asserts

Asserts are used within all Xilinx drivers to enforce constraints on argument values. Asserts can be turned off on a system-wide basis by defining, at compile time, the NDEBUG identifier. By default, asserts are turned on and it is recommended that application developers leave asserts on during development. Substantial performance improvements can be seen when asserts are disabled.

Building the driver

The **XGemac** driver is composed of several source files. Why so many? This allows the user to build and link only those parts of the driver that are necessary. Since the GEMAC hardware can be configured in various ways (e.g., with or without DMA), the driver too can be built with varying features. For the most part, this means that besides always linking in **xgemac.c**, you link in only the driver functionality you want. Some of the choices you have are polled vs. interrupt, interrupt with FIFOs only vs. interrupt with DMA, self-test diagnostics, and driver statistics. Note that currently the DMA code must be linked in, even if you don't have DMA in the device.

Note:

Xilinx drivers are typically composed of two components, one is the driver and the other is the adapter. The driver is independent of OS and processor and is intended to be highly portable. The adapter is OS-specific and facilitates communication between the driver and the OS.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Dma added, interframe gap interface change, added auto-negotiate option, removed phy function prototypes, added constant to default to no hw counters

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xparameters.h"
#include "xpacket_fifo_v2_00_a.h"
#include "xdma_channel.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XGemac
struct XGemac_Config
```

```
struct XGemac_SoftStats
struct XGemac_Stats
```

Configuration options

Device configuration options (see the **XGemac_SetOptions**() and **XGemac_GetOptions**() for information on how to use these options)

```
#define XGE_UNICAST_OPTION
#define XGE_BROADCAST_OPTION
#define XGE_PROMISC_OPTION
#define XGE_FDUPLEX_OPTION
#define XGE_POLLED_OPTION
#define XGE_LOOPBACK_OPTION
#define XGE_FLOW_CONTROL_OPTION
#define XGE_INSERT_PAD_OPTION
#define XGE_INSERT_FCS_OPTION
#define XGE_STRIP_PAD_FCS_OPTION
#define XGE_AUTO_NEGOTIATE_OPTION
```

Typedefs for callbacks

Callback functions.

```
typedef void(* XGemac_SgHandler )(void *CallbackRef, XBufDescriptor *BdPtr, Xuint32 NumBds)
typedef void(* XGemac_FifoHandler )(void *CallbackRef)
typedef void(* XGemac_ErrorHandler )(void *CallbackRef, XStatus ErrorCode)
```

Defines

```
#define XGemac_mIsSgDma(InstancePtr)
#define XGemac_mIsSimpleDma(InstancePtr)
#define XGemac_mIsDma(InstancePtr)
```

Functions

```
XStatus XGemac_Initialize (XGemac *InstancePtr, Xuint16 DeviceId)
XStatus XGemac_Start (XGemac *InstancePtr)
XStatus XGemac_Stop (XGemac *InstancePtr)
void XGemac_Reset (XGemac *InstancePtr)
XGemac_Config * XGemac_LookupConfig (Xuint16 DeviceId)
XStatus XGemac_SelfTest (XGemac *InstancePtr)
```

XStatus XGmac_PollSend (**XGmac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** ByteCount)
XStatus XGmac_PollRecv (**XGmac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** *ByteCountPtr)
XStatus XGmac_SgSend (**XGmac** *InstancePtr, XBufDescriptor *BdPtr, int Delay)
XStatus XGmac_SgRecv (**XGmac** *InstancePtr, XBufDescriptor *BdPtr)
XStatus XGmac_SetPktThreshold (**XGmac** *InstancePtr, **Xuint32** Direction, **Xuint8** Threshold)
XStatus XGmac_GetPktThreshold (**XGmac** *InstancePtr, **Xuint32** Direction, **Xuint8** *ThreshPtr)
XStatus XGmac_SetPktWaitBound (**XGmac** *InstancePtr, **Xuint32** Direction, **Xuint32** TimerValue)
XStatus XGmac_GetPktWaitBound (**XGmac** *InstancePtr, **Xuint32** Direction, **Xuint32** *WaitPtr)
XStatus XGmac_SetSgRecvSpace (**XGmac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)
XStatus XGmac_SetSgSendSpace (**XGmac** *InstancePtr, **Xuint32** *MemoryPtr, **Xuint32** ByteCount)
void **XGmac_SetSgRecvHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_SgHandler** FuncPtr)
void **XGmac_SetSgSendHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_SgHandler** FuncPtr)
void **XGmac_IntrHandlerDma** (void *InstancePtr)
XStatus XGmac_FifoSend (**XGmac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** ByteCount)
XStatus XGmac_FifoRecv (**XGmac** *InstancePtr, **Xuint8** *BufPtr, **Xuint32** *ByteCountPtr)
void **XGmac_SetFifoRecvHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_FifoHandler** FuncPtr)
void **XGmac_SetFifoSendHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_FifoHandler** FuncPtr)
void **XGmac_IntrHandlerFifo** (void *InstancePtr)
void **XGmac_SetErrorHandler** (**XGmac** *InstancePtr, void *CallBackRef, **XGmac_ErrorHandler** FuncPtr)
XStatus XGmac_SetOptions (**XGmac** *InstancePtr, **Xuint32** OptionFlag)
Xuint32 XGmac_GetOptions (**XGmac** *InstancePtr)
XStatus XGmac_SetMacAddress (**XGmac** *InstancePtr, **Xuint8** *AddressPtr)
void **XGmac_GetMacAddress** (**XGmac** *InstancePtr, **Xuint8** *BufferPtr)
XStatus XGmac_SetInterframeGap (**XGmac** *InstancePtr, **Xuint8** Ifg)
void **XGmac_GetInterframeGap** (**XGmac** *InstancePtr, **Xuint8** *IfgPtr)
void **XGmac_GetStats** (**XGmac** *InstancePtr, **XGmac_Stats** *StatsPtr)
void **XGmac_ClearStats** (**XGmac** *InstancePtr)

Define Documentation

```
#define XGE_AUTO_NEGOTIATE_OPTION
```


XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_BROADCAST_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_FDUPLEX_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_FLOW_CONTROL_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_INSERT_FCS_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_INSERT_PAD_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_LOOPBACK_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_POLLED_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_PROMISC_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_STRIP_PAD_FCS_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGE_UNICAST_OPTION

XGE_BROADCAST_OPTION	Broadcast addressing on or off (default is on)
XGE_UNICAST_OPTION	Unicast addressing on or off (default is on)
XGE_PROMISC_OPTION	Promiscuous addressing on or off (default is off)
XGE_FDUPLEX_OPTION	Full duplex on or off (default is off)
XGE_POLLED_OPTION	Polled mode on or off (default is off)
XGE_LOOPBACK_OPTION	Internal loopback on or off (default is off)
XGE_FLOW_CONTROL_OPTION	Interpret pause frames in full duplex mode (default is off)
XGE_INSERT_PAD_OPTION	Pad short frames on transmit (default is on)
XGE_INSERT_FCS_OPTION	Insert FCS (CRC) on transmit (default is on)
XGE_STRIP_PAD_FCS_OPTION	Strip padding and FCS from received frames (default is off)

#define XGemac_mIsDma(InstancePtr)

This macro determines if the device is currently configured with DMA (either simple DMA or scatter-gather DMA)

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

Boolean XTRUE if the device is configured with DMA, or XFALSE otherwise

Note:

Signature: Xboolean **XGemac_mIsDma**(XGemac *InstancePtr)

#define XGemac_mIsSgDma(InstancePtr)

This macro determines if the device is currently configured for scatter-gather DMA.

Parameters:

InstancePtr is a pointer to the [XGemac](#) instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for scatter-gather DMA, or XFALSE if it is not.

Note:

Signature: Xboolean [XGemac_mIsSgDma](#)(XGemac *InstancePtr)

```
#define XGemac_mIsSimpleDma( InstancePtr )
```

This macro determines if the device is currently configured for simple DMA.

Parameters:

InstancePtr is a pointer to the [XGemac](#) instance to be worked on.

Returns:

Boolean XTRUE if the device is configured for simple DMA, or XFALSE otherwise

Note:

Signature: Xboolean [XGemac_mIsSimpleDma](#)(XGemac *InstancePtr)

Typedef Documentation

```
typedef void(* XGemac_ErrorHandler)(void *CallbackRef, XStatus ErrorCode)
```

Callback when an asynchronous error occurs.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

ErrorCode is a Xilinx error code defined in [xstatus.h](#). Also see [XGemac_SetErrorHandler\(\)](#) for a description of possible errors.

```
typedef void(* XGemac_FifoHandler)(void *CallbackRef)
```

Callback when data is sent or received with direct FIFO communication. The user typically defines two callacks, one for send and one for receive.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

```
typedef void(* XGemac_SgHandler)(void *CallbackRef, XBufDescriptor *BdPtr, Xuint32 NumBds)
```

Callback when data is sent or received with scatter-gather DMA.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

BdPtr is a pointer to the first buffer descriptor in a list of buffer descriptors.

NumBds is the number of buffer descriptors in the list pointed to by *BdPtr*.

Function Documentation

```
void XGemac_ClearStats( XGemac * InstancePtr)
```

Clear the XGemacStats structure for this driver.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus XGemac_FifoRecv( XGemac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 * ByteCountPtr  
                        )
```

Receive an Ethernet frame into the buffer passed as an argument. This function is called in response to the callback function for received frames being called by the driver. The callback function is set up using `SetFifoRecvHandler`, and is invoked when the driver receives an interrupt indicating a received frame. The driver expects the upper layer software to call this function, `FifoRecv`, to receive the frame. The buffer supplied should be large enough to hold a maximum-size Ethernet frame.

The buffer into which the frame will be received must be word-aligned.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from the Emac to memory. This means that this buffer should not be cached. See the comment section "Simple DMA" in `xgemac.h` for more information.

Parameters:

- InstancePtr* is a pointer to the **XGemac** instance to be worked on.
- BufPtr* is a pointer to a word-aligned buffer into which the received Ethernet frame will be copied.
- ByteCountPtr* is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- `XST_SUCCESS` if the frame was sent successfully
- `XST_DEVICE_IS_STOPPED` if the device has not yet been started
- `XST_NOT_INTERRUPT` if the device is not in interrupt mode
- `XST_NO_DATA` if there is no frame to be received from the FIFO
- `XST_BUFFER_TOO_SMALL` if the buffer to receive the frame is too small for the frame waiting in the FIFO.
- `XST_DEVICE_BUSY` if configured for simple DMA and the DMA engine is busy
- `XST_DMA_ERROR` if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

The input buffer must be big enough to hold the largest Ethernet frame.

```
XStatus XGemac_FifoSend( XGemac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 ByteCount  
                        )
```

Send an Ethernet frame using packet FIFO with interrupts. The caller provides a contiguous-memory buffer and its length. The buffer must be word-aligned. The callback function set by using `SetFifoSendHandler` is invoked when the transmission is complete.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from memory to the Emac. This means that this buffer should not be cached. See the comment section "Simple DMA" in `xgemac.h` for more information.

Parameters:

- InstancePtr* is a pointer to the **XGemac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was successfully sent. An interrupt is generated when the GEMAC transmits the frame and the driver calls the callback set with **XGmac_SetFifoSendHandler()**
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_FIFO_NO_ROOM if there is no room in the FIFO for this frame
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```
void XGmac_GetInterframeGap( XGmac * InstancePtr,  
                             Xuint8 * IfgPtr  
                             )
```

Get the interframe gap. See the description of interframe gap above in **XGmac_SetInterframeGap()**.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

IfgPtr is a pointer to an 8-bit buffer into which the interframe gap value will be copied.

Returns:

None. The values of the interframe gap parts are copied into the output parameters.

```
void XGmac_GetMacAddress( XGmac * InstancePtr,  
                         Xuint8 * BufferPtr  
                         )
```

Get the MAC address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufferPtr is an output parameter, and is a pointer to a buffer into which the current MAC address will be copied. The buffer must be at least 6 bytes.

Returns:

None.

Note:

None.

Xuint32 XGmac_GetOptions(XGmac * InstancePtr)

Get Ethernet driver/device options. The 32-bit value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

The 32-bit value of the Ethernet options. The value is a bit-mask representing all options that are currently enabled. See [xgemac.h](#) for a description of the available options.

Note:

None.

XStatus XGmac_GetPktThreshold(XGmac * InstancePtr, Xuint32 Direction, Xuint8 * ThreshPtr)

Get the value of the packet count threshold for this driver/device. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

XStatus XGmac_GetPktWaitBound(XGmac * InstancePtr, Xuint32 Direction, Xuint32 * WaitPtr)

Get the packet wait bound timer for this driver/device. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void XGmac_GetStats( XGmac * InstancePtr,  
                   XGmac_Stats * StatsPtr  
                   )
```

Get a copy of the XGmacStats structure, which contains the current statistics for this driver. The statistics are only cleared at initialization or on demand using the **XGmac_ClearStats()** function.

The DmaErrors and FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the upper layer software.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

```
XStatus XGmac_Initialize( XGmac * InstancePtr,  
                         Xuint16 DeviceId  
                         )
```

Initialize a specific **XGemac** instance/driver. The initialization entails:

- Initialize fields of the **XGemac** structure
- Clear the Ethernet statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- If the device is configured with DMA, configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists are to be passed to the driver.
- Reset the Ethernet MAC

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XGemac** instance. Passing in a device id associates the generic **XGemac** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

void XGemac_IntrHandlerDma(void * *InstancePtr*)

The interrupt handler for the Ethernet driver when configured with scatter- gather DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, Send Packet FIFO, Recv DMA channel, or Send DMA channel. The packet FIFOs only interrupt during "deadlock" conditions.

Parameters:

InstancePtr is a pointer to the **XGemac** instance that just interrupted.

Returns:

None.

Note:

None.

void XGemac_IntrHandlerFifo(void * *InstancePtr*)

The interrupt handler for the Ethernet driver when configured for direct FIFO communication (as opposed to DMA).

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, or Send Packet FIFO. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the MAC.

Parameters:

InstancePtr is a pointer to the **XGmac** instance that just interrupted.

Returns:

None.

Note:

None.

XGmac_Config* XGmac_LookupConfig(Xuint16 DeviceId)

Lookup the device configuration based on the unique device ID. The table EmacConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

**XStatus XGmac_PollRecv(XGmac * InstancePtr,
Xuint8 * BufPtr,
Xuint32 * ByteCountPtr
)**

Receive an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the frame directly from the MAC's packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The MAC's error status is not checked, so statistics are not updated for polled receive. The buffer into which the frame will be received must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- o XST_SUCCESS if the frame was sent successfully

- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.

Note:

Input buffer must be big enough to hold the largest Ethernet frame. Buffer must also be 32-bit aligned.

```
XStatus XGmac_PollSend( XGmac * InstancePtr,
                        Xuint8 * BufPtr,
                        Xuint32 ByteCount
                        )
```

Send an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the frame directly to the MAC's packet FIFO, then enters a loop checking the device status for completion or error. Statistics are updated if an error occurs. The buffer to be sent must be word-aligned.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_FIFO_NO_ROOM if there is no room in the GEMAC's length FIFO for this frame
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.
- XST_EMAC_COLLISION if the send failed due to excess deferral or late collision

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread. On a 1 Gbit (1000Mbps) MAC, it takes about 12.1 usecs to transmit a maximum size Ethernet frame.

```
void XGmac_Reset( XGmac * InstancePtr)
```

Reset the Ethernet MAC. This is a graceful reset in that the device is stopped first. Resets the DMA channels, the FIFOs, the transmitter, and the receiver. The PHY is not reset. Any frames in the scatter-gather descriptor lists will remain in the lists. The side effect of doing this is that after a reset and following a restart of the device, frames that were in the list before the reset may be transmitted or received. Reset must only be called after the driver has been initialized.

The configuration after this reset is as follows:

- Disabled transmitter and receiver
- Enabled PHY (the PHY is not reset)
- MAC transmitter does pad insertion, FCS insertion, and source address overwrite.
- MAC receiver does not strip padding or FCS
- Interframe Gap as recommended by IEEE Std. 802.3 (96 bit times)
- Unicast addressing enabled
- Broadcast addressing enabled
- Multicast addressing disabled (addresses are preserved)
- Promiscuous addressing disabled
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- MAC address of all zeros

The upper layer software is responsible for re-configuring (if necessary) and restarting the MAC after the reset. Note that the PHY is not reset. PHY control is left to the upper layer software. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

None.

Note:

None.

XStatus XGemac_SelfTest(XGemac * InstancePtr)

Performs a self-test on the Ethernet device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the Ethernet device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local

buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_PFIPO_BAD_REG_VALUE	FIFO failed register self-test
XST_DMA_TRANSFER_ERROR	DMA failed data transfer self-test
XST_DMA_RESET_REGISTER_ERROR	DMA control register value was incorrect after a reset
XST_REGISTER_ERROR	Ethernet failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed
XST_IPIF_REG_WIDTH_ERROR	An invalid register width was passed into the function
XST_IPIF_RESET_REGISTER_ERROR	The value of a register at reset was invalid
XST_IPIF_DEVICE_STATUS_ERROR	A write to the device status register did not read back correctly
XST_IPIF_DEVICE_ACK_ERROR	A bit in the device status register did not reset when acked
XST_IPIF_DEVICE_ENABLE_ERROR	The device interrupt enable register was not updated correctly by the hardware when other registers were written to
XST_IPIF_IP_STATUS_ERROR	A write to the IP interrupt status register did not read back correctly
XST_IPIF_IP_ACK_ERROR	One or more bits in the IP status register did not reset when acked
XST_IPIF_IP_ENABLE_ERROR	The IP interrupt enable register was not updated correctly when other registers were written to

Note:

This function makes use of options-related functions, and the **XGemac_PollSend()** and **XGemac_PollRecv()** functions.

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the self-test thread.

```
void XGemac_SetErrorHandler( XGemac *           InstancePtr,
                             void *           CallbackRef,
                             XGemac_ErrorHandler FuncPtr
                             )
```

Set the callback function for handling asynchronous errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- `XST_DMA_ERROR` indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- `XST_FIFO_ERROR` indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- `XST_RESET_ERROR` indicates an unrecoverable MAC error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- `XST_DMA_SG_NO_LIST` indicates an attempt was made to access a scatter-gather DMA list that has not yet been created.
- `XST_DMA_SG_LIST_EMPTY` indicates the driver tried to get a descriptor from the receive descriptor list, but the list was empty.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XGmac_SetFifoRecvHandler( XGmac *           InstancePtr,  
                             void *           CallbackRef,  
                             XGmac_FifoHandler FuncPtr  
                             )
```

Set the callback function for handling confirmation of transmitted frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called by the driver once per frame sent. The callback is responsible for freeing the transmitted buffer if necessary.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XGemac_SetFifoSendHandler( XGemac * InstancePtr,
                               void * CallbackRef,
                               XGemac_FifoHandler FuncPtr
                               )
```

Set the callback function for handling received frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called once per frame received. During the callback, the upper layer software should call `FifoRecv` to retrieve the received frame.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XGemac_SetInterframeGap( XGemac * InstancePtr,
                                  Xuint8 Ifg
                                  )
```

Set the Interframe Gap (IFG), which is the time the MAC delays between transmitting frames. There are two parts required. The total interframe gap is the total of the two parts. The values provided for the Part1 and Part2 parameters are multiplied by 4 to obtain the bit-time interval. The first part should be the first 2/3 of the total interframe gap. The MAC will reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part1. Part1 may be shorter than 2/3 the total and can be as small as zero. The second part should be the last 1/3 of the total interframe gap, but can be as large as the total interframe gap. The MAC will not reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part2.

The device must be stopped before setting the interframe gap.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Ifg is the interframe gap, MSB is 8-bit time.

Returns:

- XST_SUCCESS if the interframe gap was set successfully
- XST_DEVICE_IS_STARTED if the device has not been stopped

Note:

None.

```
XStatus XGmac_SetMacAddress( XGmac * InstancePtr,  
                             Xuint8 * AddressPtr  
                             )
```

Set the MAC address for this driver/device. The address is a 48-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address.

Returns:

- XST_SUCCESS if the MAC address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus XGmac_SetOptions( XGmac * InstancePtr,  
                          Xuint32 OptionsFlag  
                          )
```

Set Ethernet driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

OptionsFlag is a bit-mask representing the Ethernet options to turn on or off. See [xgemac.h](#) for a description of the available options.

Returns:

- XST_SUCCESS if the options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

```
XStatus XGemac_SetPktThreshold( XGemac * InstancePtr,  
                                Xuint32   Direction,  
                                Xuint8    Threshold  
                                )
```

Set the packet count threshold for this device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_DMA_SG_COUNT_EXCEEDED if the threshold must be equal to or less than the number of descriptors in the list
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
XStatus XGemac_SetPktWaitBound( XGemac * InstancePtr,  
                                Xuint32   Direction,  
                                Xuint32   TimerValue  
                                )
```

Set the packet wait bound timer for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA

- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void XGemac_SetSgRecvHandler( XGemac * InstancePtr,
                             void * CallbackRef,
                             XGemac_SgHandler FuncPtr
                             )
```

Set the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame received. The head of a descriptor list is passed in along with the number of descriptors in the list. Before leaving the callback, the upper layer software should attach a new buffer to each descriptor in the list.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XGemac_SetSgRecvSpace( XGemac * InstancePtr,
                               Xuint32 * MemoryPtr,
                               Xuint32 ByteCount
                               )
```

Give the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xgemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be aligned on an 8 byte boundary. An assert will occur if asserts are turned on and the memory is not aligned correctly.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

MemoryPtr is a pointer to the 64-bit aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the [XGmac_Initialize\(\)](#) function because the DMA channel components must be initialized before the memory space is set.

```
void XGmac_SetSgSendHandler( XGmac *           InstancePtr,
                             void *           CallbackRef,
                             XGmac_SgHandler FuncPtr
                             )
```

Set the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame sent. The head of a descriptor list is passed in along with the number of descriptors in the list. The callback is responsible for freeing buffers attached to these descriptors.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XGmac_SetSgSendSpace( XGmac * InstancePtr,  
                               Xuint32 * MemoryPtr,  
                               Xuint32   ByteCount  
                               )
```

Give the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The `xgemac.h` file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be aligned on an 8 byte boundary. An assert will occur if asserts are turned on and the memory is not aligned correctly.

Parameters:

InstancePtr is a pointer to the `XGmac` instance to be worked on.

MemoryPtr is a pointer to the 64-bit aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the `XGmac_Initialize()` function because the DMA channel components must be initialized before the memory space is set.

```
XStatus XGmac_SgRecv( XGmac * InstancePtr,  
                     XBufDescriptor * BdPtr  
                     )
```

Add a descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be aligned on an 8 byte boundary. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. This function can be called when the device is started or stopped. Note that it does start the scatter-gather DMA engine. Although this is not necessary during initialization, it is not a problem during initialization because the MAC receiver is not yet started.

The buffer attached to the descriptor must be 64-bit aligned on both the front end and the back end.

Notification of received frames are done asynchronously through the receive callback function.

Parameters:

InstancePtr is a pointer to the `XGmac` instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full

- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

```
XStatus XGmac_SgSend( XGmac * InstancePtr,
                    XBufDescriptor * BdPtr,
                    int Delay
                    )
```

Send an Ethernet frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire Ethernet frame may or may not be contained within one descriptor. This function simply inserts the descriptor into the scatter-gather engine's transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be aligned on an 8 byte boundary.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

The buffer attached to the descriptor must be 64-bit aligned on the front end.

This call is non-blocking. Notification of error or successful transmission is done asynchronously through the send or error callback function.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Delay indicates whether to start the scatter-gather DMA channel immediately, or whether to wait. This allows the user to build up a list of more than one descriptor before starting the transmission of the packets, which allows the application to keep up with DMA and have a constant stream of frames being transmitted. Use XEM_SGDMA_NODELAY or XEM_SGDMA_DELAY, defined in **xgemac.h**, as the value of this argument. If the user chooses to delay and build a list, the user must call this function with the XEM_SGDMA_NODELAY option or call **XGmac_Start()** to kick off the transmissions.

Returns:

- XST_SUCCESS if the buffer was successful sent
- XST_DEVICE_IS_STOPPED if the Ethernet MAC has not been started yet
- XST_NOT_SGDMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the descriptor list for the DMA channel is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

XStatus XGmac_Start(XGmac * InstancePtr)

Start the Ethernet controller as follows:

- If in interrupt driven mode
 - Set the internal interrupt enable registers appropriately
 - Enable interrupts within the device itself. Note that connection of the driver's interrupt handler to the interrupt source (typically done using the interrupt controller component) is done by the higher layer software.
 - If the device is configured with DMA, start the DMA channels if the descriptor lists are not empty
- Enable the transmitter
- Enable the receiver

The PHY is enabled after driver initialization. We assume the upper layer software has configured it and the GEMAC appropriately before this function is called.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully
- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.

Note:

The driver tries to match the hardware configuration. So if the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has set up the buffer descriptor lists already. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used. Modifying the config table is a workaround for now until we get some experience with how users are intending to use the hardware in its different configurations. For example, if the hardware is built with scatter-gather DMA but the user is intending to use only simple DMA, the user either needs to modify the config table as a workaround or rebuild the hardware with only simple DMA.

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus XGmac_Stop(XGmac * InstancePtr)

Stop the Ethernet MAC as follows:

- If the device is configured with scatter-gather DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode (the higher layer software is responsible for disabling interrupts at the interrupt controller)

The PHY is left enabled after a Stop is called.

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the Ethernet frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

gemac/v1_00_c/src/xgemac_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of GEMAC devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release

```
#include "xgemac.h"  
#include "xparameters.h"
```

Variables

[XGmac_Config](#) [XGmac_ConfigTable](#) [XPAR_XGEMAC_NUM_INSTANCES]

Variable Documentation

[XGmac_Config](#) [XGmac_ConfigTable](#)[XPAR_XGEMAC_NUM_INSTANCES]

This table contains configuration information for each GEMAC device in the system.

gemac/v1_00_c/src/xgemac.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xgemac.h,v 1.1 2003/05/29 19:43:52 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file gemac/v1_00_c/src/xgemac.h
00026 *
00027 * The Xilinx Ethernet driver component.  This component supports the Xilinx
00028 * Ethernet 1Gbit (GEMAC).
00029 *
00030 * The Xilinx Ethernet 1Gbit MAC supports the following features:
00031 * - Scatter-gather & simple DMA operations, as well as simple memory
00032 *   mapped direct I/O interface (FIFOs)
00033 * - Gigabit Media Independent Interface (GMII) for connection to external
00034 *   1Gbit Mbps PHY transceivers. Supports 125Mhz 10 bit interface (TBI) to
00035 *   external PHY and SerDes to external transceiver
00036 * - GMII management control reads and writes with GMII PHYs
00037 * - Independent internal transmit and receive FIFOs
00038 * - CSMA/CD compliant operations for half-duplex modes
00039 * - Internal loopback
00040 * - Automatic source address insertion or overwrite (programmable)
00041 * - Automatic FCS insertion and stripping (programmable)
00042 * - Automatic pad insertion and stripping (programmable)
```

00043 * - Pause frame (flow control) detection in full-duplex mode
00044 * - Programmable interframe gap
00045 * - VLAN frame support
00046 * - Jumbo frame support
00047 * - Pause frame support
00048 *
00049 * Hardware limitations in this version
00050 * - Always in promiscuous mode
00051 * - Statistic counters not implemented
00052 * - Unicast, multicast, broadcast, and promiscuous address filtering not
00053 * implemented
00054 * - Half-duplex mode not implemented
00055 * - Auto source address insertion not implemented
00056 *
00057 * The device driver does not support the features listed below
00058 * - Programmable PHY reset signal
00059 *
00060 * Device driver limitations in this version
00061 * - Simple DMA untested
00062 * - Polled fifo, interrupt driven fifo, and scatter-gather DMA work but have
00063 * not been stress tested before release. Code that processes error
00064 * conditions has not been well tested.
00065 *
00066 * Driver Description
00067 *
00068 * The device driver enables higher layer software (e.g., an application) to
00069 * communicate to the GEMAC. The driver handles transmission and reception of
00070 * Ethernet frames, as well as configuration of the controller. It does not
00071 * handle protocol stack functionality such as Link Layer Control (LLC) or the
00072 * Address Resolution Protocol (ARP). The protocol stack that makes use of the
00073 * driver handles this functionality. This implies that the driver is simply a
00074 * pass-through mechanism between a protocol stack and the EMAC. A single device
00075 * driver can support multiple EMACs.
00076 *
00077 * The driver is designed for a zero-copy buffer scheme. That is, the driver
will
00078 * not copy buffers. This avoids potential throughput bottlenecks within the
00079 * driver.
00080 *
00081 * Since the driver is a simple pass-through mechanism between a protocol stack
00082 * and the GEMAC, no assembly or disassembly of Ethernet frames is done at the
00083 * driver-level. This assumes that the protocol stack passes a correctly
00084 * formatted Ethernet frame to the driver for transmission, and that the driver
00085 * does not validate the contents of an incoming frame
00086 *
00087 * PHY Communication
00088 *
00089 * The driver currently does not provide an interface to read or write the PHY.
00090 *
00091 * In the future, the driver provides rudimentary read and write functions to
00092 * allow the higher layer software to access the PHY. The GEMAC provides MII
00093 * registers for the driver to access. This management interface can be
00094 * parameterized away in the FPGA implementation process. If this is the case,

00095 * the PHY read and write functions of the driver return XST_NO_FEATURE.
00096 *
00097 * External loopback is usually supported at the PHY. It is up to the user to
00098 * turn external loopback on or off at the PHY. The driver simply provides pass-
00099 * through functions for configuring the PHY. After the initial reset of the PHY
00100 * during driver initialization, the driver does not read, write, or reset the
00101 * PHY on its own. All control of the PHY must be done by the user.
00102 *
00103 * **Asynchronous Callbacks**
00104 *
00105 * The driver services interrupts and passes Ethernet frames to the higher layer
00106 * software through asynchronous callback functions. When using the driver
00107 * directly (i.e., not with the RTOS protocol stack), the higher layer
00108 * software must register its callback functions during initialization. The
00109 * driver requires callback functions for received frames, for confirmation of
00110 * transmitted frames, and for asynchronous errors.
00111 *
00112 * **Interrupts**
00113 *
00114 * The driver has no dependencies on the interrupt controller. The driver
00115 * provides two interrupt handlers. XGemac_IntrHandlerDma() handles interrupts
00116 * when the GEMAC is configured with scatter-gather DMA.
XGemac_IntrHandlerFifo()
00117 * handles interrupts when the GEMAC is configured for direct FIFO I/O or simple
00118 * DMA. Either of these routines can be connected to the system interrupt
00119 * controller by the user.
00120 *
00121 * **Interrupt Frequency**
00122 *
00123 * When the GEMAC is configured with scatter-gather DMA, the frequency of
00124 * interrupts can be controlled with the interrupt coalescing features of the
00125 * scatter-gather DMA engine. The frequency of interrupts can be adjusted using
00126 * the driver API functions for setting the packet count threshold and the
packet
00127 * wait bound values.
00128 *
00129 * The scatter-gather DMA engine only interrupts when the packet count threshold
00130 * is reached, instead of interrupting for each packet. A packet is a generic
00131 * term used by the scatter-gather DMA engine, and is equivalent to an Ethernet
00132 * frame in our case.
00133 *
00134 * The packet wait bound is a timer value used during interrupt coalescing to
00135 * trigger an interrupt when not enough packets have been received to reach the
00136 * packet count threshold.
00137 *
00138 * **Device Reset**
00139 *
00140 * Some errors that can occur in the device require a device reset. These errors
00141 * are listed in the XGemac_SetErrorHandler() function header. The user's error
00142 * handler is responsible for resetting the device and re-configuring it based
on
00143 * its needs (the driver does not save the current configuration). When

00144 * integrating into an RTOS, these reset and re-configure obligations are
00145 * taken care of by the Xilinx adapter software.
00146 *
00147 * **Device Configuration**
00148 *
00149 * The device can be configured in various ways during the FPGA implementation
00150 * process. Configuration parameters are stored in the xgemac_g.c files.
00151 * A table is defined where each entry contains configuration information
00152 * for an EMAC device. This information includes such things as the base
address
00153 * of the memory-mapped device, the base addresses of IPIF, DMA, and FIFO
modules
00154 * within the device, and whether the device has DMA, counter registers,
00155 * multicast support, MII support, and flow control.
00156 *
00157 * The driver tries to use the features built into the device. So if, for
00158 * example, the hardware is configured with scatter-gather DMA, the driver
00159 * expects to start the scatter-gather channels and expects that the user has
set
00160 * up the buffer descriptor lists already. If the user expects to use the driver
00161 * in a mode different than how the hardware is configured, the user should
00162 * modify the configuration table to reflect the mode to be used. Modifying the
00163 * configuration table is a workaround for now until we get some experience with
00164 * how users are intending to use the hardware in its different configurations.
00165 * For example, if the hardware is built with scatter-gather DMA but the user is
00166 * intending to use only simple DMA, the user either needs to modify the config
00167 * table as a workaround or rebuild the hardware with only simple DMA. The
00168 * recommendation at this point is to build the hardware with the features you
00169 * intend to use. If you're inclined to modify the table, do so before the call
00170 * to XGEmac_Initialize(). Here is a snippet of code that changes a device to
00171 * simple DMA (the hardware needs to have DMA for this to work of course):
00172 *

```
  
00173 *     XGEmac_Config *ConfigPtr;  
00174 *  
00175 *     ConfigPtr = XGEmac_LookupConfig(DeviceId);  
00176 *     ConfigPtr->IpIfDmaConfig = XGE_CFG_SIMPLE_DMA;  
00177 * 
```


00178 *
00179 * **Simple DMA**
00180 *
00181 * Simple DMA is supported through the FIFO functions, FifoSend and FifoRecv, of
00182 * the driver (i.e., there is no separate interface for it). The driver makes
use
00183 * of the DMA engine for a simple DMA transfer if the device is configured with
00184 * DMA, otherwise it uses the FIFOs directly. While the simple DMA interface is
00185 * therefore transparent to the user, the caching of network buffers is not.
00186 * If the device is configured with DMA and the FIFO interface is used, the user
00187 * must ensure that the network buffers are not cached or are cache coherent,
00188 * since DMA will be used to transfer to and from the Emac device. If the device
00189 * is configured with DMA and the user really wants to use the FIFOs directly,
00190 * the user should rebuild the hardware without DMA. If unable to do this, there
00191 * is a workaround (described above in Device Configuration) to modify the
00192 * configuration table of the driver to fake the driver into thinking the device

```

00193 * has no DMA. A code snippet follows:
00194 * <pre>
00195 *     XGemac_Config *ConfigPtr;
00196 *
00197 *     ConfigPtr = XGemac_LookupConfig(DeviceId);
00198 *     ConfigPtr->IpIfDmaConfig = XGE_CFG_NO_DMA;
00199 * </pre>
00200 *
00201 * <b>Asserts</b>
00202 *
00203 * Asserts are used within all Xilinx drivers to enforce constraints on argument
00204 * values. Asserts can be turned off on a system-wide basis by defining, at
compile
00205 * time, the NDEBUG identifier. By default, asserts are turned on and it is
00206 * recommended that application developers leave asserts on during development.
00207 * Substantial performance improvements can be seen when asserts are disabled.
00208 *
00209 * <b>Building the driver</b>
00210 *
00211 * The XGemac driver is composed of several source files. Why so many? This
00212 * allows the user to build and link only those parts of the driver that are
00213 * necessary. Since the GEMAC hardware can be configured in various ways (e.g.,
00214 * with or without DMA), the driver too can be built with varying features.
00215 * For the most part, this means that besides always linking in xgemac.c, you
00216 * link in only the driver functionality you want. Some of the choices you have
00217 * are polled vs. interrupt, interrupt with FIFOs only vs. interrupt with DMA,
00218 * self-test diagnostics, and driver statistics. Note that currently the DMA
code
00219 * must be linked in, even if you don't have DMA in the device.
00220 *
00221 * @note
00222 *
00223 * Xilinx drivers are typically composed of two components, one is the driver
00224 * and the other is the adapter. The driver is independent of OS and processor
00225 * and is intended to be highly portable. The adapter is OS-specific and
00226 * facilitates communication between the driver and the OS.
00227 * <br><br>
00228 * This driver is intended to be RTOS and processor independent. It works
00229 * with physical addresses only. Any needs for dynamic memory management,
00230 * threads or thread mutual exclusion, virtual memory, or cache control must
00231 * be satisfied by the layer above this driver.
00232 *
00233 * <pre>
00234 * MODIFICATION HISTORY:
00235 *
00236 * Ver    Who    Date      Changes
00237 * -----
00238 * 1.00a  ecm    01/13/03  First release
00239 * 1.00b  ecm    03/25/03  Revision update
00240 * 1.00c  rmm    05/28/03  Dma added, interframe gap interface change, added auto-
00241 *          negotiate option, removed phy function prototypes,
00242 *          added constant to default to no hw counters

```



```

00243 * </pre>
00244 *
00245 *****/
00246
00247 #ifndef XGEMAC_H /* prevent circular inclusions */
00248 #define XGEMAC_H /* by using protection macros */
00249
00250 /***** Include Files *****/
00251
00252 #include "xbasic_types.h"
00253 #include "xstatus.h"
00254 #include "xparameters.h"
00255 #include "xpacket_fifo_v2_00_a.h" /* Uses v1.00b of Packet Fifo */
00256 #include "xdma_channel.h"
00257
00258 /***** Constant Definitions *****/
00259
00260 /*
00261  * Device information
00262  */
00263 #define XGE_DEVICE_NAME      "xgemac"
00264 #define XGE_DEVICE_DESC     "Xilinx Ethernet 1Gbit MAC"
00265
00266 /*
00267  * Hardware counters not supported in this version
00268  */
00269 #define GEMAC_NO_HW_COUNTERS
00270
00271 /** @name Configuration options
00272  *
00273  * Device configuration options (see the XGemac_SetOptions() and
00274  * XGemac_GetOptions() for information on how to use these options)
00275  * @{
00276  */
00277 /**
00278  * <pre>
00279  *   XGE_BROADCAST_OPTION      Broadcast addressing on or off (default is on)
00280  *   XGE_UNICAST_OPTION        Unicast addressing on or off (default is on)
00281  *   XGE_PROMISC_OPTION        Promiscuous addressing on or off (default is
00282  *                               off)
00283  *   XGE_FDUPLEX_OPTION        Full duplex on or off (default is off)
00284  *   XGE_POLLED_OPTION         Polled mode on or off (default is off)
00285  *   XGE_LOOPBACK_OPTION       Internal loopback on or off (default is off)
00286  *   XGE_FLOW_CONTROL_OPTION   Interpret pause frames in full duplex mode
00287  *                               (default is off)
00288  *   XGE_INSERT_PAD_OPTION     Pad short frames on transmit (default is on)
00289  *   XGE_INSERT_FCS_OPTION     Insert FCS (CRC) on transmit (default is on)
00290  *   XGE_STRIP_PAD_FCS_OPTION  Strip padding and FCS from received frames
00291  *                               (default is off)
00292  * </pre>
00293  */
00293 #define XGE_UNICAST_OPTION      0x00000001UL

```

```

00294 #define XGE_BROADCAST_OPTION      0x00000002UL
00295 #define XGE_PROMISC_OPTION         0x00000004UL
00296 #define XGE_FDUPLEX_OPTION         0x00000008UL
00297 #define XGE_POLLED_OPTION           0x00000010UL
00298 #define XGE_LOOPBACK_OPTION         0x00000020UL
00299 #define XGE_FLOW_CONTROL_OPTION     0x00000080UL
00300 #define XGE_INSERT_PAD_OPTION       0x00000100UL
00301 #define XGE_INSERT_FCS_OPTION       0x00000200UL
00302 #define XGE_STRIP_PAD_FCS_OPTION    0x00002000UL
00303 #define XGE_AUTO_NEGOTIATE_OPTION   0x00004000UL
00304
00305 /*@}*/
00306 /*
00307  * Not supported yet:
00308  *   XGE_MULTICAST_OPTION           Multicast addressing on or off (default is
off)
00309  *   XGE_INSERT_ADDR_OPTION         Insert source address on transmit (default is
on)
00310  *   XGE_OVWRT_ADDR_OPTION          Overwrite source address on transmit. This is
00311  *                                   only used if source address insertion is on.
00312  *                                   (default is on)
00313  */
00314 /* NOT SUPPORTED YET... */
00315 #define XGE_MULTICAST_OPTION         0x00000040UL
00316 #define XGE_INSERT_ADDR_OPTION       0x00000400UL
00317 #define XGE_OVWRT_ADDR_OPTION        0x00000800UL
00318
00319 /*
00320  * Some default values for interrupt coalescing within the scatter-gather
00321  * DMA engine.
00322  */
00323 #define XGE_SGDMA_DFT_THRESHOLD      1          /* Default pkt threshold */
00324 #define XGE_SGDMA_MAX_THRESHOLD      255       /* Maximum pkt threshold */
00325 #define XGE_SGDMA_DFT_WAITBOUND     5          /* Default pkt wait bound (msec) */
00326 #define XGE_SGDMA_MAX_WAITBOUND     1023      /* Maximum pkt wait bound (msec) */
00327
00328 /*
00329  * Direction identifiers. These are used for setting values like packet
00330  * thresholds and wait bound for specific channels
00331  */
00332 #define XGE_SEND      1
00333 #define XGE_RECV      2
00334
00335 /*
00336  * Arguments to SgSend function to indicate whether to hold off starting
00337  * the scatter-gather engine.
00338  */
00339 #define XGE_SGDMA_NODELAY      0          /* start SG DMA immediately */
00340 #define XGE_SGDMA_DELAY       1          /* do not start SG DMA */
00341
00342 /*

```

```

00343 * Constants to determine the configuration of the hardware device. They are
00344 * used to allow the driver to verify it can operate with the hardware.
00345 */
00346 #define XGE_CFG_NO_IPIF          0          /* Not supported by the driver */
00347 #define XGE_CFG_NO_DMA          1          /* No DMA */
00348 #define XGE_CFG_SIMPLE_DMA      2          /* Simple DMA */
00349 #define XGE_CFG_DMA_SG          3          /* DMA scatter gather */
00350
00351 /*
00352 * The next few constants help upper layers determine the size of memory
00353 * pools used for Ethernet buffers and descriptor lists.
00354 */
00355 #define XGE_MAC_ADDR_SIZE       6          /* six-byte MAC address */
00356 #define XGE_MTU                  1500     /* max size of Ethernet frame */
00357 #define XGE_HDR_SIZE            14        /* size of Ethernet header */
00358 #define XGE_HDR_VLAN_SIZE       18        /* size of Ethernet header with VLAN */
00359 #define XGE_TRL_SIZE            4         /* size of Ethernet trailer (FCS) */
00360 #define XGE_MAX_FRAME_SIZE      (XGE_MTU + XGE_HDR_SIZE + XGE_TRL_SIZE)
00361 #define XGE_MAX_VLAN_FRAME_SIZE (XGE_MTU + XGE_HDR_VLAN_SIZE + XGE_TRL_SIZE)
00362
00363 /*
00364 * Define a default number of send and receive buffers
00365 */
00366 #define XGE_MIN_RECV_BUFS       32        /* minimum # of recv buffers */
00367 #define XGE_DFT_RECV_BUFS      64        /* default # of recv buffers */
00368
00369 #define XGE_MIN_SEND_BUFS       16        /* minimum # of send buffers */
00370 #define XGE_DFT_SEND_BUFS      32        /* default # of send buffers */
00371
00372 #define XGE_MIN_BUFFERS         (XGE_MIN_RECV_BUFS + XGE_MIN_SEND_BUFS)
00373 #define XGE_DFT_BUFFERS         (XGE_DFT_RECV_BUFS + XGE_DFT_SEND_BUFS)
00374
00375 /*
00376 * Define the number of send and receive buffer descriptors, used for
00377 * scatter-gather DMA
00378 */
00379 #define XGE_MIN_RECV_DESC       16        /* minimum # of recv descriptors */
00380 #define XGE_DFT_RECV_DESC       32        /* default # of recv descriptors */
00381
00382 #define XGE_MIN_SEND_DESC       8         /* minimum # of send descriptors */
00383 #define XGE_DFT_SEND_DESC       16       /* default # of send descriptors */
00384
00385 /***** Type Definitions *****/
00386
00387 /**
00388 * Ethernet statistics (see XGemac_GetStats() and XGemac_ClearStats())
00389 */
00390 typedef struct
00391 {
00392     Xuint32 XmitFramesMSL;           /**< Number of frames transmitted upper 32
bits */
00393     Xuint32 XmitFrames;              /**< Number of frames transmitted */

```

```

00394     Xuint32 XmitBytesMSL;           /**< Number of bytes transmitted upper 32
bits */
00395     Xuint32 XmitBytes;               /**< Number of bytes transmitted */
00396     Xuint32 XmitLateCollisionErrorsMSL; /**< Number of transmission failures
00397                                     due to late collisions upper 32 bits
*/
00398     Xuint32 XmitLateCollisionErrors; /**< Number of transmission failures
00399                                     due to late collisions */
00400     Xuint32 XmitExcessDeferralMSL;    /**< Number of transmission failures
00401                                     due to excess collision deferrals
upper 32 bits */
00402     Xuint32 XmitExcessDeferral;      /**< Number of transmission failures
00403                                     due to excess collision deferrals */
00404     Xuint32 XmitOverrunErrors;       /**< Number of transmit overrun errors
upper 32 bits */
00405     Xuint32 XmitUnderrunErrorsMSL;   /**< Number of transmit underrun errors */
00406     Xuint32 XmitUnderrunErrors;     /**< Number of transmit underrun errors */
00407     Xuint32 RecvFramesMSL;           /**< Number of frames received upper 32
bits */
00408     Xuint32 RecvFrames;              /**< Number of frames received */
00409     Xuint32 RecvBytesMSL;           /**< Number of bytes received upper 32
bits */
00410     Xuint32 RecvBytes;               /**< Number of bytes received */
00411     Xuint32 RecvFcsErrorsMSL;       /**< Number of frames discarded due
00412                                     to FCS errors upper 32 bits */
00413     Xuint32 RecvFcsErrors;          /**< Number of frames discarded due
00414                                     to FCS errors */
00415     Xuint32 SlotLengthErrors;        /**< Number of frames received with
00416                                     slot length errors */
00417     Xuint32 RecvOverrunErrors;       /**< Number of frames discarded due
00418                                     to overrun errors */
00419     Xuint32 RecvUnderrunErrors;      /**< Number of recv underrun errors */
00420     Xuint32 RecvMissedFrameErrors;   /**< Number of frames missed by MAC */
00421     Xuint32 RecvCollisionErrorsMSL;  /**< Number of frames discarded due
00422                                     to collisions upper 32 bits */
00423     Xuint32 RecvCollisionErrors;     /**< Number of frames discarded due
00424                                     to collisions */
00425     Xuint32 RecvLengthFieldErrors;   /**< Number of frames discarded with
00426                                     invalid length field */
00427     Xuint32 RecvShortErrors;         /**< Number of short frames discarded */
00428     Xuint32 RecvLongErrors;         /**< Number of long frames discarded */
00429     Xuint32 DmaErrors;               /**< Number of DMA errors since init */
00430     Xuint32 FifoErrors;              /**< Number of FIFO errors since init */
00431     Xuint32 RecvInterrupts;          /**< Number of receive interrupts */
00432     Xuint32 XmitInterrupts;          /**< Number of transmit interrupts */
00433     Xuint32 EmacInterrupts;          /**< Number of MAC (device) interrupts */
00434     Xuint32 TotalIntrs;              /**< Total interrupts */
00435 } XGemac_Stats;
00436

```

```

00437 /**
00438  * Ethernet statistics Soft - no hw counter so maintained in instance.
00439  */
00440 typedef struct
00441 {
00442     Xuint32 XmitOverrunErrors;        /**< Number of transmit overrun errors */
00443     Xuint32 SlotLengthErrors;        /**< Number of frames received with
00444                                     slot length errors */
00445     Xuint32 RecvOverrunErrors;       /**< Number of frames discarded due
00446                                     to overrun errors */
00447     Xuint32 RecvUnderrunErrors;     /**< Number of recv underrun errors */
00448     Xuint32 RecvMissedFrameErrors;  /**< Number of frames missed by MAC */
00449     Xuint32 RecvCollisionErrors;    /**< Number of frames discarded due
00450                                     to collisions */
00451     Xuint32 RecvLengthFieldErrors;  /**< Number of frames discarded with
00452                                     invalid length field */
00453     Xuint32 RecvShortErrors;        /**< Number of short frames discarded */
00454     Xuint32 RecvLongErrors;        /**< Number of long frames discarded */
00455     Xuint32 DmaErrors;              /**< Number of DMA errors since init */
00456     Xuint32 FifoErrors;             /**< Number of FIFO errors since init */
00457     Xuint32 RecvInterrupts;        /**< Number of receive interrupts */
00458     Xuint32 XmitInterrupts;        /**< Number of transmit interrupts */
00459     Xuint32 EmacInterrupts;        /**< Number of MAC (device) interrupts */
00460     Xuint32 TotalIntrs;             /**< Total interrupts */
00461 } XGemac_SoftStats;
00462
00463 /**
00464  * This typedef contains configuration information for a device.
00465  */
00466 typedef struct
00467 {
00468     Xuint16 DeviceId;                /**< Unique ID of device */
00469     Xuint32 BaseAddress;             /**< Register base address */
00470     Xuint8 IpIfDmaConfig;           /**< IPIF/DMA hardware configuration */
00471     Xboolean HasMii;                /**< Does device support MII? */
00472 } XGemac_Config;
00473
00474
00475
00476 /** @name Typedefs for callbacks
00477  * Callback functions.
00478  * @{
00479  */
00480 /**
00481  * Callback when data is sent or received with scatter-gather DMA.
00482  *
00483  * @param CallbackRef is a callback reference passed in by the upper layer
00484  *         when setting the callback functions, and passed back to the upper
00485  *         layer when the callback is invoked.
00486  * @param BdPtr is a pointer to the first buffer descriptor in a list of

```

```

00487 *         buffer descriptors.
00488 * @param NumBds is the number of buffer descriptors in the list pointed
00489 *         to by BdPtr.
00490 */
00491 typedef void (*XGemac_SgHandler)(void *CallBackRef, XBufDescriptor *BdPtr,
00492                                 Xuint32 NumBds);
00493
00494 /**
00495 * Callback when data is sent or received with direct FIFO communication.
00496 * The user typically defines two callacks, one for send and one for receive.
00497 *
00498 * @param CallBackRef is a callback reference passed in by the upper layer
00499 *         when setting the callback functions, and passed back to the upper
00500 *         layer when the callback is invoked.
00501 */
00502 typedef void (*XGemac_FifoHandler)(void *CallBackRef);
00503
00504 /**
00505 * Callback when an asynchronous error occurs.
00506 *
00507 * @param CallBackRef is a callback reference passed in by the upper layer
00508 *         when setting the callback functions, and passed back to the upper
00509 *         layer when the callback is invoked.
00510 * @param ErrorCode is a Xilinx error code defined in xstatus.h. Also see
00511 *         XGemac_SetErrorHandler() for a description of possible errors.
00512 */
00513 typedef void (*XGemac_ErrorHandler)(void *CallBackRef, XStatus ErrorCode);
00514 /*@}*/
00515
00516 /**
00517 * The XGemac driver instance data. The user is required to allocate a
00518 * variable of this type for every EMAC device in the system. A pointer
00519 * to a variable of this type is then passed to the driver API functions.
00520 */
00521 typedef struct
00522 {
00523     Xuint32 BaseAddress;           /* Base address (of IPIF) */
00524     Xuint32 IsStarted;            /* Device is currently started */
00525     Xuint32 IsReady;              /* Device is initialized and ready */
00526     Xboolean IsPolled;            /* Device is in polled mode */
00527     Xuint8 IpIfDmaConfig;         /* IPIF/DMA hardware configuration */
00528     Xboolean HasMii;              /* Does device support MII? */
00529     Xboolean HasMulticastHash;    /* Does device support multicast hash table?
00530 */
00531
00532     XGemac_SoftStats Stats;
00533     XPacketFifoV200a RecvFifo;    /* FIFO used by receive DMA channel */
00534     XPacketFifoV200a SendFifo;    /* FIFO used by send DMA channel */
00535
00536     /*
00537      * Callbacks
00538      */

```

```

00537     */
00538     XGemac_FifoHandler FifoRecvHandler; /* callback for non-DMA interrupts */
00539     void *FifoRecvRef;
00540     XGemac_FifoHandler FifoSendHandler; /* callback for non-DMA interrupts */
00541     void *FifoSendRef;
00542     XGemac_ErrorHandler ErrorHandler; /* callback for asynchronous errors */
00543     void *ErrorRef;
00544
00545     XDmaChannel RecvChannel; /* DMA receive channel driver */
00546     XDmaChannel SendChannel; /* DMA send channel driver */
00547
00548     XGemac_SgHandler SgRecvHandler; /* callback for scatter-gather DMA */
00549     void *SgRecvRef;
00550     XGemac_SgHandler SgSendHandler; /* callback for scatter-gather DMA */
00551     void *SgSendRef;
00552 } XGemac;
00553
00554
00555 /***** Macros (Inline Functions) Definitions *****/
00556
00557 /*****
00558 /**
00559 *
00560 * This macro determines if the device is currently configured for
00561 * scatter-gather DMA.
00562 *
00563 * @param InstancePtr is a pointer to the XGemac instance to be worked on.
00564 *
00565 * @return
00566 *
00567 * Boolean XTRUE if the device is configured for scatter-gather DMA, or XFALSE
00568 * if it is not.
00569 *
00570 * @note
00571 *
00572 * Signature: Xboolean XGemac_mIsSgDma(XGemac *InstancePtr)
00573 *
00574 *****/
00575 #define XGemac_mIsSgDma(InstancePtr) \
00576     ((InstancePtr)->IpIfDmaConfig == XGE_CFG_DMA_SG)
00577
00578 /*****
00579 /**
00580 *
00581 * This macro determines if the device is currently configured for simple DMA.
00582 *
00583 * @param InstancePtr is a pointer to the XGemac instance to be worked on.
00584 *
00585 * @return
00586 *
00587 * Boolean XTRUE if the device is configured for simple DMA, or XFALSE otherwise
00588 *

```

```

00589 * @note
00590 *
00591 * Signature: Xboolean XGemac_mIsSimpleDma(XGemac *InstancePtr)
00592 *
00593 *****/
00594 #define XGemac_mIsSimpleDma(InstancePtr) \
00595     ((InstancePtr)->IpIfDmaConfig == XGE_CFG_SIMPLE_DMA)
00596
00597 *****/
00598 /**
00599 *
00600 * This macro determines if the device is currently configured with DMA (either
00601 * simple DMA or scatter-gather DMA)
00602 *
00603 * @param InstancePtr is a pointer to the XGemac instance to be worked on.
00604 *
00605 * @return
00606 *
00607 * Boolean XTRUE if the device is configured with DMA, or XFALSE otherwise
00608 *
00609 * @note
00610 *
00611 * Signature: Xboolean XGemac_mIsDma(XGemac *InstancePtr)
00612 *
00613 *****/
00614 #define XGemac_mIsDma(InstancePtr) \
00615     (XGemac_mIsSimpleDma(InstancePtr) || XGemac_mIsSgDma(InstancePtr))
00616
00617
00618 *****/
00619
00620 /**
00621 * Initialization functions in xgemac.c
00622 */
00623 XStatus XGemac_Initialize(XGemac *InstancePtr, Xuint16 DeviceId);
00624 XStatus XGemac_Start(XGemac *InstancePtr);
00625 XStatus XGemac_Stop(XGemac *InstancePtr);
00626 void XGemac_Reset(XGemac *InstancePtr);
00627 XGemac_Config *XGemac_LookupConfig(Xuint16 DeviceId);
00628
00629 /**
00630 * Diagnostic functions in xgemac_selftest.c
00631 */
00632 XStatus XGemac_SelfTest(XGemac *InstancePtr);
00633
00634 /**
00635 * Polled functions in xgemac_polled.c
00636 */
00637 XStatus XGemac_PollSend(XGemac *InstancePtr, Xuint8 *BufPtr, Xuint32
ByteCount);
00638 XStatus XGemac_PollRecv(XGemac *InstancePtr, Xuint8 *BufPtr,

```



```

00639             Xuint32 *ByteCountPtr);
00640
00641 /*
00642  * Interrupts with scatter-gather DMA functions in xgemac_intr_dma.c
00643  * This functionality is not yet supported, calling any of these functions
00644  * will result in an assert.
00645  */
00646 XStatus XGmac_SgSend(XGmac *InstancePtr, XBufDescriptor *BdPtr, int Delay);
00647 XStatus XGmac_SgRecv(XGmac *InstancePtr, XBufDescriptor *BdPtr);
00648 XStatus XGmac_SetPktThreshold(XGmac *InstancePtr, Xuint32 Direction,
00649                               Xuint8 Threshold);
00650 XStatus XGmac_GetPktThreshold(XGmac *InstancePtr, Xuint32 Direction,
00651                               Xuint8 *ThreshPtr);
00652 XStatus XGmac_SetPktWaitBound(XGmac *InstancePtr, Xuint32 Direction,
00653                               Xuint32 TimerValue);
00654 XStatus XGmac_GetPktWaitBound(XGmac *InstancePtr, Xuint32 Direction,
00655                               Xuint32 *WaitPtr);
00656 XStatus XGmac_SetSgRecvSpace(XGmac *InstancePtr, Xuint32 *MemoryPtr,
00657                               Xuint32 ByteCount);
00658 XStatus XGmac_SetSgSendSpace(XGmac *InstancePtr, Xuint32 *MemoryPtr,
00659                               Xuint32 ByteCount);
00660 void XGmac_SetSgRecvHandler(XGmac *InstancePtr, void *CallBackRef,
00661                             XGmac_SgHandler FuncPtr);
00662 void XGmac_SetSgSendHandler(XGmac *InstancePtr, void *CallBackRef,
00663                             XGmac_SgHandler FuncPtr);
00664
00665 void XGmac_IntrHandlerDma(void *InstancePtr);           /* interrupt handler */
00666
00667 /*
00668  * Interrupts with direct FIFO functions in xgemac_intr_fifo.c
00669  */
00670 XStatus XGmac_FifoSend(XGmac *InstancePtr, Xuint8 *BufPtr, Xuint32
ByteCount);
00671 XStatus XGmac_FifoRecv(XGmac *InstancePtr, Xuint8 *BufPtr,
00672                       Xuint32 *ByteCountPtr);
00673 void XGmac_SetFifoRecvHandler(XGmac *InstancePtr, void *CallBackRef,
00674                               XGmac_FifoHandler FuncPtr);
00675 void XGmac_SetFifoSendHandler(XGmac *InstancePtr, void *CallBackRef,
00676                               XGmac_FifoHandler FuncPtr);
00677
00678 void XGmac_IntrHandlerFifo(void *InstancePtr);         /* interrupt handler */
00679
00680 /*
00681  * General interrupt-related functions in xgemac_intr.c
00682  */
00683 void XGmac_SetErrorHandler(XGmac *InstancePtr, void *CallBackRef,
00684                             XGmac_ErrorHandler FuncPtr);
00685
00686 /*

```

```
00687 * MAC configuration in xgemac_options.c
00688 */
00689 XStatus XGemac_SetOptions(XGemac *InstancePtr, Xuint32 OptionFlag);
00690 Xuint32 XGemac_GetOptions(XGemac *InstancePtr);
00691 XStatus XGemac_SetMacAddress(XGemac *InstancePtr, Xuint8 *AddressPtr);
00692 void XGemac_GetMacAddress(XGemac *InstancePtr, Xuint8 *BufferPtr);
00693 XStatus XGemac_SetInterframeGap(XGemac *InstancePtr, Xuint8 Ifg);
00694 void XGemac_GetInterframeGap(XGemac *InstancePtr, Xuint8 *IfgPtr);
00695
00696 /*
00697 * Multicast functions in xgemac_multicast.c
00698 */
00699 XStatus XGemac_MulticastAdd(XGemac *InstancePtr, Xuint8 *AddressPtr);
00700 XStatus XGemac_MulticastClear(XGemac *InstancePtr);
00701
00702
00703 /*
00704 * Statistics in xgemac_stats.c
00705 */
00706 void XGemac_GetStats(XGemac *InstancePtr, XGemac_Stats *StatsPtr);
00707 void XGemac_ClearStats(XGemac *InstancePtr);
00708
00709
00710 #endif /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

XGmac_Stats Struct Reference

```
#include <xgemac.h>
```

Detailed Description

Ethernet statistics (see [XGmac_GetStats\(\)](#) and [XGmac_ClearStats\(\)](#))

Data Fields

[Xuint32 XmitFramesMSL](#)

[Xuint32 XmitFrames](#)

[Xuint32 XmitBytesMSL](#)

[Xuint32 XmitBytes](#)

[Xuint32 XmitLateCollisionErrorsMSL](#)

[Xuint32 XmitLateCollisionErrors](#)

[Xuint32 XmitExcessDeferralMSL](#)

[Xuint32 XmitExcessDeferral](#)

[Xuint32 XmitOverrunErrors](#)

[Xuint32 XmitUnderrunErrorsMSL](#)

[Xuint32 XmitUnderrunErrors](#)

[Xuint32 RecvFramesMSL](#)

[Xuint32 RecvFrames](#)

[Xuint32 RecvBytesMSL](#)

[Xuint32 RecvBytes](#)

[Xuint32 RecvFcsErrorsMSL](#)

[Xuint32 RecvFcsErrors](#)

[Xuint32 SlotLengthErrors](#)

Xuint32 RecvOverrunErrors
Xuint32 RecvUnderrunErrors
Xuint32 RecvMissedFrameErrors
Xuint32 RecvCollisionErrorsMSL
Xuint32 RecvCollisionErrors
Xuint32 RecvLengthFieldErrors
Xuint32 RecvShortErrors
Xuint32 RecvLongErrors
Xuint32 DmaErrors
Xuint32 FifoErrors
Xuint32 RecvInterrupts
Xuint32 XmitInterrupts
Xuint32 EmacInterrupts
Xuint32 TotalIntrs

Field Documentation

Xuint32 XGmac_Stats::DmaErrors

Number of DMA errors since init

Xuint32 XGmac_Stats::EmacInterrupts

Number of MAC (device) interrupts

Xuint32 XGmac_Stats::FifoErrors

Number of FIFO errors since init

Xuint32 XGmac_Stats::RecvBytes

Number of bytes received

Xuint32 XGmac_Stats::RecvBytesMSL

Number of bytes received upper 32 bits

Xuint32 XGmac_Stats::RecvCollisionErrors

Number of frames discarded due to collisions

Xuint32 XGemac_Stats::RecvCollisionErrorsMSL

Number of frames discarded due to collisions upper 32 bits

Xuint32 XGemac_Stats::RecvFcsErrors

Number of frames discarded due to FCS errors

Xuint32 XGemac_Stats::RecvFcsErrorsMSL

Number of frames discarded due to FCS errors upper 32 bits

Xuint32 XGemac_Stats::RecvFrames

Number of frames received

Xuint32 XGemac_Stats::RecvFramesMSL

Number of frames received upper 32 bits

Xuint32 XGemac_Stats::RecvInterrupts

Number of receive interrupts

Xuint32 XGemac_Stats::RecvLengthFieldErrors

Number of frames discarded with invalid length field

Xuint32 XGemac_Stats::RecvLongErrors

Number of long frames discarded

Xuint32 XGemac_Stats::RecvMissedFrameErrors

Number of frames missed by MAC

Xuint32 XGemac_Stats::RecvOverrunErrors

Number of frames discarded due to overrun errors

Xuint32 XGemac_Stats::RecvShortErrors

Number of short frames discarded

Xuint32 XGemac_Stats::RecvUnderrunErrors

Number of recv underrun errors

Xuint32 XGemac_Stats::SlotLengthErrors

Number of frames received with slot length errors

Xuint32 XGemac_Stats::TotalIntrs

Total interrupts

Xuint32 XGemac_Stats::XmitBytes

Number of bytes transmitted

Xuint32 XGemac_Stats::XmitBytesMSL

Number of bytes transmitted upper 32 bits

Xuint32 XGemac_Stats::XmitExcessDeferral

Number of transmission failures due to excess collision deferrals

Xuint32 XGemac_Stats::XmitExcessDeferralMSL

Number of transmission failures due to excess collision deferrals upper 32 bits

Xuint32 XGemac_Stats::XmitFrames

Number of frames transmitted

Xuint32 XGemac_Stats::XmitFramesMSL

Number of frames transmitted upper 32 bits

Xuint32 XGemac_Stats::XmitInterrupts

Number of transmit interrupts

Xuint32 XGmac_Stats::XmitLateCollisionErrors

Number of transmission failures due to late collisions

Xuint32 XGmac_Stats::XmitLateCollisionErrorsMSL

Number of transmission failures due to late collisions upper 32 bits

Xuint32 XGmac_Stats::XmitOverrunErrors

Number of transmit overrun errors upper 32 bits

Xuint32 XGmac_Stats::XmitUnderrunErrors

Number of transmit underrun errors

Xuint32 XGmac_Stats::XmitUnderrunErrorsMSL

Number of transmit underrun errors

The documentation for this struct was generated from the following file:

- [gemac/v1_00_c/src/xgemac.h](#)

XGemac_SoftStats Struct Reference

```
#include <xgemac.h>
```

Detailed Description

Ethernet statistics Soft - no hw counter so maintained in instance.

Data Fields

[Xuint32 XmitOverrunErrors](#)

[Xuint32 SlotLengthErrors](#)

[Xuint32 RecvOverrunErrors](#)

[Xuint32 RecvUnderrunErrors](#)

[Xuint32 RecvMissedFrameErrors](#)

[Xuint32 RecvCollisionErrors](#)

[Xuint32 RecvLengthFieldErrors](#)

[Xuint32 RecvShortErrors](#)

[Xuint32 RecvLongErrors](#)

[Xuint32 DmaErrors](#)

[Xuint32 FifoErrors](#)

[Xuint32 RecvInterrupts](#)

[Xuint32 XmitInterrupts](#)

[Xuint32 EmacInterrupts](#)

[Xuint32 TotalIntrs](#)

Field Documentation

Xuint32 XGemac_SoftStats::DmaErrors

Number of DMA errors since init

Xuint32 XGemac_SoftStats::EmacInterrupts

Number of MAC (device) interrupts

Xuint32 XGemac_SoftStats::FifoErrors

Number of FIFO errors since init

Xuint32 XGemac_SoftStats::RecvCollisionErrors

Number of frames discarded due to collisions

Xuint32 XGemac_SoftStats::RecvInterrupts

Number of receive interrupts

Xuint32 XGemac_SoftStats::RecvLengthFieldErrors

Number of frames discarded with invalid length field

Xuint32 XGemac_SoftStats::RecvLongErrors

Number of long frames discarded

Xuint32 XGemac_SoftStats::RecvMissedFrameErrors

Number of frames missed by MAC

Xuint32 XGemac_SoftStats::RecvOverrunErrors

Number of frames discarded due to overrun errors

Xuint32 XGemac_SoftStats::RecvShortErrors

Number of short frames discarded

Xuint32 XGemac_SoftStats::RecvUnderrunErrors

Number of recv underrun errors

Xuint32 XGemac_SoftStats::SlotLengthErrors

Number of frames received with slot length errors

Xuint32 XGemac_SoftStats::TotalIntrs

Total interrupts

Xuint32 XGemac_SoftStats::XmitInterrupts

Number of transmit interrupts

Xuint32 XGemac_SoftStats::XmitOverrunErrors

Number of transmit overrun errors

The documentation for this struct was generated from the following file:

- [gemac/v1_00_c/src/xgemac.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XGmac_Config Struct Reference

```
#include <xgemac.h>
```

Detailed Description

This typedef contains configuration information for a device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xuint8 IpIfDmaConfig

Xboolean HasMii

Field Documentation

Xuint32 XGmac_Config::BaseAddress

Register base address

Xuint16 XGmac_Config::DeviceId

Unique ID of device

Xboolean XGmac_Config::HasMii

Does device support MII?

Xuint8 XGmac_Config::IpIfDmaConfig

IPIF/DMA hardware configuration

The documentation for this struct was generated from the following file:

- [gemac/v1_00_c/src/xgemac.h](#)
-

Generated on 29 May 2003 for Xilinx Device Drivers

XGmac Struct Reference

```
#include <xgmac.h>
```

Detailed Description

The XGmac driver instance data. The user is required to allocate a variable of this type for every EMAC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [gemac/v1_00_c/src/xgmac.h](#)
-

gemac/v1_00_c/src/xgemac.c File Reference

Detailed Description

The **XGemac** driver. Functions in this file are the minimum required functions for this driver. See [xgemac.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	DMA mods

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

```
XStatus XGemac_Initialize (XGemac *InstancePtr, Xuint16 DeviceId)  
XStatus XGemac_Start (XGemac *InstancePtr)  
XStatus XGemac_Stop (XGemac *InstancePtr)  
void XGemac_Reset (XGemac *InstancePtr)
```

XStatus **XGmac_SetMacAddress** (**XGmac** *InstancePtr, **Xuint8** *AddressPtr)

void **XGmac_GetMacAddress** (**XGmac** *InstancePtr, **Xuint8** *BufferPtr)

XGmac_Config * **XGmac_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void XGmac_GetMacAddress( XGmac * InstancePtr,  
                        Xuint8 * BufferPtr  
                        )
```

Get the MAC address for this driver/device.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufferPtr is an output parameter, and is a pointer to a buffer into which the current MAC address will be copied. The buffer must be at least 6 bytes.

Returns:

None.

Note:

None.

```
XStatus XGmac_Initialize( XGmac * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initialize a specific **XGmac** instance/driver. The initialization entails:

- Initialize fields of the **XGmac** structure
- Clear the Ethernet statistics for this device
- Initialize the IPIF component with its register base address
- Configure the FIFO components with their register base addresses.
- If the device is configured with DMA, configure the DMA channel components with their register base addresses. At some later time, memory pools for the scatter-gather descriptor lists are to be passed to the driver.
- Reset the Ethernet MAC

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XGemac** instance. Passing in a device id associates the generic **XGemac** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

XGemac_Config* XGemac_LookupConfig(Xuint16 DeviceId)

Lookup the device configuration based on the unique device ID. The table EmacConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device being looked up.

Returns:

A pointer to the configuration table entry corresponding to the given device ID, or XNULL if no match is found.

Note:

None.

void XGemac_Reset(XGemac * InstancePtr)

Reset the Ethernet MAC. This is a graceful reset in that the device is stopped first. Resets the DMA channels, the FIFOs, the transmitter, and the receiver. The PHY is not reset. Any frames in the scatter-gather descriptor lists will remain in the lists. The side effect of doing this is that after a reset and following a restart of the device, frames that were in the list before the reset may be transmitted or received. Reset must only be called after the driver has been initialized.

The configuration after this reset is as follows:

- Disabled transmitter and receiver
- Enabled PHY (the PHY is not reset)
- MAC transmitter does pad insertion, FCS insertion, and source address overwrite.
- MAC receiver does not strip padding or FCS
- Interframe Gap as recommended by IEEE Std. 802.3 (96 bit times)
- Unicast addressing enabled
- Broadcast addressing enabled
- Multicast addressing disabled (addresses are preserved)
- Promiscuous addressing disabled
- Default packet threshold and packet wait bound register values for scatter-gather DMA operation
- MAC address of all zeros

The upper layer software is responsible for re-configuring (if necessary) and restarting the MAC after the reset. Note that the PHY is not reset. PHY control is left to the upper layer software. Note also that driver statistics are not cleared on reset. It is up to the upper layer software to clear the statistics if needed.

When a reset is required due to an internal error, the driver notifies the upper layer software of this need through the ErrorHandler callback and specific status codes. The upper layer software is responsible for calling this Reset function and then re-configuring the device.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus XGemac_SetMacAddress( XGemac * InstancePtr,  
                               Xuint8 * AddressPtr  
                               )
```

Set the MAC address for this driver/device. The address is a 48-bit value. The device must be stopped before calling this function.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

AddressPtr is a pointer to a 6-byte MAC address.

Returns:

- XST_SUCCESS if the MAC address was set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

None.

```
XStatus XGemac_Start( XGemac * InstancePtr )
```

Start the Ethernet controller as follows:

- If in interrupt driven mode
 - Set the internal interrupt enable registers appropriately
 - Enable interrupts within the device itself. Note that connection of the driver's interrupt handler to the interrupt source (typically done using the interrupt controller component) is done by the higher layer software.
 - If the device is configured with DMA, start the DMA channels if the descriptor lists are not empty
- Enable the transmitter
- Enable the receiver

The PHY is enabled after driver initialization. We assume the upper layer software has configured it and the GEMAC appropriately before this function is called.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was started successfully

- XST_NO_CALLBACK if a callback function has not yet been registered using the SetxxxHandler function. This is required if in interrupt mode.
- XST_DEVICE_IS_STARTED if the device is already started
- XST_DMA_SG_NO_LIST if configured for scatter-gather DMA and a descriptor list has not yet been created for the send or receive channel.

Note:

The driver tries to match the hardware configuration. So if the hardware is configured with scatter-gather DMA, the driver expects to start the scatter-gather channels and expects that the user has set up the buffer descriptor lists already. If the user expects to use the driver in a mode different than how the hardware is configured, the user should modify the configuration table to reflect the mode to be used. Modifying the config table is a workaround for now until we get some experience with how users are intending to use the hardware in its different configurations. For example, if the hardware is built with scatter-gather DMA but the user is intending to use only simple DMA, the user either needs to modify the config table as a workaround or rebuild the hardware with only simple DMA.

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

XStatus XGemac_Stop(XGemac * InstancePtr)

Stop the Ethernet MAC as follows:

- If the device is configured with scatter-gather DMA, stop the DMA channels (wait for acknowledgment of stop)
- Disable the transmitter and receiver
- Disable interrupts if not in polled mode (the higher layer software is responsible for disabling interrupts at the interrupt controller)

The PHY is left enabled after a Stop is called.

If the device is configured for scatter-gather DMA, the DMA engine stops at the next buffer descriptor in its list. The remaining descriptors in the list are not removed, so anything in the list will be transmitted or received when the device is restarted. The side effect of doing this is that the last buffer descriptor processed by the DMA engine before stopping may not be the last descriptor in the Ethernet frame. So when the device is restarted, a partial frame (i.e., a bad frame) may be transmitted/received. This is only a concern if a frame can span multiple buffer descriptors, which is dependent on the size of the network buffers.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Returns:

- XST_SUCCESS if the device was stopped successfully
- XST_DEVICE_IS_STOPPED if the device is already stopped

Note:

This function makes use of internal resources that are shared between the Start, Stop, and SetOptions functions. So if one task might be setting device options while another is trying to start the device, the user is required to provide protection of this shared data (typically using a semaphore).

gemac/v1_00_c/src/xgemac_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xgemac_i.h,v 1.1 2003/05/29 19:43:52 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file gemac/v1_00_c/src/xgemac_i.h
00026 *
00027 * This header file contains internal identifiers, which are those shared
00028 * between XGemac components.  The identifiers in this file are not intended for
00029 * use external to the driver.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date      Changes
00035 * -----
00036 * 1.00a  ecm    01/13/03  First release
00037 * 1.00b  ecm    03/25/03  Revision update
00038 * 1.00c  rmm    05/28/03  Revision update
00039 * </pre>
00040 *
00041 *****/
00042
```

```

00043 #ifndef XGEMAC_I_H /* prevent circular inclusions */
00044 #define XGEMAC_I_H /* by using protection macros */
00045
00046 /***** Include Files *****/
00047
00048 #include "xgemac.h"
00049 #include "xgemac_1.h"
00050
00051 /***** Constant Definitions *****/
00052
00053 /*
00054  * Default buffer descriptor control word masks. The default send BD control
00055  * is set for incrementing the source address by one for each byte transferred,
00056  * and specify that the destination address (FIFO) is local to the device. The
00057  * default receive BD control is set for incrementing the destination address
00058  * by one for each byte transferred, and specify that the source address is
00059  * local to the device.
00060  */
00061 #define XGE_DFT_SEND_BD_MASK (XDC_DMOCR_SOURCE_INCR_MASK | \
00062                             XDC_DMOCR_DEST_LOCAL_MASK)
00063 #define XGE_DFT_RECV_BD_MASK (XDC_DMOCR_DEST_INCR_MASK | \
00064                              XDC_DMOCR_SOURCE_LOCAL_MASK)
00065
00066 /*
00067  * Masks for the IPIF Device Interrupt enable and status registers.
00068  */
00069 #define XGE_IPIF_EMAC_MASK 0x00000004UL /* MAC interrupt */
00070 #define XGE_IPIF_SEND_DMA_MASK 0x00000008UL /* Send DMA interrupt */
00071 #define XGE_IPIF_RECV_DMA_MASK 0x00000010UL /* Receive DMA interrupt */
00072 #define XGE_IPIF_RECV_FIFO_MASK 0x00000020UL /* Receive FIFO interrupt */
00073 #define XGE_IPIF_SEND_FIFO_MASK 0x00000040UL /* Send FIFO interrupt */
00074
00075 /*
00076  * Default IPIF Device Interrupt mask when configured for DMA
00077  */
00078 #define XGE_IPIF_DMA_DFT_MASK (XGE_IPIF_SEND_DMA_MASK | \
00079                               XGE_IPIF_RECV_DMA_MASK | \
00080                               XGE_IPIF_EMAC_MASK | \
00081                               XGE_IPIF_SEND_FIFO_MASK | \
00082                               XGE_IPIF_RECV_FIFO_MASK)
00083
00084 /*
00085  * Default IPIF Device Interrupt mask when configured without DMA
00086  */
00087 #define XGE_IPIF_FIFO_DFT_MASK (XGE_IPIF_EMAC_MASK | \
00088                                 XGE_IPIF_SEND_FIFO_MASK | \
00089                                 XGE_IPIF_RECV_FIFO_MASK)
00090
00091 #define XGE_IPIF_DMA_DEV_INTR_COUNT 7 /* Number of interrupt sources */
00092 #define XGE_IPIF_FIFO_DEV_INTR_COUNT 5 /* Number of interrupt sources */
00093 #define XGE_IPIF_DEVICE_INTR_COUNT 7 /* Number of interrupt sources */
00094 #define XGE_IPIF_IP_INTR_COUNT 22 /* Number of MAC interrupts */

```



```

00148
00149 /***** Type Definitions *****/
00150
00151 /***** Macros (Inline Functions) Definitions *****/
00152
00153
00154 /*****
00155 /*
00156 *
00157 * Clears a structure of given size, in bytes, by setting each byte to 0.
00158 *
00159 * @param StructPtr is a pointer to the structure to be cleared.
00160 * @param NumBytes is the number of bytes in the structure.
00161 *
00162 * @return
00163 *
00164 * None.
00165 *
00166 * @note
00167 *
00168 * Signature: void XGemac_mClearStruct(Xuint8 *StructPtr, unsigned int NumBytes)
00169 *
00170 *****/
00171 #define XGemac_mClearStruct(StructPtr, NumBytes) \
00172 { \
00173     int i; \
00174     Xuint8 *BytePtr = (Xuint8 *) (StructPtr); \
00175     for (i=0; i < (unsigned int) (NumBytes); i++) \
00176     { \
00177         *BytePtr++ = 0; \
00178     } \
00179 }
00180
00181 /***** Variable Definitions *****/
00182
00183 extern XGemac_Config XGemac_ConfigTable[];
00184
00185 /***** Function Prototypes *****/
00186
00187 void XGemac_CheckEmacError(XGemac *InstancePtr, Xuint32 IntrStatus);
00188 void XGemac_CheckFifoRecvError(XGemac *InstancePtr);
00189 void XGemac_CheckFifoSendError(XGemac *InstancePtr);
00190
00191 #endif /* end of protection macro */

```


gemac/v1_00_c/src/xgemac_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between **XGemac** components. The identifiers in this file are not intended for use external to the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Revision update

```
#include "xgemac.h"  
#include "xgemac_1.h"
```

[Go to the source code of this file.](#)

Variables

[XGemac_Config](#) [XGemac_ConfigTable](#) []

Variable Documentation

XGmac_Config XGmac_ConfigTable[]()

This table contains configuration information for each GEMAC device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

gemac/v1_00_c/src/xgemac_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xgemac_l.h,v 1.1 2003/05/29 19:43:53 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file gemac/v1_00_c/src/xgemac_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  High-level driver functions
00029 * are defined in xgemac.h.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date        Changes
00035 * -----
00036 * 1.00a  ecm    01/13/03  First release
00037 * 1.00b  ecm    03/25/03  Revision update
00038 * 1.00c  rmm    05/28/03  Removed tabs characters in file, added auto negotiate
00039 *                               bit to ECR constants.
00040 * </pre>
00041 *
00042 *****/
```

```

00043
00044 #ifndef XGEMAC_L_H /* prevent circular inclusions */
00045 #define XGEMAC_L_H /* by using protection macros */
00046
00047 /***** Include Files *****/
00048
00049 #include "xbasic_types.h"
00050 #include "xio.h"
00051
00052 /***** Constant Definitions *****/
00053
00054 /* Offset of the MAC registers from the IPIF base address */
00055 #define XGE_REG_OFFSET      0x1000UL
00056
00057 /*
00058  * Register offsets for the Ethernet MAC. Each register is 32 bits.
00059  */
00060 #define XGE_EMIR_OFFSET      (XGE_REG_OFFSET + 0x0)    /* GEMAC Module ID */
00061 #define XGE_ECR_OFFSET      (XGE_REG_OFFSET + 0x4)    /* MAC Control */
00062 #define XGE_IFGP_OFFSET     (XGE_REG_OFFSET + 0x8)    /* Interframe Gap */
00063 #define XGE_SAH_OFFSET      (XGE_REG_OFFSET + 0xC)    /* Station addr, high */
00064 #define XGE_SAL_OFFSET      (XGE_REG_OFFSET + 0x10)   /* Station addr, low */
00065 #define XGE_MGTCR_OFFSET    (XGE_REG_OFFSET + 0x14)   /* MII mgmt control */
00066 #define XGE_MGTDNR_OFFSET   (XGE_REG_OFFSET + 0x18)   /* MII mgmt data */
00067 #define XGE_RPLR_OFFSET     (XGE_REG_OFFSET + 0x1C)   /* Rx packet length */
00068 #define XGE_TPLR_OFFSET     (XGE_REG_OFFSET + 0x20)   /* Tx packet length */
00069 #define XGE_TSR_OFFSET      (XGE_REG_OFFSET + 0x24)   /* Tx status */
00070 #define XGE_TPPR_OFFSET     (XGE_REG_OFFSET + 0x28)   /* Tx Pause Pkt */
00071 #define XGE_CEAH_OFFSET     (XGE_REG_OFFSET + 0x2C)   /* CAM Entry Address High */
00072 #define XGE_CEAL_OFFSET     (XGE_REG_OFFSET + 0x30)   /* CAM Entry Address Low */
00073 #define XGE_SLEC_OFFSET     (XGE_REG_OFFSET + 0x34)   /* Slot Length errors */
00074 #define XGE_TEDC_OFFSET     (XGE_REG_OFFSET + 0x38)   /* Transmit excess
00075                                     * deferral cnt */
00076
00077 /*
00078  * Register offsets for the IPIF components
00079  */
00080 #define XGE_ISR_OFFSET      0x20UL                    /* Interrupt status */
00081
00082 #define XGE_DMA_OFFSET      0x2300UL
00083 #define XGE_DMA_SEND_OFFSET (XGE_DMA_OFFSET + 0x0)   /* DMA send channel */
00084 #define XGE_DMA_RECV_OFFSET (XGE_DMA_OFFSET + 0x40)  /* DMA recv channel */
00085
00086 #define XGE_PFIFO_OFFSET    0x2000UL
00087 #define XGE_PFIFO_TXREG_OFFSET (XGE_PFIFO_OFFSET + 0x0) /* Tx registers */
00088 #define XGE_PFIFO_RXREG_OFFSET (XGE_PFIFO_OFFSET + 0x10) /* Rx registers */
00089 #define XGE_PFIFO_TXDATA_OFFSET (XGE_PFIFO_OFFSET + 0x100) /* Tx keyhole */
00090 #define XGE_PFIFO_RXDATA_OFFSET (XGE_PFIFO_OFFSET + 0x200) /* Rx keyhole */
00091
00092 /*
00093  * GEMAC Module Identification Register (EMIR)
00094  */

```

```

00095 #define XGE_EMIR_VERSION_MASK      0xFFFFF000UL      /* Device version */
00096 #define XGE_EMIR_TYPE_MASK         0x0000FF00UL      /* Device type */
00097
00098 /*
00099  * GEMAC Control Register (ECR)
00100  */
00101 #define XGE_ECR_FULL_DUPLEX_MASK     0x80000000UL /* Full duplex mode */
00102 #define XGE_ECR_XMIT_RESET_MASK     0x40000000UL /* Reset transmitter */
00103 #define XGE_ECR_XMIT_ENABLE_MASK    0x20000000UL /* Enable transmitter */
00104 #define XGE_ECR_RECV_RESET_MASK     0x10000000UL /* Reset receiver */
00105 #define XGE_ECR_RECV_ENABLE_MASK    0x08000000UL /* Enable receiver */
00106 #define XGE_ECR_PHY_ENABLE_MASK     0x04000000UL /* Enable PHY */
00107 #define XGE_ECR_XMIT_PAD_ENABLE_MASK 0x02000000UL /* Enable xmit pad insert
*/
00108 #define XGE_ECR_XMIT_FCS_ENABLE_MASK 0x01000000UL /* Enable xmit FCS insert
*/
00109 #define XGE_ECR_XMIT_ADDR_INSERT_MASK 0x00800000UL /* Enable xmit source
addr
00110
*/ insertion */
00111 #define XGE_ECR_XMIT_ERROR_INSERT_MASK 0x00400000UL /* Insert xmit error */
00112 #define XGE_ECR_XMIT_ADDR_OVWRT_MASK 0x00200000UL /* Enable xmit source
addr
00113
*/ overwrite */
00114 #define XGE_ECR_LOOPBACK_MASK      0x00100000UL /* Enable internal
00115
*/ loopback */
00116 #define XGE_ECR_RECV_STRIP_ENABLE_MASK 0x00080000UL /* Enable recv pad/fcs
strip */
00117 #define XGE_ECR_UNICAST_ENABLE_MASK 0x00020000UL /* Enable unicast addr */
00118 #define XGE_ECR_MULTI_ENABLE_MASK   0x00010000UL /* Enable multicast addr
*/
00119 #define XGE_ECR_BROAD_ENABLE_MASK   0x00008000UL /* Enable broadcast addr
*/
00120 #define XGE_ECR_PROMISC_ENABLE_MASK 0x00004000UL /* Enable promiscuous
mode */
00121
00122 #define XGE_ECR_REO_ENABLE_MASK     0x00002000UL /* Enable Receive Error
00123
*/ Override */
00124 #define XGE_ECR_RECV_JUMBO_ENABLE_MASK 0x00001000UL /* Enable RX of Jumbo
Frames */
00125 #define XGE_ECR_XMIT_PAUSE_ENABLE_MASK 0x00000800UL /* Enable TX of Pause
Pkts */
00126 #define XGE_ECR_RECV_PAUSE_ENABLE_MASK 0x00000400UL /* Enable RX of Pause
Pkts */
00127 #define XGE_ECR_XMIT_VLAN_ENABLE_MASK 0x00000200UL /* Enable TX of VLAN Pkts
*/
00128 #define XGE_ECR_RECV_VLAN_ENABLE_MASK 0x00000100UL /* Enable RX of VLAN Pkts
*/
00129 #define XGE_ECR_AUTONEG_ENABLE_MASK 0x00000080UL /* Enable PCS/PMA
autonegotiate */
00130
00131 #define XGE_ECR_ADD_HASH_ADDR_MASK   0x00000FFFUL /* Todo - fix */
00132 #define XGE_ECR_CLEAR_HASH_MASK     0x00000000UL /* Todo - fix */
00133

```

```

00134 /*
00135  * Interframe Gap Register (IFGR)
00136  */
00137 #define XGE_IFGP_MASK                0xFF000000UL /* Interframe Gap Mask */
00138 #define XGE_IFGP_SHIFT                24
00139
00140 /*
00141  * Station Address High Register (SAH)
00142  */
00143 #define XGE_SAH_ADDR_MASK            0x0000FFFFUL /* Station address high bytes
*/
00144
00145 /*
00146  * Station Address Low Register (SAL)
00147  */
00148 #define XGE_SAL_ADDR_MASK            0xFFFFFFFFUL /* Station address low bytes
*/
00149
00150 /*
00151  * MII Management Control Register (MGTCR)
00152  */
00153 #define XGE_MGTCR_START_MASK          0x80000000UL /* Start/Busy */
00154 #define XGE_MGTCR_RW_NOT_MASK         0x40000000UL /* Read/Write Not (direction)
*/
00155 #define XGE_MGTCR_PHY_ADDR_MASK       0x3E000000UL /* PHY address */
00156 #define XGE_MGTCR_PHY_ADDR_SHIFT     25           /* PHY address shift */
00157 #define XGE_MGTCR_REG_ADDR_MASK       0x01F00000UL /* Register address */
00158 #define XGE_MGTCR_REG_ADDR_SHIFT     20           /* Register addr shift */
00159 #define XGE_MGTCR_MII_ENABLE_MASK     0x00080000UL /* Enable MII from GEMAC */
00160 #define XGE_MGTCR_RD_ERROR_MASK       0x00040000UL /* MII mgmt read error */
00161
00162 /*
00163  * MII Management Data Register (MGTD R)
00164  */
00165 #define XGE_MGTD R_DATA_MASK          0x0000FFFFUL /* MII data */
00166
00167 /*
00168  * Receive Packet Length Register (RPLR)
00169  */
00170 #define XGE_RPLR_LENGTH_MASK          0x0000FFFFUL /* Receive packet length */
00171
00172 /*
00173  * Transmit Packet Length Register (TPLR)
00174  */
00175 #define XGE_TPLR_LENGTH_MASK          0x0000FFFFUL /* Transmit packet length */
00176
00177 /*
00178  * Transmit Status Register (TSR)
00179  */
00180 #define XGE_TSR_EXCESS_DEFERRAL_MASK  0x80000000UL /* Transmit excess deferral
*/
00181 #define XGE_TSR_FIFO_UNDERRUN_MASK    0x40000000UL /* Packet FIFO underrun */

```

```

00182 #define XGE_TSR_ATTEMPTS_MASK          0x3E000000UL /* Transmission attempts */
00183 #define XGE_TSR_LATE_COLLISION_MASK    0x01000000UL /* Transmit late collision */
00184
00185 /*
00186  * Transmit Pause Packet Register (TPPR)
00187  */
00188 #define XGE_TPPR_DATA_MASK              0x0000FFFFUL
00189
00190 /*
00191  * CAM Entry Address High Register (CEAH)
00192  */
00193 #define XGE_CLOC_DATA_MASK              0x000F0000UL
00194 #define XGE_CEAH_DATA_MASK              0x0000FFFFUL
00195
00196 /*
00197  * Transmit Excess Deferral Count (TEDC)
00198  */
00199 #define XGE_TEDC_DATA_MASK              0x0000FFFFUL
00200
00201
00202 /*
00203  * GEMAC Interrupt Registers (Status and Enable) masks. These registers are
00204  * part of the IPIF IP Interrupt registers
00205  */
00206 #define XGE_EIR_XMIT_DONE_MASK          0x00000001UL /* Xmit complete */
00207 #define XGE_EIR_RECV_DONE_MASK          0x00000002UL /* Recv complete */
00208 #define XGE_EIR_XMIT_ERROR_MASK         0x00000004UL /* Xmit error */
00209 #define XGE_EIR_RECV_ERROR_MASK         0x00000008UL /* Recv error */
00210 #define XGE_EIR_XMIT_SFIFO_EMPTY_MASK   0x00000010UL /* Xmit status fifo empty
*/
00211 #define XGE_EIR_RECV_LFIFO_EMPTY_MASK   0x00000020UL /* Recv length fifo empty
*/
00212 #define XGE_EIR_XMIT_LFIFO_FULL_MASK     0x00000040UL /* Xmit length fifo full */
00213 #define XGE_EIR_RECV_LFIFO_OVER_MASK     0x00000080UL /* Recv length fifo
00214  * overrun */
00215 #define XGE_EIR_RECV_LFIFO_UNDER_MASK    0x00000100UL /* Recv length fifo
00216  * underrun */
00217 #define XGE_EIR_XMIT_SFIFO_OVER_MASK     0x00000200UL /* Xmit status fifo
00218  * overrun */
00219 #define XGE_EIR_XMIT_SFIFO_UNDER_MASK    0x00000400UL /* Transmit status fifo
00220  * underrun */
00221 #define XGE_EIR_XMIT_LFIFO_OVER_MASK     0x00000800UL /* Transmit length fifo
00222  * overrun */
00223 #define XGE_EIR_XMIT_LFIFO_UNDER_MASK    0x00001000UL /* Transmit length fifo
00224  * underrun */
00225 #define XGE_EIR_XMIT_PAUSE_MASK          0x00002000UL /* Transmit pause pkt
00226  * received */
00227 #define XGE_EIR_RECV_DFIFO_OVER_MASK     0x00004000UL /* Receive data fifo
00228  * overrun */
00229 #define XGE_EIR_RECV_MISSED_FRAME_MASK   0x00008000UL /* Receive missed frame
00230  * error */
00231 #define XGE_EIR_RECV_COLLISION_MASK      0x00010000UL /* Receive collision
00232  * error */

```

```

00233 #define XGE_EIR_RECV_FCS_ERROR_MASK      0x00020000UL /* Receive FCS error */
00234 #define XGE_EIR_RECV_LEN_ERROR_MASK      0x00040000UL /* Receive length field
00235                                     * error */
00236 #define XGE_EIR_RECV_SHORT_ERROR_MASK     0x00080000UL /* Receive short frame
00237                                     * error */
00238 #define XGE_EIR_RECV_LONG_ERROR_MASK      0x00100000UL /* Receive long frame
00239                                     * error */
00240 #define XGE_EIR_RECV_ALIGN_ERROR_MASK     0x00200000UL /* Receive alignment
00241                                     * error */
00242 #define XGE_EIR_SLOT_LENGTH_ERROR_MASK    0x00200000UL /* Slot Length
00243                                     * error */
00244
00245 /* Offset of the MAC Statistics registers from the IPIF base address */
00246 #define XGE_STAT_REG_OFFSET              0x1100UL
00247
00248 /*
00249  * Register offsets for the MAC Statistics registers. Each register is 32 bits.
00250  */
00251
00252 /* Frames RX'd ok */
00253 #define XGE_STAT_RXOK_OFFSET              (XGE_STAT_REG_OFFSET + 0x00)
00254
00255 /* RX FCS error */
00256 #define XGE_STAT_FCSERR_OFFSET           (XGE_STAT_REG_OFFSET + 0x08)
00257
00258 /* Broadcast Frames RX'd ok */
00259 #define XGE_STAT_BFRXOK_OFFSET           (XGE_STAT_REG_OFFSET + 0x10)
00260
00261 /* Multicast Frames RX'd ok */
00262 #define XGE_STAT_MCRXOK_OFFSET           (XGE_STAT_REG_OFFSET + 0x18)
00263
00264 /* 64 byte frames RX'd ok */
00265 #define XGE_STAT_64RXOK_OFFSET           (XGE_STAT_REG_OFFSET + 0x20)
00266
00267 /* 65-127 byte frames RX'd ok */
00268 #define XGE_STAT_127RXOK_OFFSET          (XGE_STAT_REG_OFFSET + 0x28)
00269
00270 /* 128-255 byte frames RX'd ok */
00271 #define XGE_STAT_255RXOK_OFFSET          (XGE_STAT_REG_OFFSET + 0x30)
00272
00273 /* 256-511 byte frames RX'd ok */
00274 #define XGE_STAT_511RXOK_OFFSET          (XGE_STAT_REG_OFFSET + 0x38)
00275
00276 /* 512-1023 byte frames RX'd ok */
00277 #define XGE_STAT_1023RXOK_OFFSET         (XGE_STAT_REG_OFFSET + 0x40)
00278
00279 /* 1024-1518 byte frames RX'd ok */
00280 #define XGE_STAT_1518RXOK_OFFSET         (XGE_STAT_REG_OFFSET + 0x48)
00281
00282 /* Control Frames RX'd ok */
00283 #define XGE_STAT_CFRXOK_OFFSET           (XGE_STAT_REG_OFFSET + 0x50)
00284

```



```
00285 /* length/type out of range */
00286 #define XGE_STAT_LTERROR_OFFSET      (XGE_STAT_REG_OFFSET + 0x58)
00287
00288 /* VLAN Frames RX'd ok */
00289 #define XGE_STAT_VLANRXOK_OFFSET    (XGE_STAT_REG_OFFSET + 0x60)
00290
00291 /* Pause Frames RX'd ok */
00292 #define XGE_STAT_PFRXOK_OFFSET      (XGE_STAT_REG_OFFSET + 0x68)
00293
00294 /* Control Frames with unsupported opcode RX's */
00295 #define XGE_STAT_CFUNSUP_OFFSET     (XGE_STAT_REG_OFFSET + 0x70)
00296
00297 /* Oversize Frames RX'd ok */
00298 #define XGE_STAT_OFRXOK_OFFSET      (XGE_STAT_REG_OFFSET + 0x78)
00299
00300 /* Undersize Frames RX'd */
00301 #define XGE_STAT_UFRX_OFFSET        (XGE_STAT_REG_OFFSET + 0x80)
00302
00303 /* Fragment Frames RX'd */
00304 #define XGE_STAT_FRAGRX_OFFSET      (XGE_STAT_REG_OFFSET + 0x88)
00305
00306 /* RX Byte Count */
00307 #define XGE_STAT_RXBYTES_OFFSET     (XGE_STAT_REG_OFFSET + 0x90)
00308
00309 /* TX Byte Count */
00310 #define XGE_STAT_TXBYTES_OFFSET     (XGE_STAT_REG_OFFSET + 0x98)
00311
00312 /* Frames TX'd ok */
00313 #define XGE_STAT_TXOK_OFFSET        (XGE_STAT_REG_OFFSET + 0xA0)
00314
00315 /* Broadcast Frames TX'd ok */
00316 #define XGE_STAT_BFTXOK_OFFSET      (XGE_STAT_REG_OFFSET + 0xA8)
00317
00318 /* Multicast Frames TX'd ok */
00319 #define XGE_STAT_MFTXOK_OFFSET      (XGE_STAT_REG_OFFSET + 0xB0)
00320
00321 /* TX Underrun error */
00322 #define XGE_STAT_TXURUNERR_OFFSET   (XGE_STAT_REG_OFFSET + 0xB8)
00323
00324 /* Control Frames TX'd ok */
00325 #define XGE_STAT_CFTXOK_OFFSET      (XGE_STAT_REG_OFFSET + 0xC0)
00326
00327 /* 64 byte frames RX's ok*/
00328 #define XGE_STAT_64TXOK_OFFSET      (XGE_STAT_REG_OFFSET + 0xC8)
00329
00330 /* 65-127 byte frames TX'd ok*/
00331 #define XGE_STAT_127TXOK_OFFSET     (XGE_STAT_REG_OFFSET + 0xD0)
00332
00333 /* 128-255 byte frames TX'd ok*/
00334 #define XGE_STAT_255TXOK_OFFSET     (XGE_STAT_REG_OFFSET + 0xD8)
00335
00336 /* 256-511 byte frames TX'd ok*/
00337 #define XGE_STAT_511TXOK_OFFSET     (XGE_STAT_REG_OFFSET + 0xE0)
```

```
00338
00339 /* 512-1023 byte frames TX'd ok */
00340 #define XGE_STAT_1023TXOK_OFFSET    (XGE_STAT_REG_OFFSET + 0xE8)
00341
00342 /* 1024-1518 byte frames TX'd ok */
00343 #define XGE_STAT_1518TXOK_OFFSET    (XGE_STAT_REG_OFFSET + 0xF0)
00344
00345 /* VLAN Frames TX'd ok */
00346 #define XGE_STAT_VLANTXOK_OFFSET    (XGE_STAT_REG_OFFSET + 0xF8)
00347
00348 /* Pause Frames TX'd ok */
00349 #define XGE_STAT_PFTXOK_OFFSET      (XGE_STAT_REG_OFFSET + 0x100)
00350
00351 /* Oversize Frames TX'd ok */
00352 #define XGE_STAT_OFTXOK_OFFSET      (XGE_STAT_REG_OFFSET + 0x108)
00353
00354 /* Single Collision Frames */
00355 #define XGE_STAT_SCOLL_OFFSET       (XGE_STAT_REG_OFFSET + 0x110)
00356
00357 /* Multiple Collision Frames */
00358 #define XGE_STAT_MCOLL_OFFSET       (XGE_STAT_REG_OFFSET + 0x118)
00359
00360 /* Deferred Frames */
00361 #define XGE_STAT_DEFERRED_OFFSET     (XGE_STAT_REG_OFFSET + 0x120)
00362
00363 /* Late Collision Frames */
00364 #define XGE_STAT_LATECOLL_OFFSET     (XGE_STAT_REG_OFFSET + 0x128)
00365
00366 /* Frames aborted due to excess collisions */
00367 #define XGE_STAT_TXABORTED_OFFSET    (XGE_STAT_REG_OFFSET + 0x130)
00368
00369 /* Carrier sense errors */
00370 #define XGE_STAT_CARRIERERR_OFFSET  (XGE_STAT_REG_OFFSET + 0x138)
00371
00372 /* Excess Deferral error */
00373 #define XGE_STAT_EXCESSDEF_OFFSET    (XGE_STAT_REG_OFFSET + 0x140)
00374
00375 /***** Type Definitions *****/
00376
00377 /***** Macros (Inline Functions) Definitions *****/
00378
00379 /*****
00380 *
00381 * Low-level driver macros and functions. The list below provides signatures
00382 * to help the user use the macros.
00383 *
00384 * Xuint32 XGemac_mReadReg(Xuint32 BaseAddress, int RegOffset)
00385 * void XGemac_mWriteReg(Xuint32 BaseAddress, int RegOffset, Xuint32 Mask)
00386 *
00387 * void XGemac_mSetControlReg(Xuint32 BaseAddress, Xuint32 Mask)
00388 * void XGemac_mSetMacAddress(Xuint32 BaseAddress, Xuint8 *AddressPtr)
00389 *
```

```

00390 * void XGemac_mEnable(Xuint32 BaseAddress)
00391 * void XGemac_mDisable(Xuint32 BaseAddress)
00392 *
00393 * Xboolean XGemac_mIsTxDone(Xuint32 BaseAddress)
00394 * Xboolean XGemac_mIsRxEmpty(Xuint32 BaseAddress)
00395 *
00396 * void XGemac_SendFrame(Xuint32 BaseAddress, Xuint8 *FramePtr, int Size)
00397 * int XGemac_RecvFrame(Xuint32 BaseAddress, Xuint8 *FramePtr)
00398 *
00399 *****/
00400
00401 /***/
00402 /**
00403 *
00404 * Read the given register.
00405 *
00406 * @param BaseAddress is the base address of the device
00407 * @param RegOffset is the register offset to be read
00408 *
00409 * @return The 32-bit value of the register
00410 *
00411 * @note None.
00412 *
00413 *****/
00414 #define XGemac_mReadReg(BaseAddress, RegOffset) \
00415     XIo_In32((BaseAddress) + (RegOffset))
00416
00417
00418 *****/
00419 /**
00420 *
00421 * Write the given register.
00422 *
00423 * @param BaseAddress is the base address of the device
00424 * @param RegOffset is the register offset to be written
00425 * @param Data is the 32-bit value to write to the register
00426 *
00427 * @return None.
00428 *
00429 * @note None.
00430 *
00431 *****/
00432 #define XGemac_mWriteReg(BaseAddress, RegOffset, Data) \
00433     XIo_Out32((BaseAddress) + (RegOffset), (Data))
00434
00435
00436 *****/
00437 /**
00438 *
00439 * Set the contents of the control register. Use the XGE_ECR_* constants
00440 * defined above to create the bit-mask to be written to the register.
00441 *

```

```

00442 * @param      BaseAddress is the base address of the device
00443 * @param      Mask is the 16-bit value to write to the control register
00444 *
00445 * @return     None.
00446 *
00447 * @note      None.
00448 *
00449 *****/
00450 #define XGEmac_mSetControlReg(BaseAddress, Mask) \
00451         XIo_Out32((BaseAddress) + XGE_ECR_OFFSET, (Mask))
00452
00453
00454 *****/
00455 /**
00456 *
00457 * Set the station address of the GEMAC device.
00458 *
00459 * @param      BaseAddress is the base address of the device
00460 * @param      AddressPtr is a pointer to a 6-byte MAC address
00461 *
00462 * @return     None.
00463 *
00464 * @note      None.
00465 *
00466 *****/
00467 #define XGEmac_mSetMacAddress(BaseAddress, AddressPtr) \
00468 { \
00469     Xuint32 MacAddr; \
00470 \
00471     MacAddr = ((AddressPtr)[0] << 8) | (AddressPtr)[1]; \
00472     XIo_Out32((BaseAddress) + XGE_SAH_OFFSET, MacAddr); \
00473 \
00474     MacAddr = ((AddressPtr)[2] << 24) | ((AddressPtr)[3] << 16) | \
00475             ((AddressPtr)[4] << 8) | (AddressPtr)[5]; \
00476 \
00477     XIo_Out32((BaseAddress) + XGE_SAL_OFFSET, MacAddr); \
00478 }
00479
00480
00481 *****/
00482 /**
00483 *
00484 * Enable the transmitter and receiver. Preserve the contents of the control
00485 * register.
00486 *
00487 * @param      BaseAddress is the base address of the device
00488 *
00489 * @return     None.
00490 *
00491 * @note      None.
00492 *
00493 *****/

```

```

00494 #define XGEmac_mEnable(BaseAddress) \
00495 { \
00496     Xuint32 Control; \
00497     Control = XIo_In32((BaseAddress) + XGE_ECR_OFFSET); \
00498     Control &= ~(XGE_ECR_XMIT_RESET_MASK | XGE_ECR_RECV_RESET_MASK); \
00499     Control |= (XGE_ECR_XMIT_ENABLE_MASK | XGE_ECR_RECV_ENABLE_MASK); \
00500     XIo_Out32((BaseAddress) + XGE_ECR_OFFSET, Control); \
00501 }
00502
00503
00504 /*****
00505 /**
00506 *
00507 * Disable the transmitter and receiver. Preserve the contents of the control
00508 * register.
00509 *
00510 * @param    BaseAddress is the base address of the device
00511 *
00512 * @return   None.
00513 *
00514 * @note    None.
00515 *
00516 *****/
00517 #define XGEmac_mDisable(BaseAddress) \
00518     XIo_Out32((BaseAddress) + XGE_ECR_OFFSET, \
00519             XIo_In32((BaseAddress) + XGE_ECR_OFFSET) & \
00520             ~(XGE_ECR_XMIT_ENABLE_MASK | XGE_ECR_RECV_ENABLE_MASK))
00521
00522
00523 /*****
00524 /**
00525 *
00526 * Check to see if the transmission is complete.
00527 *
00528 * @param    BaseAddress is the base address of the device
00529 *
00530 * @return   XTRUE if it is done, or XFALSE if it is not.
00531 *
00532 * @note    None.
00533 *
00534 *****/
00535 #define XGEmac_mIsTxDone(BaseAddress) \
00536     (XIo_In32((BaseAddress) + XGE_ISR_OFFSET) &
XGE_EIR_XMIT_DONE_MASK)
00537
00538
00539 /*****
00540 /**
00541 *
00542 * Check to see if the receive FIFO is empty.
00543 *
00544 * @param    BaseAddress is the base address of the device

```

```

00545 *
00546 * @return XTRUE if it is empty, or XFALSE if it is not.
00547 *
00548 * @note None.
00549 *
00550 *****/
00551 #define XGemac_mIsRxEmpty(BaseAddress) \
00552     (!(XIo_In32((BaseAddress) + XGE_ISR_OFFSET) &
XGE_EIR_RECV_DONE_MASK))
00553
00554
00555 *****/
00556 /**
00557 *
00558 * Reset MII compliant PHY
00559 *
00560 * @param BaseAddress is the base address of the device
00561 *
00562 * @return None.
00563 *
00564 * @note None.
00565 *
00566 *****/
00567 #define XGemac_mPhyReset(BaseAddress) \
00568 { \
00569     Xuint32 Control; \
00570     Control = XIo_In32((BaseAddress) + XGE_ECR_OFFSET); \
00571     Control &= ~XGE_ECR_PHY_ENABLE_MASK; \
00572     XIo_Out32((BaseAddress) + XGE_ECR_OFFSET, Control); \
00573     Control |= XGE_ECR_PHY_ENABLE_MASK; \
00574     XIo_Out32((BaseAddress) + XGE_ECR_OFFSET, Control); \
00575 }
00576
00577
00578 *****/ Function Prototypes *****/
00579
00580 void XGemac_SendFrame(Xuint32 BaseAddress, Xuint8 *FramePtr, int Size);
00581 int XGemac_RecvFrame(Xuint32 BaseAddress, Xuint8 *FramePtr);
00582
00583
00584 #endif /* end of protection macro */

```

gemac/v1_00_c/src/xgemac_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xgemac.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Removed tabs characters in file, added auto negotiate bit to ECR constants.

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XGemac_mReadReg(BaseAddress, RegOffset)  
#define XGemac_mWriteReg(BaseAddress, RegOffset, Data)  
#define XGemac_mSetControlReg(BaseAddress, Mask)  
#define XGemac_mSetMacAddress(BaseAddress, AddressPtr)  
#define XGemac_mEnable(BaseAddress)
```

```
#define XGemac_mDisable(BaseAddress)
#define XGemac_mIsTxDone(BaseAddress)
#define XGemac_mIsRxEmpty(BaseAddress)
#define XGemac_mPhyReset(BaseAddress)
```

Define Documentation

```
#define XGemac_mDisable( BaseAddress )
```

Disable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XGemac_mEnable( BaseAddress )
```

Enable the transmitter and receiver. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XGemac_mIsRxEmpty( BaseAddress )
```


Check to see if the receive FIFO is empty.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is empty, or XFALSE if it is not.

Note:

None.

```
#define XGmac_mIsTxDone( BaseAddress )
```

Check to see if the transmission is complete.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if it is done, or XFALSE if it is not.

Note:

None.

```
#define XGmac_mPhyReset( BaseAddress )
```

Reset MII compliant PHY

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XGmac_mReadReg( BaseAddress,  
                        RegOffset )
```

Read the given register.

Parameters:

BaseAddress is the base address of the device
RegOffset is the register offset to be read

Returns:

The 32-bit value of the register

Note:

None.

```
#define XGemac_mSetControlReg( BaseAddress,  
                               Mask          )
```

Set the contents of the control register. Use the XGE_ECR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device
Mask is the 16-bit value to write to the control register

Returns:

None.

Note:

None.

```
#define XGemac_mSetMacAddress( BaseAddress,  
                               AddressPtr  )
```

Set the station address of the GEMAC device.

Parameters:

BaseAddress is the base address of the device

AddressPtr is a pointer to a 6-byte MAC address

Returns:

None.

Note:

None.

```
#define XGemac_mWriteReg( BaseAddress,  
                          RegOffset,  
                          Data          )
```

Write the given register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the register offset to be written

Data is the 32-bit value to write to the register

Returns:

None.

Note:

None.

gemac/v1_00_c/src/xgemac_selftest.c File Reference

Detailed Description

Self-test and diagnostic functions of the **XGemac** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Revision update

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

XStatus XGemac_SelfTest (**XGemac** *InstancePtr)

Function Documentation

XStatus XGemac_SelfTest(**XGemac** * *InstancePtr*)

Performs a self-test on the Ethernet device. The test includes:

- Run self-test on DMA channel, FIFO, and IPIF components
- Reset the Ethernet device, check its registers for proper reset values, and run an internal loopback test on the device. The internal loopback uses the device in polled mode.

This self-test is destructive. On successful completion, the device is reset and returned to its default configuration. The caller is responsible for re-configuring the device after the self-test is run, and starting it when ready to send and receive frames.

It should be noted that data caching must be disabled when this function is called because the DMA self-test uses two local buffers (on the stack) for the transfer test.

Parameters:

InstancePtr is a pointer to the [XGEmac](#) instance to be worked on.

Returns:

XST_SUCCESS	Self-test was successful
XST_PFIPO_BAD_REG_VALUE	FIFO failed register self-test
XST_DMA_TRANSFER_ERROR	DMA failed data transfer self-test
XST_DMA_RESET_REGISTER_ERROR	DMA control register value was incorrect after a reset
XST_REGISTER_ERROR	Ethernet failed register reset test
XST_LOOPBACK_ERROR	Internal loopback failed
XST_IPIF_REG_WIDTH_ERROR	An invalid register width was passed into the function
XST_IPIF_RESET_REGISTER_ERROR	The value of a register at reset was invalid
XST_IPIF_DEVICE_STATUS_ERROR	A write to the device status register did not read back correctly
XST_IPIF_DEVICE_ACK_ERROR	A bit in the device status register did not reset when acked
XST_IPIF_DEVICE_ENABLE_ERROR	The device interrupt enable register was not updated correctly by the hardware when other registers were written to
XST_IPIF_IP_STATUS_ERROR	A write to the IP interrupt status register did not read back correctly
XST_IPIF_IP_ACK_ERROR	One or more bits in the IP status register did not reset when acked
XST_IPIF_IP_ENABLE_ERROR	The IP interrupt enable register was not updated correctly when other registers were written to

Note:

This function makes use of options-related functions, and the [XGEmac_PollSend\(\)](#) and [XGEmac_PollRecv\(\)](#) functions.

Because this test uses the PollSend function for its loopback testing, there is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread

to monitor the self-test thread.

Generated on 29 May 2003 for Xilinx Device Drivers

gemac/v1_00_c/src/xgemac_polled.c File Reference

Detailed Description

Contains functions used when the driver is in polled mode. Use the [XGemac_SetOptions\(\)](#) function to put the driver into polled mode.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Revision update

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

[XStatus XGemac_PollSend](#) ([XGemac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) ByteCount)

[XStatus XGemac_PollRecv](#) ([XGemac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) *ByteCountPtr)

Function Documentation

```
XStatus XGemac_PollRecv( XGemac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 * ByteCountPtr  
                        )
```

Receive an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver receives the frame directly from the MAC's packet FIFO. This is a non-blocking receive, in that if there is no frame ready to be received at the device, the function returns with an error. The MAC's error status is not checked, so statistics are not updated for polled receive. The buffer into which the frame will be received must be word-aligned.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_NO_DATA if there is no frame to be received from the FIFO
- XST_BUFFER_TOO_SMALL if the buffer to receive the frame is too small for the frame waiting in the FIFO.

Note:

Input buffer must be big enough to hold the largest Ethernet frame. Buffer must also be 32-bit aligned.

```
XStatus XGemac_PollSend( XGemac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 ByteCount  
                        )
```


Send an Ethernet frame in polled mode. The device/driver must be in polled mode before calling this function. The driver writes the frame directly to the MAC's packet FIFO, then enters a loop checking the device status for completion or error. Statistics are updated if an error occurs. The buffer to be sent must be word-aligned.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was sent successfully
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_POLLED if the device is not in polled mode
- XST_FIFO_NO_ROOM if there is no room in the GEMAC's length FIFO for this frame
- XST_FIFO_ERROR if the FIFO was overrun or underrun. This error is critical and requires the caller to reset the device.
- XST_EMAC_COLLISION if the send failed due to excess deferral or late collision

Note:

There is the possibility that this function will not return if the hardware is broken (i.e., it never sets the status bit indicating that transmission is done). If this is of concern to the user, the user should provide protection from this problem - perhaps by using a different timer thread to monitor the PollSend thread. On a 1 Gbit (1000Mbps) MAC, it takes about 12.1 usecs to transmit a maximum size Ethernet frame.

gemac/v1_00_c/src/xgemac_intr_dma.c

File Reference

Detailed Description

Contains functions used in interrupt mode when configured with scatter-gather DMA.

The interrupt handler, [XGemac_IntrHandlerDma\(\)](#), must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00c	rmm	05/28/03	New capability for driver

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xbuf_descriptor.h"  
#include "xdma_channel.h"  
#include "xipif_v1_23_b.h"
```

Functions

[XStatus XGemac_SgSend](#) ([XGemac](#) *InstancePtr, [XBufDescriptor](#) *BdPtr, int Delay)

[XStatus XGemac_SgRecv](#) ([XGemac](#) *InstancePtr, [XBufDescriptor](#) *BdPtr)

```

void XGmac_IntrHandlerDma (void *InstancePtr)
XStatus XGmac_SetPktThreshold (XGmac *InstancePtr, Xuint32 Direction, Xuint8 Threshold)
XStatus XGmac_GetPktThreshold (XGmac *InstancePtr, Xuint32 Direction, Xuint8 *ThreshPtr)
XStatus XGmac_SetPktWaitBound (XGmac *InstancePtr, Xuint32 Direction, Xuint32
    TimerValue)
XStatus XGmac_GetPktWaitBound (XGmac *InstancePtr, Xuint32 Direction, Xuint32 *WaitPtr)
XStatus XGmac_SetSgRecvSpace (XGmac *InstancePtr, Xuint32 *MemoryPtr, Xuint32
    ByteCount)
XStatus XGmac_SetSgSendSpace (XGmac *InstancePtr, Xuint32 *MemoryPtr, Xuint32
    ByteCount)
void XGmac_SetSgRecvHandler (XGmac *InstancePtr, void *CallBackRef,
    XGmac_SgHandler FuncPtr)
void XGmac_SetSgSendHandler (XGmac *InstancePtr, void *CallBackRef,
    XGmac_SgHandler FuncPtr)

```

Function Documentation

```

XStatus XGmac_GetPktThreshold( XGmac * InstancePtr,
                                Xuint32 Direction,
                                Xuint8 * ThreshPtr
                                )

```

Get the value of the packet count threshold for this driver/device. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

ThreshPtr is a pointer to the byte into which the current value of the packet threshold register will be copied. An output parameter. A value of 0 indicates the use of packet threshold by the hardware is disabled.

Returns:

- XST_SUCCESS if the packet threshold was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts

would also catch this error.

Note:

None.

```
XStatus XGemac_GetPktWaitBound( XGemac * InstancePtr,  
                                Xuint32   Direction,  
                                Xuint32 * WaitPtr  
                                )
```

Get the packet wait bound timer for this driver/device. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

WaitPtr is a pointer to the byte into which the current value of the packet wait bound register will be copied. An output parameter. Units are in milliseconds in the range 0 - 1023. A value of 0 indicates the packet wait bound timer is disabled.

Returns:

- XST_SUCCESS if the packet wait bound was retrieved successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void XGemac_IntrHandlerDma( void * InstancePtr)
```

The interrupt handler for the Ethernet driver when configured with scatter-gather DMA.

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, Send Packet FIFO, Recv DMA channel, or Send DMA channel. The packet FIFOs only interrupt during "deadlock" conditions.

Parameters:

InstancePtr is a pointer to the **XGemac** instance that just interrupted.

Returns:

None.

Note:

None.

```
XStatus XGemac_SetPktThreshold( XGemac * InstancePtr,  
                                Xuint32  Direction,  
                                Xuint8   Threshold  
                                )
```

Set the packet count threshold for this device. The device must be stopped before setting the threshold. The packet count threshold is used for interrupt coalescing, which reduces the frequency of interrupts from the device to the processor. In this case, the scatter-gather DMA engine only interrupts when the packet count threshold is reached, instead of interrupting for each packet. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

Threshold is the value of the packet threshold count used during interrupt coalescing. A value of 0 disables the use of packet threshold by the hardware.

Returns:

- XST_SUCCESS if the threshold was successfully set
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_DMA_SG_COUNT_EXCEEDED if the threshold must be equal to or less than the number of descriptors in the list
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts

would also catch this error.

Note:

None.

```
XStatus XGemac_SetPktWaitBound( XGemac * InstancePtr,  
                                Xuint32   Direction,  
                                Xuint32   TimerValue  
                                )
```

Set the packet wait bound timer for this driver/device. The device must be stopped before setting the timer value. The packet wait bound is used during interrupt coalescing to trigger an interrupt when not enough packets have been received to reach the packet count threshold. A packet is a generic term used by the scatter-gather DMA engine, and is equivalent to an Ethernet frame in our case. The timer is in milliseconds.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Direction indicates the channel, send or receive, from which the threshold register is read.

TimerValue is the value of the packet wait bound used during interrupt coalescing. It is in milliseconds in the range 0 - 1023. A value of 0 disables the packet wait bound timer.

Returns:

- XST_SUCCESS if the packet wait bound was set successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DEVICE_IS_STARTED if the device has not been stopped
- XST_INVALID_PARAM if the Direction parameter is invalid. Turning on asserts would also catch this error.

Note:

None.

```
void XGemac_SetSgRecvHandler( XGemac * InstancePtr,  
                              void *    CallbackRef,  
                              XGemac_SgHandler FuncPtr  
                              )
```

Set the callback function for handling received frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame received. The head of a descriptor list is passed in along with the number of descriptors in the list. Before leaving the callback, the upper layer software should attach a new buffer to each descriptor in the list.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
XStatus XGmac_SetSgRecvSpace( XGmac * InstancePtr,  
                               Xuint32 * MemoryPtr,  
                               Xuint32   ByteCount  
                               )
```

Give the driver the memory space to be used for the scatter-gather DMA receive descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xgemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be aligned on an 8 byte boundary. An assert will occur if asserts are turned on and the memory is not aligned correctly.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

MemoryPtr is a pointer to the 64-bit aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XGemac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
void XGemac_SetSgSendHandler( XGemac *           InstancePtr,
                             void *           CallbackRef,
                             XGemac_SgHandler FuncPtr
                             )
```

Set the callback function for handling confirmation of transmitted frames in scatter-gather DMA mode. The upper layer software should call this function during initialization. The callback is called once per frame sent. The head of a descriptor list is passed in along with the number of descriptors in the list. The callback is responsible for freeing buffers attached to these descriptors.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.


```
XStatus XGemac_SetSgSendSpace( XGemac * InstancePtr,  
                                Xuint32 * MemoryPtr,  
                                Xuint32   ByteCount  
                                )
```

Give the driver the memory space to be used for the scatter-gather DMA transmit descriptor list. This function should only be called once, during initialization of the Ethernet driver. The memory space must be big enough to hold some number of descriptors, depending on the needs of the system. The [xgemac.h](#) file defines minimum and default numbers of descriptors which can be used to allocate this memory space.

The memory space must be aligned on an 8 byte boundary. An assert will occur if asserts are turned on and the memory is not aligned correctly.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

MemoryPtr is a pointer to the 64-bit aligned memory.

ByteCount is the length, in bytes, of the memory space.

Returns:

- XST_SUCCESS if the space was initialized successfully
- XST_NOT_SGDMMA if the MAC is not configured for scatter-gather DMA
- XST_DMA_SG_LIST_EXISTS if this list space has already been created

Note:

If the device is configured for scatter-gather DMA, this function must be called AFTER the **XGemac_Initialize()** function because the DMA channel components must be initialized before the memory space is set.

```
XStatus XGemac_SgRecv( XGemac * InstancePtr,  
                       XBufDescriptor * BdPtr  
                       )
```

Add a descriptor, with an attached empty buffer, into the receive descriptor list. The buffer attached to the descriptor must be aligned on an 8 byte boundary. This is used by the upper layer software during initialization when first setting up the receive descriptors, and also during reception of frames to replace filled buffers with empty buffers. This function can be called when the device is started or stopped. Note that it does start the scatter-gather DMA engine. Although this is not necessary during initialization, it is not a problem during initialization because the MAC receiver is not yet started.

The buffer attached to the descriptor must be 64-bit aligned on both the front end and the back end.

Notification of received frames are done asynchronously through the receive callback function.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

BdPtr is a pointer to the buffer descriptor that will be added to the descriptor list.

Returns:

- XST_SUCCESS if a descriptor was successfully returned to the driver
- XST_NOT_SG_DMA if the device is not in scatter-gather DMA mode
- XST_DMA_SG_LIST_FULL if the receive descriptor list is full
- XST_DMA_SG_BD_LOCKED if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point.
- XST_DMA_SG_NOTHING_TO_COMMIT if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit.

```
XStatus XGemac_SgSend( XGemac * InstancePtr,  
                      XBufDescriptor * BdPtr,  
                      int Delay  
                      )
```

Send an Ethernet frame using scatter-gather DMA. The caller attaches the frame to one or more buffer descriptors, then calls this function once for each descriptor. The caller is responsible for allocating and setting up the descriptor. An entire Ethernet frame may or may not be contained within one descriptor. This function simply inserts the descriptor into the scatter-gather engine's transmit list. The caller is responsible for providing mutual exclusion to guarantee that a frame is contiguous in the transmit list. The buffer attached to the descriptor must be aligned on an 8 byte boundary.

The driver updates the descriptor with the device control register before being inserted into the transmit list. If this is the last descriptor in the frame, the inserts are committed, which means the descriptors for this frame are now available for transmission.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field. It is also assumed that upper layer software does not append FCS at the end of the frame.

The buffer attached to the descriptor must be 64-bit aligned on the front end.

This call is non-blocking. Notification of error or successful transmission is done asynchronously through the send or error callback function.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BdPtr is the address of a descriptor to be inserted into the transmit ring.

Delay indicates whether to start the scatter-gather DMA channel immediately, or whether to wait. This allows the user to build up a list of more than one descriptor before starting the transmission of the packets, which allows the application to keep up with DMA and have a constant stream of frames being transmitted. Use `XEM_SGDMA_NODELAY` or `XEM_SGDMA_DELAY`, defined in **xgemac.h**, as the value of this argument. If the user chooses to delay and build a list, the user must call this function with the `XEM_SGDMA_NODELAY` option or call **XGmac_Start()** to kick off the transmissions.

Returns:

- `XST_SUCCESS` if the buffer was successful sent
- `XST_DEVICE_IS_STOPPED` if the Ethernet MAC has not been started yet
- `XST_NOT_SGDMA` if the device is not in scatter-gather DMA mode
- `XST_DMA_SG_LIST_FULL` if the descriptor list for the DMA channel is full
- `XST_DMA_SG_BD_LOCKED` if the DMA channel cannot insert the descriptor into the list because a locked descriptor exists at the insert point
- `XST_DMA_SG_NOTHING_TO_COMMIT` if even after inserting a descriptor into the list, the DMA channel believes there are no new descriptors to commit. If this is ever encountered, there is likely a thread mutual exclusion problem on transmit.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

gemac/v1_00_c/src/xgemac_intr_fifo.c

File Reference

Detailed Description

Contains functions related to interrupt mode using direct FIFO communication.

The interrupt handler, [XGmac_IntrHandlerFifo\(\)](#), must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00a	ecm	02/05/03	includes support for simple DMA
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Revision update

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

[XStatus XGmac_FifoSend](#) ([XGmac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) ByteCount)

[XStatus XGmac_FifoRecv](#) ([XGmac](#) *InstancePtr, [Xuint8](#) *BufPtr, [Xuint32](#) *ByteCountPtr)

```
void XGemac_IntrHandlerFifo (void *InstancePtr)
void XGemac_SetFifoRecvHandler (XGemac *InstancePtr, void *CallBackRef,
XGemac_FifoHandler FuncPtr)
void XGemac_SetFifoSendHandler (XGemac *InstancePtr, void *CallBackRef,
XGemac_FifoHandler FuncPtr)
```

Function Documentation

```
XStatus XGemac_FifoRecv( XGemac * InstancePtr,
                        Xuint8 * BufPtr,
                        Xuint32 * ByteCountPtr
                        )
```

Receive an Ethernet frame into the buffer passed as an argument. This function is called in response to the callback function for received frames being called by the driver. The callback function is set up using `SetFifoRecvHandler`, and is invoked when the driver receives an interrupt indicating a received frame. The driver expects the upper layer software to call this function, `FifoRecv`, to receive the frame. The buffer supplied should be large enough to hold a maximum-size Ethernet frame.

The buffer into which the frame will be received must be word-aligned.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from the Emac to memory. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xgemac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer into which the received Ethernet frame will be copied.

ByteCountPtr is both an input and an output parameter. It is a pointer to a 32-bit word that contains the size of the buffer on entry into the function and the size the received frame on return from the function.

Returns:

- `XST_SUCCESS` if the frame was sent successfully
- `XST_DEVICE_IS_STOPPED` if the device has not yet been started
- `XST_NOT_INTERRUPT` if the device is not in interrupt mode
- `XST_NO_DATA` if there is no frame to be received from the FIFO
- `XST_BUFFER_TOO_SMALL` if the buffer to receive the frame is too small for the

frame waiting in the FIFO.

- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

The input buffer must be big enough to hold the largest Ethernet frame.

```
XStatus XGmac_FifoSend( XGmac * InstancePtr,  
                        Xuint8 * BufPtr,  
                        Xuint32 ByteCount  
                        )
```

Send an Ethernet frame using packet FIFO with interrupts. The caller provides a contiguous-memory buffer and its length. The buffer must be word-aligned. The callback function set by using SetFifoSendHandler is invoked when the transmission is complete.

It is assumed that the upper layer software supplies a correctly formatted Ethernet frame, including the destination and source addresses, the type/length field, and the data field.

If the device is configured with DMA, simple DMA will be used to transfer the buffer from memory to the Emac. This means that this buffer should not be cached. See the comment section "Simple DMA" in [xgmac.h](#) for more information.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

BufPtr is a pointer to a word-aligned buffer containing the Ethernet frame to be sent.

ByteCount is the size of the Ethernet frame.

Returns:

- XST_SUCCESS if the frame was successfully sent. An interrupt is generated when the GEMAC transmits the frame and the driver calls the callback set with [XGmac_SetFifoSendHandler\(\)](#)
- XST_DEVICE_IS_STOPPED if the device has not yet been started
- XST_NOT_INTERRUPT if the device is not in interrupt mode
- XST_FIFO_NO_ROOM if there is no room in the FIFO for this frame
- XST_DEVICE_BUSY if configured for simple DMA and the DMA engine is busy
- XST_DMA_ERROR if an error occurred during the DMA transfer (simple DMA). The user should treat this as a fatal error that requires a reset of the EMAC device.

Note:

This function is not thread-safe. The user must provide mutually exclusive access to this function if there are to be multiple threads that can call it.

```
void XGemac_IntrHandlerFifo( void * InstancePtr)
```

The interrupt handler for the Ethernet driver when configured for direct FIFO communication (as opposed to DMA).

Get the interrupt status from the IpIf to determine the source of the interrupt. The source can be: MAC, Recv Packet FIFO, or Send Packet FIFO. The packet FIFOs only interrupt during "deadlock" conditions. All other FIFO-related interrupts are generated by the MAC.

Parameters:

InstancePtr is a pointer to the **XGemac** instance that just interrupted.

Returns:

None.

Note:

None.

```
void XGemac_SetFifoRecvHandler( XGemac * InstancePtr,  
void * CallbackRef,  
XGemac_FifoHandler FuncPtr  
)
```

Set the callback function for handling confirmation of transmitted frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called by the driver once per frame sent. The callback is responsible for freeing the transmitted buffer if necessary.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

```
void XGemac_SetFifoSendHandler( XGemac *           InstancePtr,  
                               void *           CallbackRef,  
                               XGemac_FifoHandler FuncPtr  
                               )
```

Set the callback function for handling received frames when configured for direct memory-mapped I/O using FIFOs. The upper layer software should call this function during initialization. The callback is called once per frame received. During the callback, the upper layer software should call `FifoRecv` to retrieve the received frame.

The callback is invoked by the driver within interrupt context, so it needs to do its job quickly. Sending the received frame up the protocol stack should be done at task-level. If there are other potentially slow operations within the callback, these too should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

gemac/v1_00_c/src/xgemac_intr.c File Reference

Detailed Description

This file contains general interrupt-related functions of the **XGemac** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Revision update

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

```
void XGemac_SetErrorHandler (XGemac *InstancePtr, void *CallBackRef, XGemac_ErrorHandler  
FuncPtr)
```

Function Documentation

```
void XGemac_SetErrorHandler( XGemac *           InstancePtr,  
                             void *           CallbackRef,  
                             XGemac_ErrorHandler FuncPtr  
                             )
```

Set the callback function for handling asynchronous errors. The upper layer software should call this function during initialization.

The error callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

The Xilinx errors that must be handled by the callback are:

- XST_DMA_ERROR indicates an unrecoverable DMA error occurred. This is typically a bus error or bus timeout. The handler must reset and re-configure the device.
- XST_FIFO_ERROR indicates an unrecoverable FIFO error occurred. This is a deadlock condition in the packet FIFO. The handler must reset and re-configure the device.
- XST_RESET_ERROR indicates an unrecoverable MAC error occurred, usually an overrun or underrun. The handler must reset and re-configure the device.
- XST_DMA_SG_NO_LIST indicates an attempt was made to access a scatter-gather DMA list that has not yet been created.
- XST_DMA_SG_LIST_EMPTY indicates the driver tried to get a descriptor from the receive descriptor list, but the list was empty.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

CallbackRef is a reference pointer to be passed back to the adapter in the callback. This helps the adapter correlate the callback to a particular driver.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

None.

gemac/v1_00_c/src/xgemac_options.c File Reference

Detailed Description

Functions in this file handle configuration of the **XGemac** driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Changed interframe gap API, process auto negotiate option

```
#include "xbasic_types.h"  
#include "xgemac_i.h"  
#include "xio.h"
```

Data Structures

```
struct OptionMap
```

Functions

```
XStatus XGemac_SetOptions (XGemac *InstancePtr, Xuint32 OptionsFlag)
```

```
Xuint32 XGemac_GetOptions (XGemac *InstancePtr)
```

```
XStatus XGemac_SetInterframeGap (XGemac *InstancePtr, Xuint8 Ifg)
```

```
void XGemac_GetInterframeGap (XGemac *InstancePtr, Xuint8 *IfgPtr)
```

Function Documentation

```
void XGmac_GetInterframeGap( XGmac * InstancePtr,  
                             Xuint8 * IfgPtr  
                             )
```

Get the interframe gap. See the description of interframe gap above in [XGmac_SetInterframeGap\(\)](#).

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

IfgPtr is a pointer to an 8-bit buffer into which the interframe gap value will be copied.

Returns:

None. The values of the interframe gap parts are copied into the output parameters.

```
Xuint32 XGmac_GetOptions( XGmac * InstancePtr)
```

Get Ethernet driver/device options. The 32-bit value returned is a bit-mask representing the options. A one (1) in the bit-mask means the option is on, and a zero (0) means the option is off.

Parameters:

InstancePtr is a pointer to the [XGmac](#) instance to be worked on.

Returns:

The 32-bit value of the Ethernet options. The value is a bit-mask representing all options that are currently enabled. See [xgemac.h](#) for a description of the available options.

Note:

None.

```
XStatus XGmac_SetInterframeGap( XGmac * InstancePtr,  
                                Xuint8 Ifg  
                                )
```

Set the Interframe Gap (IFG), which is the time the MAC delays between transmitting frames. There are two parts required. The total interframe gap is the total of the two parts. The values provided for the Part1 and Part2 parameters are multiplied by 4 to obtain the bit-time interval. The first part should be the first 2/3 of the total interframe gap. The MAC will reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part1. Part1 may be shorter than 2/3 the total and can be as small as zero. The second part should be the last 1/3 of the total interframe gap, but can be as large as the total interframe gap. The MAC will not reset the interframe gap timer if carrier sense becomes true during the period defined by interframe gap Part2.

The device must be stopped before setting the interframe gap.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

Ifg is the interframe gap, MSB is 8-bit time.

Returns:

- XST_SUCCESS if the interframe gap was set successfully
- XST_DEVICE_IS_STARTED if the device has not been stopped

Note:

None.

```
XStatus XGemac_SetOptions( XGemac * InstancePtr,  
                          Xuint32 OptionsFlag  
                          )
```

Set Ethernet driver/device options. The device must be stopped before calling this function. The options are contained within a bit-mask with each bit representing an option (i.e., you can OR the options together). A one (1) in the bit-mask turns an option on, and a zero (0) turns the option off.

Parameters:

InstancePtr is a pointer to the **XGemac** instance to be worked on.

OptionsFlag is a bit-mask representing the Ethernet options to turn on or off. See [xgemac.h](#) for a description of the available options.

Returns:

- XST_SUCCESS if the options were set successfully
- XST_DEVICE_IS_STARTED if the device has not yet been stopped

Note:

This function is not thread-safe and makes use of internal resources that are shared between the Start, Stop, and SetOptions functions, so if one task might be setting device options while another is trying to start the device, protection of this shared data (typically using a semaphore) is required.

gemac/v1_00_c/src/xgemac_stats.c File Reference

Detailed Description

Contains functions to get and clear the **XGemac** driver statistics.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/13/03	First release
1.00b	ecm	03/25/03	Revision update
1.00c	rmm	05/28/03	Revision update

```
#include "xbasic_types.h"  
#include "xgemac_i.h"
```

Functions

```
void XGemac_GetStats (XGemac *InstancePtr, XGemac_Stats *StatsPtr)  
void XGemac_ClearStats (XGemac *InstancePtr)
```

Function Documentation

```
void XGemac_ClearStats( XGemac * InstancePtr)
```

Clear the XGmacStats structure for this driver.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

Returns:

None.

Note:

None.

```
void XGmac_GetStats( XGmac *      InstancePtr,  
                    XGmac_Stats * StatsPtr  
                    )
```

Get a copy of the XGmacStats structure, which contains the current statistics for this driver. The statistics are only cleared at initialization or on demand using the **XGmac_ClearStats()** function.

The DmaErrors and FifoErrors counts indicate that the device has been or needs to be reset. Reset of the device is the responsibility of the upper layer software.

Parameters:

InstancePtr is a pointer to the **XGmac** instance to be worked on.

StatsPtr is an output parameter, and is a pointer to a stats buffer into which the current statistics will be copied.

Returns:

None.

Note:

None.

gpio/v1_00_a/src/xgpio_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between the files of the driver. It is intended for internal use only.

NOTES:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	03/13/02	First release

```
#include "xgpio.h"
```

[Go to the source code of this file.](#)

Variables

XGpio_Config [XGpio_ConfigTable](#) []

Variable Documentation

XGpio_Config XGpio_ConfigTable[]()

This table contains configuration information for each GPIO device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

gpio/v1_00_a/src/xgpio.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xgpio.h,v 1.4 2002/05/02 20:44:36 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 * @file gpio/v1_00_a/src/xgpio.h
00025 *
00026 * This file contains the software API definition of the Xilinx General Purpose
00027 * I/O (XGpio) component.
00028 *
00029 * The Xilinx GPIO controller is a soft IP core designed for Xilinx FPGAs and
00030 * contains the following general features:
00031 *   - Support for 8, 16, or 32 I/O discrettes
00032 *   - Each of the discrettes can be configured for input or output.
00033 *
00034 * @note
00035 *
00036 * This API utilizes 32 bit I/O to the GPIO registers. With 16 and 8 bit GPIO
00037 * components, the unused bits from registers are read as zero and written as
00038 * don't cares.
00039 *
00040 * <pre>
00041 * MODIFICATION HISTORY:
00042 *
```

```

00043 * Ver   Who   Date     Changes
00044 * -----
00045 * 1.00a rmm  03/13/02 First release
00046 * </pre>
00047 *****/
00048
00049 #ifndef XGPIO_H /* prevent circular inclusions */
00050 #define XGPIO_H /* by using protection macros */
00051
00052 /***** Include Files *****/
00053
00054 #include "xbasic_types.h"
00055 #include "xstatus.h"
00056 #include "xgpio_1.h"
00057
00058 /***** Constant Definitions *****/
00059
00060 /***** Type Definitions *****/
00061
00062 /*
00063  * This typedef contains configuration information for the device.
00064  */
00065 typedef struct
00066 {
00067     Xuint16 DeviceId;      /* Unique ID of device */
00068     Xuint32 BaseAddress;   /* Device base address */
00069 } XGpio_Config;
00070
00071 /**
00072  * The XGpio driver instance data. The user is required to allocate a
00073  * variable of this type for every GPIO device in the system. A pointer
00074  * to a variable of this type is then passed to the driver API functions.
00075  */
00076 typedef struct
00077 {
00078     Xuint32 BaseAddress;   /* Device base address */
00079     Xuint32 IsReady;      /* Device is initialized and ready */
00080 } XGpio;
00081
00082
00083 /***** Macros (Inline Functions) Definitions *****/
00084
00085
00086 /***** Function Prototypes *****/
00087
00088 /*
00089  * API Basic functions implemented in xgpio.c
00090  */
00091 XStatus XGpio_Initialize(XGpio *InstancePtr, Xuint16 DeviceId);
00092 void    XGpio_SetDataDirection(XGpio *InstancePtr, Xuint32 DirectionMask);
00093 Xuint32 XGpio_DiscreteRead(XGpio *InstancePtr);

```

```
00094 void      XGpio_DiscreteWrite(XGpio *InstancePtr, Xuint32 Mask);
00095
00096 XGpio_Config *XGpio_LookupConfig(Xuint16 DeviceId);
00097
00098 /*
00099  * API Functions implemented in xgpio_extra.c
00100  */
00101 void      XGpio_DiscreteSet(XGpio *InstancePtr, Xuint32 Mask);
00102 void      XGpio_DiscreteClear(XGpio *InstancePtr, Xuint32 Mask);
00103
00104 /*
00105  * API Functions implemented in xgpio_selftest.c
00106  */
00107 XStatus XGpio_SelfTest(XGpio *InstancePtr);
00108 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

gpio/v1_00_a/src/xgpio.h File Reference

Detailed Description

This file contains the software API definition of the Xilinx General Purpose I/O (**XGpio**) component.

The Xilinx GPIO controller is a soft IP core designed for Xilinx FPGAs and contains the following general features:

- Support for 8, 16, or 32 I/O discretes
- Each of the discretes can be configured for input or output.

Note:

This API utilizes 32 bit I/O to the GPIO registers. With 16 and 8 bit GPIO components, the unused bits from registers are read as zero and written as don't cares.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	03/13/02	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xgpio_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XGpio  
struct XGpio_Config
```

Functions

```
XStatus XGpio_Initialize (XGpio *InstancePtr, Xuint16 DeviceId)  
void XGpio_SetDataDirection (XGpio *InstancePtr, Xuint32 DirectionMask)  
Xuint32 XGpio_DiscreteRead (XGpio *InstancePtr)  
void XGpio_DiscreteWrite (XGpio *InstancePtr, Xuint32 Mask)  
XGpio_Config * XGpio_LookupConfig (Xuint16 DeviceId)  
void XGpio_DiscreteSet (XGpio *InstancePtr, Xuint32 Mask)  
void XGpio_DiscreteClear (XGpio *InstancePtr, Xuint32 Mask)  
XStatus XGpio_SelfTest (XGpio *InstancePtr)
```

Function Documentation

```
void XGpio_DiscreteClear( XGpio * InstancePtr,  
                        Xuint32 Mask  
                        )
```

Set output discrete(s) to logic 0.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Mask is the set of bits that will be set to 0 in the discrete data register. All other bits in the data register are unaffected.

Note:

None

```
Xuint32 XGpio_DiscreteRead( XGpio * InstancePtr)
```

Read state of discretes.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Returns:

Current copy of the discretes register.

Note:

None

```
void XGpio_DiscreteSet( XGpio * InstancePtr,  
                       Xuint32 Mask  
                       )
```

Set output discrete(s) to logic 1.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Mask is the set of bits that will be set to 1 in the discrete data register. All other bits in the data register are unaffected.

Note:

None

```
void XGpio_DiscreteWrite( XGpio * InstancePtr,  
                          Xuint32 Data  
                          )
```

Write to discretes register

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Data is the value to be written to the discretes register.

Note:

See also **XGpio_DiscreteSet()** and **XGpio_DiscreteClear()**.


```
XStatus XGpio_Initialize( XGpio * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initialize the **XGpio** instance provided by the caller based on the given DeviceID.

Nothing is done except to initialize the InstancePtr.

Parameters:

InstancePtr is a pointer to an **XGpio** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XGpio** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XGpio** component. Passing in a device id associates the generic **XGpio** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS Initialization was successful.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

NOTES:

None

```
XGpio_Config* XGpio_LookupConfig( Xuint16 DeviceId)
```

Lookup the device configuration based on the unique device ID. The table ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceID is the device identifier to lookup.

Returns:

- **XGpio** configuration structure pointer if DeviceID is found.
- XNULL if DeviceID is not found.

```
XStatus XGpio_SelfTest( XGpio * InstancePtr)
```

Run a self-test on the driver/device. This includes the following tests:

- Register reads.

ARGUMENTS:

Parameters:

InstancePtr is a pointer to the **XGpio** instance to be worked on. This parameter must have been previously initialized with **XGpio_Initialize()**.

Returns:

- XST_SUCCESS If test passed
- XST_FAILURE If test failed

Note:

Assume that the device is in it's reset state which means that the TRI register is set to all inputs. We cannot twiddle bits in the data register since this may lead to a real disaster (i.e. whatever is hooked to those pins gets activated when you'd least expect).

```
void XGpio_SetDataDirection( XGpio * InstancePtr,  
                             Xuint32 DirectionMask  
                             )
```

Set the input/output direction of all discrete signals.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

DirectionMask is a bitmask specifying which discrettes are input and which are output. Bits set to 0 are output and bits set to 1 are input.

Note:

None

XGpio Struct Reference

```
#include <xgpio.h>
```

Detailed Description

The XGpio driver instance data. The user is required to allocate a variable of this type for every GPIO device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [gpio/v1_00_a/src/xgpio.h](#)
-

gpio/v1_00_a/src/xgpios_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xgpios_l.h,v 1.1 2002/05/02 20:39:10 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file gpio/v1_00_a/src/xgpios_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  The user should refer to the
00029 * hardware device specification for more details of the device operation.
00030 * High-level driver functions are defined in xgpios.h.
00031 *
00032 * <pre>
00033 * MODIFICATION HISTORY:
00034 *
00035 * Ver   Who   Date       Changes
00036 * -----
00037 * 1.00b jhl   04/24/02 First release
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XGPIO_L_H /* prevent circular inclusions */
```

```

00043 #define XGPIO_L_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048 #include "xio.h"
00049
00050 /***** Constant Definitions *****/
00051
00052 /** @name Registers
00053  *
00054  * Register offsets for this device. This device does not utilize IPIF
00055  * @{
00056  */
00057 /**
00058  * - XGPIO_DATA_OFFSET      Data register
00059  * - XGPIO_TRI_OFFSET      Three state register (sets input/output direction)
00060  *                          0 configures pin for output and 1 for input.
00061  */
00062 #define XGPIO_DATA_OFFSET    0x00000000
00063 #define XGPIO_TRI_OFFSET    0x00000004
00064 /* @} */
00065
00066 /***** Type Definitions *****/
00067
00068
00069 /***** Macros (Inline Functions) Definitions *****/
00070
00071
00072 /*****
00073  *
00074  * Low-level driver macros. The list below provides signatures to help the
00075  * user use the macros.
00076  *
00077  * Xuint32 XGpio_mReadReg(Xuint32 BaseAddress, int RegOffset, Xuint32 Data)
00078  * void XGpio_mWriteReg(Xuint32 BaseAddress, int RegOffset, Xuint32 Data)
00079  *
00080  * void XGpio_mSetDataDirection(Xuint32 BaseAddress, Xuint32 DirectionMask)
00081  *
00082  * Xuint32 XGpio_mGetDataReg(Xuint32 BaseAddress)
00083  * void XGpio_mSetDataReg(Xuint32 BaseAddress, Xuint32 Mask)
00084  *
00085  *****/
00086
00087 /*****
00088  /**
00089  *
00090  * Write a value to a GPIO register. A 32 bit write is performed. If the
00091  * GPIO component is implemented in a smaller width, only the least
00092  * significant data is written.
00093  *

```

```

00094 * @param BaseAddress is the base address of the GPIO device.
00095 * @param RegOffset is the register offset from the base to write to.
00096 * @param Data is the data written to the register.
00097 *
00098 * @return None.
00099 *
00100 * @note None.
00101 *
00102 *****/
00103 #define XGpio_mWriteReg(BaseAddress, RegOffset, Data) \
00104     (XIo_Out32((BaseAddress) + (RegOffset), (Xuint32)(Data)))
00105
00106 /**
00107 **
00108 *
00109 * Read a value from a GPIO register. A 32 bit read is performed. If the
00110 * GPIO component is implemented in a smaller width, only the least
00111 * significant data is read from the register. The most significant data
00112 * will be read as 0.
00113 *
00114 * @param BaseAddress is the base address of the GPIO device.
00115 * @param Register is the register offset from the base to write to.
00116 * @param Data is the data written to the register.
00117 *
00118 * @return None.
00119 *
00120 * @note None.
00121 *
00122 *****/
00123 #define XGpio_mReadReg(BaseAddress, RegOffset) \
00124     (XIo_In32((BaseAddress) + (RegOffset)))
00125
00126 /**
00127 *
00128 * Set the input/output direction of all signals.
00129 *
00130 * @param BaseAddress contains the base address of the GPIO device.
00131 * @param DirectionMask is a bitmask specifying which discrettes are input and
00132 * which are output. Bits set to 0 are output and bits set to 1 are
00133 * input.
00134 *
00135 * @return None.
00136 *
00137 * @note None.
00138 *
00139 *****/
00140 #define XGpio_mSetDataDirection(BaseAddress, DirectionMask) \
00141     XGpio_mWriteReg((BaseAddress), XGPIO_TRI_OFFSET, (DirectionMask))
00142
00143
00144 /**
00145 **

```

```

00146 * Get the data register.
00147 *
00148 * @param    BaseAddress contains the base address of the GPIO device.
00149 *
00150 * @return   The contents of the data register.
00151 *
00152 * @note     None.
00153 *
00154 *****/
00155 #define XGpio_mGetDataReg(BaseAddress) \
00156     XGpio_mReadReg(BaseAddress, XGPIO_DATA_OFFSET)
00157
00158 *****/
00159 /**
00160 * Set the data register.
00161 *
00162 * @param    BaseAddress contains the base address of the GPIO device.
00163 * @param    Data is the value to be written to the data register.
00164 *
00165 * @return   None.
00166 *
00167 * @note     None.
00168 *
00169 *****/
00170 #define XGpio_mSetDataReg(BaseAddress, Data) \
00171     XGpio_mWriteReg((BaseAddress), XGPIO_DATA_OFFSET, (Data));
00172
00173
00174 *****/ Function Prototypes *****/
00175
00176 *****/ Variable Definitions *****/
00177
00178 #endif          /* end of protection macro */
00179

```

gpio/v1_00_a/src/xgpio_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xgpio.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Registers

Register offsets for this device. This device does not utilize IPIF registers.

```
#define XGPIO_DATA_OFFSET  
#define XGPIO_TRI_OFFSET
```


Defines

```
#define XGpio_mWriteReg(BaseAddress, RegOffset, Data)
#define XGpio_mReadReg(BaseAddress, RegOffset)
#define XGpio_mGetDataReg(BaseAddress)
#define XGpio_mSetDataReg(BaseAddress, Data)
```

Define Documentation

```
#define XGPIO_DATA_OFFSET
```

- **XGPIO_DATA_OFFSET** Data register
 - **XGPIO_TRI_OFFSET** Three state register (sets input/output direction) 0 configures pin for output and 1 for input.

```
#define XGpio_mGetDataReg( BaseAddress )
```

Get the data register.

Parameters:

BaseAddress contains the base address of the GPIO device.

Returns:

The contents of the data register.

Note:

None.

```
#define XGpio_mReadReg( BaseAddress,  
                          RegOffset )
```

Read a value from a GPIO register. A 32 bit read is performed. If the GPIO component is implemented in a smaller width, only the least significant data is read from the register. The most significant data will be read as 0.

Parameters:

BaseAddress is the base address of the GPIO device.
Register is the register offset from the base to write to.
Data is the data written to the register.

Returns:

None.

Note:

None.

```
#define XGpio_mSetDataReg( BaseAddress,  
                          Data          )
```

Set the data register.

Parameters:

BaseAddress contains the base address of the GPIO device.
Data is the value to be written to the data register.

Returns:

None.

Note:

None.

```
#define XGpio_mWriteReg( BaseAddress,  
                        RegOffset,  
                        Data          )
```

Write a value to a GPIO register. A 32 bit write is performed. If the GPIO component is implemented in a smaller width, only the least significant data is written.

Parameters:

- BaseAddress* is the base address of the GPIO device.
- RegOffset* is the register offset from the base to write to.
- Data* is the data written to the register.

Returns:

None.

Note:

None.

#define XGPIO_TRI_OFFSET

- XGPIO_DATA_OFFSET Data register
 - XGPIO_TRI_OFFSET Three state register (sets input/output direction) 0 configures pin for output and 1 for input.

gpio/v1_00_a/src/xgpio.c File Reference

Detailed Description

The implementation of the **XGpio** component's basic functionality. See [xgpio.h](#) for more information about the component.

Note:

None

```
#include "xparameters.h"  
#include "xgpio.h"  
#include "xgpio_i.h"  
#include "xstatus.h"
```

Functions

```
XStatus XGpio_Initialize (XGpio *InstancePtr, Xuint16 DeviceId)  
XGpio_Config * XGpio_LookupConfig (Xuint16 DeviceId)  
void XGpio_SetDataDirection (XGpio *InstancePtr, Xuint32 DirectionMask)  
Xuint32 XGpio_DiscreteRead (XGpio *InstancePtr)  
void XGpio_DiscreteWrite (XGpio *InstancePtr, Xuint32 Data)
```

Function Documentation

```
Xuint32 XGpio_DiscreteRead( XGpio * InstancePtr)
```

Read state of discretes.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Returns:

Current copy of the discretes register.

Note:

None

```
void XGpio_DiscreteWrite( XGpio * InstancePtr,  
                          Xuint32 Data  
                          )
```

Write to discretes register

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Data is the value to be written to the discretes register.

Note:

See also **XGpio_DiscreteSet()** and **XGpio_DiscreteClear()**.

```
XStatus XGpio_Initialize( XGpio * InstancePtr,  
                          Xuint16 DeviceId  
                          )
```

Initialize the **XGpio** instance provided by the caller based on the given DeviceID.

Nothing is done except to initialize the InstancePtr.

Parameters:

InstancePtr is a pointer to an **XGpio** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XGpio** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XGpio** component. Passing in a device id associates the generic **XGpio** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS Initialization was successful.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

NOTES:

None

XGpio_Config* XGpio_LookupConfig(Xuint16 DeviceId)

Lookup the device configuration based on the unique device ID. The table ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceID is the device identifier to lookup.

Returns:

- **XGpio** configuration structure pointer if DeviceID is found.
- XNULL if DeviceID is not found.

**void XGpio_SetDataDirection(XGpio * InstancePtr,
Xuint32 DirectionMask
)**

Set the input/output direction of all discrete signals.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

DirectionMask is a bitmask specifying which discrettes are input and which are output. Bits set to 0 are output and bits set to 1 are input.

Note:

None

Generated on 29 May 2003 for Xilinx Device Drivers

gpio/v1_00_a/src/xgpio_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xgpio_i.h,v 1.3 2002/05/02 20:44:36 linnj Exp $: */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022
00023 /**
00024 * @file gpio/v1_00_a/src/xgpio_i.h
00025 *
00026 * This header file contains internal identifiers, which are those shared
00027 * between the files of the driver. It is intended for internal use only.
00028 *
00029 * NOTES:
00030 *
00031 * None.
00032 *
00033 * <pre>
00034 * MODIFICATION HISTORY:
00035 *
00036 * Ver   Who   Date       Changes
00037 * ----  ---  -
00038 * 1.00a rmm   03/13/02 First release
00039 * </pre>
00040 *****/
00041
```



```
00042 #ifndef XGPIO_I_H /* prevent circular inclusions */
00043 #define XGPIO_I_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xgpio.h"
00048
00049 /***** Constant Definitions *****/
00050
00051
00052 /***** Type Definitions *****/
00053
00054
00055 /***** Macros (Inline Functions) Definitions *****/
00056
00057
00058 /***** Function Prototypes *****/
00059
00060
00061 /***** Variable Definitions *****/
00062
00063 extern XGpio_Config XGpio_ConfigTable[];
00064
00065 #endif /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

gpio/v1_00_a/src/xgpio_extra.c File Reference

Detailed Description

The implementation of the **XGpio** component's advanced discrete functions. See [xgpio.h](#) for more information about the component.

Note:

None

```
#include "xgpio.h"
```

Functions

```
void XGpio_DiscreteSet (XGpio *InstancePtr, Xuint32 Mask)
```

```
void XGpio_DiscreteClear (XGpio *InstancePtr, Xuint32 Mask)
```

Function Documentation

```
void XGpio_DiscreteClear( XGpio * InstancePtr,  
                          Xuint32 Mask  
                          )
```

Set output discrete(s) to logic 0.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Mask is the set of bits that will be set to 0 in the discrete data register. All other bits in the data register are unaffected.

Note:

None

```
void XGpio_DiscreteSet( XGpio * InstancePtr,  
                        Xuint32 Mask  
                        )
```

Set output discrete(s) to logic 1.

Parameters:

InstancePtr is a pointer to an **XGpio** instance to be worked on.

Mask is the set of bits that will be set to 1 in the discrete data register. All other bits in the data register are unaffected.

Note:

None

gpio/v1_00_a/src/xgpio_selftest.c File Reference

Detailed Description

The implementation of the [XGpio](#) component's self test function. See [xgpio.h](#) for more information about the component.

Note:

None

```
#include "xgpio.h"
```

Functions

[XStatus](#) [XGpio_SelfTest](#) ([XGpio](#) *InstancePtr)

Function Documentation

[XStatus](#) [XGpio_SelfTest](#)([XGpio](#) * *InstancePtr*)

Run a self-test on the driver/device. This includes the following tests:

- Register reads.

ARGUMENTS:

Parameters:

InstancePtr is a pointer to the **XGpio** instance to be worked on. This parameter must have been previously initialized with **XGpio_Initialize()**.

Returns:

- XST_SUCCESS If test passed
- XST_FAILURE If test failed

Note:

Assume that the device is in it's reset state which means that the TRI register is set to all inputs. We cannot twiddle bits in the data register since this may lead to a real disaster (i.e. whatever is hooked to those pins gets activated when you'd least expect).

gpio/v1_00_a/src/xgpio_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of GPIO devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	02/04/02	First release

```
#include "xgpio.h"  
#include "xparameters.h"
```

Variables

XGpio_Config [XGpio_ConfigTable](#) []

Variable Documentation

XGpio_Config XGpio_ConfigTable[]

This table contains configuration information for each GPIO device in the system.

iic/v1_01_c/src/xiic.h File Reference

Detailed Description

XIic is the driver for an IIC master or slave device.

In order to reduce the memory requirements of the driver it is partitioned such that there are optional parts of the driver. Slave, master, and multimaster features are optional such that these files are not required. In order to use the slave and multimaster features of the driver, the user must call functions (XIic_SlaveInclude and XIic_MultiMasterInclude) to dynamically include the code . These functions may be called at any time.

Bus Throttling

The IIC hardware provides bus throttling which allows either the device, as either a master or a slave, to stop the clock on the IIC bus. This feature allows the software to perform the appropriate processing for each interrupt without an unreasonable response restriction. With this design, it is important for the user to understand the implications of bus throttling.

Repeated Start

An application can send multiple messages, as a master, to a slave device and re-acquire the IIC bus each time a message is sent. The repeated start option allows the application to send multiple messages without re-acquiring the IIC bus for each message. This feature also could cause the application to lock up, or monopolize the IIC bus, should repeated start option be enabled and sequences of messages never end (periodic data collection). Also when repeated start is not disable before the last master message is sent or received, will leave the bus captive to the master, but unused.

Addressing

The IIC hardware is parameterized such that it can be built for 7 or 10 bit addresses. The driver provides the ability to control which address size is sent in messages as a master to a slave device. The address size which the hardware responds to as a slave is parameterized as 7 or 10 bits but fixed by the hardware build.

Addresses are represented as hex values with no adjustment for the data direction bit as the software manages address bit placement. This is especially important as the bit placement is not handled the same depending on which options are used such as repeated start and 7 vs 10 bit addressing.

Data Rates

The IIC hardware is parameterized such that it can be built to support data rates from DC to 400KBit. The frequency of the interrupts which occur is proportional to the data rate.

Polled Mode Operation

This driver does not provide a polled mode of operation primarily because polled mode which is non-blocking is difficult with the amount of interaction with the hardware that is necessary.

Interrupts

The device has many interrupts which allow IIC data transactions as well as bus status processing to occur.

The interrupts are divided into two types, data and status. Data interrupts indicate data has been received or transmitted while the status interrupts indicate the status of the IIC bus. Some of the interrupts, such as Not Addressed As Slave and Bus Not Busy, are only used when these specific events must be recognized as opposed to being enabled at all times.

Many of the interrupts are not a single event in that they are continuously present such that they must be disabled after recognition or when undesired. Some of these interrupts, which are data related, may be acknowledged by the software by reading or writing data to the appropriate register, or must be disabled. The following interrupts can be continuous rather than single events.

- Data Transmit Register Empty/Transmit FIFO Empty
- Data Receive Register Full/Receive FIFO
- Transmit FIFO Half Empty
- Bus Not Busy
- Addressed As Slave
- Not Addressed As Slave

The following interrupts are not passed directly to the application thru the status callback. These are only used internally for the driver processing and may result in the receive and send handlers being called to indicate completion of an operation. The following interrupts are data related rather than status.

- Data Transmit Register Empty/Transmit FIFO Empty
- Data Receive Register Full/Receive FIFO
- Transmit FIFO Half Empty
- Slave Transmit Complete

Interrupt To Event Mapping

The following table provides a mapping of the interrupts to the events which are passed to the status handler and the intended role (master or slave) for the event. Some interrupts can cause multiple events which are combined together into a single status event such as XII_MASTER_WRITE_EVENT and XII_GENERAL_CALL_EVENT

Interrupt	Event (s)	Role
Arbitration Lost Interrupt	XII_ARB_LOST_EVENT	Master
Transmit Error	XII_SLAVE_NO_ACK_EVENT	Master
IIC Bus Not Busy	XII_BUS_NOT_BUSY_EVENT	Master
Addressed As Slave	XII_MASTER_READ_EVENT,	Slave
	XII_MASTER_WRITE_EVENT,	Slave
	XII_GENERAL_CALL_EVENT	Slave

Not Addressed As Slave Interrupt

The Not Addressed As Slave interrupt is not passed directly to the application thru the status callback. It is used to determine the end of a message being received by a slave when there was no stop condition (repeated start). It will cause the receive handler to be called to indicate completion of the operation.

RTOS Independence

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xipif_v1_23_b.h"  
#include "xiic_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XIic  
struct XIic\_Config  
struct XIicStats
```

Configuration options

The following options may be specified or retrieved for the device and enable/disable additional features of the IIC bus. Each of the options are bit fields such that more than one may be specified.

```
#define XII\_GENERAL\_CALL\_OPTION  
#define XII\_REPEATED\_START\_OPTION  
#define XII\_SEND\_10\_BIT\_OPTION
```

Status events

The following status events occur during IIC bus processing and are passed to the status callback. Each event is only valid during the appropriate processing of the IIC bus. Each of these events are bit fields such that more than one may be specified.

```
#define XII\_BUS\_NOT\_BUSY\_EVENT  
#define XII\_ARB\_LOST\_EVENT  
#define XII\_SLAVE\_NO\_ACK\_EVENT  
#define XII\_MASTER\_READ\_EVENT  
#define XII\_MASTER\_WRITE\_EVENT  
#define XII\_GENERAL\_CALL\_EVENT
```

Defines

```
#define XII\_ADDR\_TO\_SEND\_TYPE  
#define XII\_ADDR\_TO\_RESPOND\_TYPE
```

Typedefs

```
typedef void(* XIic_Handler )(void *CallBackRef, int ByteCount)
typedef void(* XIic_StatusHandler )(void *CallBackRef, XStatus StatusEvent)
```

Functions

```
XStatus XIic_Initialize (XIic *InstancePtr, Xuint16 DeviceId)
XStatus XIic_Start (XIic *InstancePtr)
XStatus XIic_Stop (XIic *InstancePtr)
void XIic_Reset (XIic *InstancePtr)
XStatus XIic_SetAddress (XIic *InstancePtr, int AddressType, int Address)
Xuint16 XIic_GetAddress (XIic *InstancePtr, int AddressType)
XIic_Config * XIic_LookupConfig (Xuint16 DeviceId)
void XIic_InterruptHandler (void *InstancePtr)
void XIic_SetRecvHandler (XIic *InstancePtr, void *CallBackRef, XIic_Handler FuncPtr)
void XIic_SetSendHandler (XIic *InstancePtr, void *CallBackRef, XIic_Handler FuncPtr)
void XIic_SetStatusHandler (XIic *InstancePtr, void *CallBackRef, XIic_StatusHandler FuncPtr)
XStatus XIic_MasterRecv (XIic *InstancePtr, Xuint8 *RxMsgPtr, int ByteCount)
XStatus XIic_MasterSend (XIic *InstancePtr, Xuint8 *TxMsgPtr, int ByteCount)
void XIic_GetStats (XIic *InstancePtr, XIicStats *StatsPtr)
void XIic_ClearStats (XIic *InstancePtr)
XStatus XIic_SelfTest (XIic *InstancePtr)
void XIic_SetOptions (XIic *InstancePtr, Xuint32 Options)
Xuint32 XIic_GetOptions (XIic *InstancePtr)
void XIic_MultiMasterInclude (void)
```

Define Documentation

#define XII_ADDR_TO_RESPOND_TYPE

this device's bus address when slave

#define XII_ADDR_TO_SEND_TYPE

bus address of slave device

#define XII_ARB_LOST_EVENT

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define XII_BUS_NOT_BUSY_EVENT

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define XII_GENERAL_CALL_EVENT

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define XII_GENERAL_CALL_OPTION

XII_GENERAL_CALL_OPTION	The general call option allows an IIC slave to recognized the general call address. The status handler is called as usual indicating the device has been addressed as a slave with a general call. It is the application's responsibility to perform any special processing for the general call.
XII_REPEATED_START_OPTION	The repeated start option allows multiple messages to be sent/received on the IIC bus without rearbitrating for the bus. The messages are sent as a series of messages such that the option must be enabled before the 1st message of the series, to prevent an stop condition from being generated on the bus, and disabled before the last message of the series, to allow the stop condition to be generated.
XII_SEND_10_BIT_OPTION	The send 10 bit option allows 10 bit addresses to be sent on the bus when the device is a master. The device can be configured to respond as to 7 bit addresses even though it may be communicating with other devices that support 10 bit addresses. When this option is not enabled, only 7 bit addresses are sent on the bus.

#define XII_MASTER_READ_EVENT

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define XII_MASTER_WRITE_EVENT

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

#define XII_REPEATED_START_OPTION

XII_GENERAL_CALL_OPTION	The general call option allows an IIC slave to recognized the general call address. The status handler is called as usual indicating the device has been addressed as a slave with a general call. It is the application's responsibility to perform any special processing for the general call.
-------------------------	---

XII_REPEATED_START_OPTION	The repeated start option allows multiple messages to be sent/received on the IIC bus without rearbitrating for the bus. The messages are sent as a series of messages such that the option must be enabled before the 1st message of the series, to prevent an stop condition from being generated on the bus, and disabled before the last message of the series, to allow the stop condition to be generated.
---------------------------	--

XII_SEND_10_BIT_OPTION	The send 10 bit option allows 10 bit addresses to be sent on the bus when the device is a master. The device can be configured to respond as to 7 bit addresses even though it may be communicating with other devices that support 10 bit addresses. When this option is not enabled, only 7 bit addresses are sent on the bus.
------------------------	--

#define XII_SEND_10_BIT_OPTION

XII_GENERAL_CALL_OPTION	The general call option allows an IIC slave to recognized the general call address. The status handler is called as usual indicating the device has been addressed as a slave with a general call. It is the application's responsibility to perform any special processing for the general call.
XII_REPEATED_START_OPTION	The repeated start option allows multiple messages to be sent/received on the IIC bus without rearbitrating for the bus. The messages are sent as a series of messages such that the option must be enabled before the 1st message of the series, to prevent an stop condition from being generated on the bus, and disabled before the last message of the series, to allow the stop condition to be generated.
XII_SEND_10_BIT_OPTION	The send 10 bit option allows 10 bit addresses to be sent on the bus when the device is a master. The device can be configured to respond as to 7 bit addresses even though it may be communicating with other devices that support 10 bit addresses. When this option is not enabled, only 7 bit addresses are sent on the bus.

#define XII_SLAVE_NO_ACK_EVENT

XII_BUS_NOT_BUSY_EVENT	bus transitioned to not busy
XII_ARB_LOST_EVENT	arbitration was lost
XII_SLAVE_NO_ACK_EVENT	slave did not acknowledge data (had error)
XII_MASTER_READ_EVENT	master reading from slave
XII_MASTER_WRITE_EVENT	master writing to slave
XII_GENERAL_CALL_EVENT	general call to all slaves

Typedef Documentation

```
typedef void(* XIic_Handler)(void *CallBackRef, int ByteCount)
```

This callback function data type is defined to handle the asynchronous processing of sent and received data of the IIC driver. The application using this driver is expected to define a handler of this type to support interrupt driven mode. The handlers are called in an interrupt context such that minimal processing should be performed. The handler data type is utilized for both send and receive handlers.

Parameters:

CallBackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. Its type is unimportant to the driver component, so it is a void pointer.

ByteCount indicates the number of bytes remaining to be sent or received. A value of zero indicates that the requested number of bytes were sent or received.

```
typedef void(* XIic_StatusHandler)(void *CallBackRef, XStatus StatusEvent)
```

This callback function data type is defined to handle the asynchronous processing of status events of the IIC driver. The application using this driver is expected to define a handler of this type to support interrupt driven mode. The handler is called in an interrupt context such that minimal processing should be performed.

Parameters:

CallBackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. Its type is unimportant to the driver component, so it is a void pointer.

StatusEvent indicates one or more status events that occurred. See the definition of the status events above.

Function Documentation

```
void XIic_ClearStats( XIic * InstancePtr)
```

Clears the statistics for the IIC device by zeroing all counts.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint16 XIic_GetAddress( XIic * InstancePtr,  
                        int AddressType  
                        )
```

This function gets the addresses for the IIC device driver. The addresses include the device address that the device responds to as a slave, or the slave address to communicate with on the bus. The address returned has the same format whether 7 or 10 bits.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

AddressType indicates which address, the address which this responds to on the IIC bus as a slave, or the slave address to communicate with when this device is a master. One of the following values must be contained in this argument.

XII_ADDRESS_TO_SEND_TYPE	slave being addressed as a master
XII_ADDRESS_TO_RESPOND_TYPE	slave address to respond to as a slave

If neither of the two valid arguments are used, the function returns the address of the slave device

Returns:

The address retrieved.

Note:

None.

Xuint32 XIic_GetOptions(XIic * InstancePtr)

This function gets the current options for the IIC device. Options control the how the device behaves on the IIC bus. See SetOptions for more information on options.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

The options of the IIC device. See [xiic.h](#) for a list of available options.

Note:

Options enabled will have a 1 in its appropriate bit position.

```
void XIic_GetStats( XIic * InstancePtr,
                  XIicStats * StatsPtr
                  )
```

Gets a copy of the statistics for an IIC device.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

StatsPtr is a pointer to a **XIicStats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.


```
XStatus Xlic_Initialize( Xlic * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initializes a specific **Xlic** instance. The initialization entails:

- Check the device has an entry in the configuration table.
- Initialize the driver to allow access to the device registers and initialize other subcomponents necessary for the operation of the device.
- Default options to:
 - 7-bit slave addressing
 - Send messages as a slave device
 - Repeated start off
 - General call recognition disabled
- Clear messaging and error statistics

The **Xlic_Start()** function must be called after this function before the device is ready to send and receive data on the IIC bus.

Before **Xlic_Start()** is called, the interrupt control must connect the ISR routine to the interrupt handler. This is done by the user, and not **Xlic_Start()** to allow the user to use an interrupt controller of their choice.

Parameters:

InstancePtr is a pointer to the **Xlic** instance to be worked on.

DeviceId is the unique id of the device controlled by this **Xlic** instance. Passing in a device id associates the generic **Xlic** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS when successful
- XST_DEVICE_IS_STARTED indicates the device is started (i.e. interrupts enabled and messaging is possible). Must stop before re-initialization is allowed.

Note:

None.

```
void Xlic_InterruptHandler( void * InstancePtr)
```

This function is the interrupt handler for the **Xlic** driver. This function should be connected to the interrupt system.

Only one interrupt source is handled for each interrupt allowing higher priority system interrupts quicker response time.

Parameters:

InstancePtr is a pointer to the **Xlic** instance to be worked on.

Returns:

None.

```
Xlic_Config* Xlic_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID. The table IicConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for

Returns:

A pointer to the configuration data of the device, or XNULL if no match is found.

Note:

None.

```
XStatus XIic_MasterRecv( XIic * InstancePtr,  
                        Xuint8 * RxMsgPtr,  
                        int ByteCount  
                        )
```

This function receives data as a master from a slave device on the IIC bus. If the bus is busy, it will indicate so and then enable an interrupt such that the status handler will be called when the bus is no longer busy. The slave address which has been set with the [XIic_SetAddress\(\)](#) function is the address from which data is received. Receiving data on the bus performs a read operation.

Parameters:

InstancePtr is a pointer to the Iic instance to be worked on.

RxMsgPtr is a pointer to the data to be transmitted

ByteCount is the number of message bytes to be sent

Returns:

- XST_SUCCESS indicates the message reception processes has been initiated.
- XST_IIC_BUS_BUSY indicates the bus was in use and that the BusNotBusy interrupt is enabled which will update the EventStatus when the bus is no longer busy.
- XST_IIC_GENERAL_CALL_ADDRESS indicates the slave address is set to the the general call address. This is not allowed for Master receive mode.

```
XStatus XIic_MasterSend( XIic * InstancePtr,  
                        Xuint8 * TxMsgPtr,  
                        int ByteCount  
                        )
```

This function sends data as a master on the IIC bus. If the bus is busy, it will indicate so and then enable an interrupt such that the status handler will be called when the bus is no longer busy. The slave address which has been set with the [XIic_SetAddress\(\)](#) function is the address to which the specific data is sent. Sending data on the bus performs a write operation.

Parameters:

InstancePtr points to the Iic instance to be worked on.

TxMsgPtr points to the data to be transmitted

ByteCount is the number of message bytes to be sent

Returns:

- XST_SUCCESS indicates the message transmission has been initiated.
- XST_IIC_BUS_BUSY indicates the bus was in use and that the BusNotBusy interrupt is enabled which will update the EventStatus when the bus is no longer busy.

Note:

None

void XIic_MultiMasterInclude(void)

This function includes multi-master code such that multi-master events are handled properly. Multi-master events include a loss of arbitration and the bus transitioning from busy to not busy. This function allows the multi-master processing to be optional. This function must be called prior to allowing any multi-master events to occur, such as after the driver is initialized.

Note:

None

void XIic_Reset(XIic * InstancePtr)

Resets the IIC device. Reset must only be called after the driver has been initialized. The configuration after this reset is as follows:

- Repeated start is disabled
- General call is disabled

The upper layer software is responsible for initializing and re-configuring (if necessary) and restarting the IIC device after the reset.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

None.

Note:

None.

XStatus XIic_SelfTest(XIic * InstancePtr)

Runs a limited self-test on the driver/device. The self-test is destructive in that a reset of the device is performed in order to check the reset values of the registers and to get the device into a known state. There is no loopback capabilities for the device such that this test does not send or receive data.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

XST_SUCCESS

No errors found

XST_IIC_STAND_REG_ERROR

One or more IIC regular registers did not zero on reset or read back correctly based on what was written to it

XST_IIC_TX_FIFO_REG_ERROR

One or more IIC parametrizable TX FIFO registers did not zero on reset or read back correctly based on what was written to it

XST_IIC_RX_FIFO_REG_ERROR	One or more IIC parametrizable RX FIFO registers did not zero on reset or read back correctly based on what was written to it
XST_IIC_STAND_REG_RESET_ERROR	A non parameterizable reg value after reset not valid
XST_IIC_TX_FIFO_REG_RESET_ERROR	Tx fifo, included in design, value after reset not valid
XST_IIC_RX_FIFO_REG_RESET_ERROR	Rx fifo, included in design, value after reset not valid
XST_IIC_TBA_REG_RESET_ERROR	10 bit addr, incl in design, value after reset not valid
XST_IIC_CR_READBACK_ERROR	Read of the control register didn't return value written
XST_IIC_DTR_READBACK_ERROR	Read of the data Tx reg didn't return value written
XST_IIC_DRR_READBACK_ERROR	Read of the data Receive reg didn't return value written
XST_IIC_ADR_READBACK_ERROR	Read of the data Tx reg didn't return value written
XST_IIC_TBA_READBACK_ERROR	Read of the 10 bit addr reg didn't return written value

Note:

Only the registers that have be included into the hardware design are tested, such as, 10-bit vs 7-bit addressing.

```
XStatus XIic_SetAddress( XIic * InstancePtr,
                        int AddressType,
                        int Address
                      )
```

This function sets the bus addresses. The addresses include the device address that the device responds to as a slave, or the slave address to communicate with on the bus. The IIC device hardware is built to allow either 7 or 10 bit slave addressing only at build time rather than at run time. When this device is a master, slave addressing can be selected at run time to match addressing modes for other bus devices.

Addresses are represented as hex values with no adjustment for the data direction bit as the software manages address bit placement. Example: For a 7 address written to the device of 1010 011X where X is the transfer direction (send/recv), the address parameter for this function needs to be 01010011 or 0x53 where the correct bit allignment will be handled for 7 as well as 10 bit devices. This is especially important as the bit placement is not handled the same depending on which options are used such as repeated start.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

AddressType indicates which address is being modified; the address which this device responds to on the IIC bus as a slave, or the slave address to communicate with when this device is a master. One of the following values must be contained in this argument.

XII_ADDRESS_TO_SEND	Slave being addressed by a this master
XII_ADDRESS_TO_RESPOND	Address to respond to as a slave device

Address contains the address to be set; 7 bit or 10 bit address. A ten bit address must be within the range: 0 - 1023 and a 7 bit address must be within the range 0 - 127.

Returns:

- XST_SUCCESS is returned if the address was successfully set, otherwise one of the following errors is returned.
- XST_IIC_NO_10_BIT_ADDRESSING indicates only 7 bit addressing supported.
 - XST_INVALID_PARAM indicates an invalid parameter was specified.

Note:

Upper bits of 10-bit address is written only when current device is built as a ten bit device.

```
void XIic_SetOptions( XIic * InstancePtr,
                    Xuint32 NewOptions
                    )
```

This function sets the options for the IIC device driver. The options control how the device behaves relative to the IIC bus. If an option applies to how messages are sent or received on the IIC bus, it must be set prior to calling functions which send or receive data.

To set multiple options, the values must be ORed together. To not change existing options, read/modify/write with the current options using [XIic_GetOptions\(\)](#).

USAGE EXAMPLE:

Read/modify/write to enable repeated start:

```
Xuint8 Options;
Options = XIic_GetOptions(&Iic);
XIic_SetOptions(&Iic, Options | XII_REPEATED_START_OPTION);
```

Disabling General Call:

```
Options = XIic_GetOptions(&Iic);
XIic_SetOptions(&Iic, Options &= ~XII_GENERAL_CALL_OPTION);
```

Parameters:

InstancePtr is a pointer to the [XIic](#) instance to be worked on.

NewOptions are the options to be set. See [xiic.h](#) for a list of the available options.

Returns:

None.

Note:

Sending or receiving messages with repeated start enabled, and then disabling repeated start, will not take effect until another master transaction is completed. i.e. After using repeated start, the bus will continue to be throttled after repeated start is disabled until a master transaction occurs allowing the IIC to release the bus.

Options enabled will have a 1 in its appropriate bit position.

```
void Xlic_SetRecvHandler( Xlic *           InstancePtr,
                        void *           CallbackRef,
                        Xlic_Handler FuncPtr
                        )
```

Sets the receive callback function, the receive handler, which the driver calls when it finishes receiving data. The number of bytes used to signal when the receive is complete is the number of bytes set in the `Xlic_Recv` function.

The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context.

The number of bytes received is passed to the handler as an argument.

Parameters:

InstancePtr is a pointer to the **Xlic** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
void Xlic_SetSendHandler( Xlic *           InstancePtr,
                        void *           CallbackRef,
                        Xlic_Handler FuncPtr
                        )
```

Sets the send callback function, the send handler, which the driver calls when it receives confirmation of sent data. The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context.

Parameters:

InstancePtr the pointer to the **Xlic** instance to be worked on.

CallbackRef the upper layer callback reference passed back when the callback function is invoked.

FuncPtr the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
void Xlic_SetStatusHandler( Xlic *           InstancePtr,
                          void *           CallbackRef,
                          Xlic_StatusHandler FuncPtr
                          )
```

Sets the status callback function, the status handler, which the driver calls when it encounters conditions which are not data related. The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context. The status events that can be returned are described in [xiic.h](#).

Parameters:

- InstancePtr* points to the [XIic](#) instance to be worked on.
- CallBackRef* is the upper layer callback reference passed back when the callback function is invoked.
- FuncPtr* is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

XStatus XIic_Start(XIic * InstancePtr)

This function starts the IIC device and driver by enabling the proper interrupts such that data may be sent and received on the IIC bus. This function must be called before the functions to send and receive data.

Before [XIic_Start\(\)](#) is called, the interrupt control must connect the ISR routine to the interrupt handler. This is done by the user, and not [XIic_Start\(\)](#) to allow the user to use an interrupt controller of their choice.

Start enables:

- IIC device
- Interrupts:
 - Addressed as slave to allow messages from another master
 - Arbitration Lost to detect Tx arbitration errors
 - Global IIC interrupt within the IPIF interface

Parameters:

InstancePtr is a pointer to the [XIic](#) instance to be worked on.

Returns:

XST_SUCCESS always

Note:

The device interrupt is connected to the interrupt controller, but no "messaging" interrupts are enabled. Addressed as Slave is enabled to reception of messages when this devices address is written to the bus. The correct messaging interrupts are enabled when sending or receiving via the [IicSend\(\)](#) and [IicRecv\(\)](#) functions. No action is required by the user to control any IIC interrupts as the driver completely manages all 8 interrupts. Start and Stop control the ability to use the device. Stopping the device completely stops all device interrupts from the processor.

XStatus XIic_Stop(XIic * InstancePtr)

This function stops the IIC device and driver such that data is no longer sent or received on the IIC bus. This function stops the device by disabling interrupts. This function only disables interrupts within the device such that the caller is responsible for disconnecting the interrupt handler of the device from the interrupt source and disabling interrupts at other levels.

Due to bus throttling that could hold the bus between messages when using repeated start option, stop will not occur when the device is actively sending or receiving data from the IIC bus or the bus is being throttled by this device, but instead return XST_IIC_BUS_BUSY.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

- XST_SUCCESS indicates all IIC interrupts are disabled. No messages can be received or transmitted until **XIic_Start()** is called.
- XST_IIC_BUS_BUSY indicates this device is currently engaged in message traffic and cannot be stopped.

Note:

None.

Xlic Struct Reference

```
#include <xiic.h>
```

Detailed Description

The Xlic driver instance data. The user is required to allocate a variable of this type for every IIC device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `iic/v1_01_c/src/xiic.h`
-

iic/v1_01_c/src/xiic.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xiic.h,v 1.3 2003/01/13 21:47:59 meinelte Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file iic/v1_01_c/src/xiic.h
00026 *
00027 * XIic is the driver for an IIC master or slave device.
00028 *
00029 * In order to reduce the memory requirements of the driver it is partitioned
00030 * such that there are optional parts of the driver.  Slave, master, and
00031 * multimaster features are optional such that these files are not required.
00032 * In order to use the slave and multimaster features of the driver, the user
00033 * must call functions (XIic_SlaveInclude and XIic_MultiMasterInclude)
00034 * to dynamically include the code .  These functions may be called at any time.
00035 *
00036 * <b>Bus Throttling</b>
00037 *
00038 * The IIC hardware provides bus throttling which allows either the device, as
00039 * either a master or a slave, to stop the clock on the IIC bus. This feature
00040 * allows the software to perform the appropriate processing for each interrupt
00041 * without an unreasonable response restriction.  With this design, it is
00042 * important for the user to understand the implications of bus throttling.
```

00043 *
00044 * **Repeated Start**
00045 *
00046 * An application can send multiple messages, as a master, to a slave device
00047 * and re-acquire the IIC bus each time a message is sent. The repeated start
00048 * option allows the application to send multiple messages without re-acquiring
00049 * the IIC bus for each message. This feature also could cause the application
00050 * to lock up, or monopolize the IIC bus, should repeated start option be
00051 * enabled and sequences of messages never end (periodic data collection).
00052 * Also when repeated start is not disabled before the last master message is
00053 * sent or received, will leave the bus captive to the master, but unused.
00054 *
00055 * **Addressing**
00056 *
00057 * The IIC hardware is parameterized such that it can be built for 7 or 10
00058 * bit addresses. The driver provides the ability to control which address
00059 * size is sent in messages as a master to a slave device. The address size
00060 * which the hardware responds to as a slave is parameterized as 7 or 10 bits
00061 * but fixed by the hardware build.
00062 *
00063 * Addresses are represented as hex values with no adjustment for the data
00064 * direction bit as the software manages address bit placement. This is
00065 * especially important as the bit placement is not handled the same depending
00066 * on which options are used such as repeated start and 7 vs 10 bit addressing.
00067 *
00068 * **Data Rates**
00069 *
00070 * The IIC hardware is parameterized such that it can be built to support
00071 * data rates from DC to 400KBit. The frequency of the interrupts which
00072 * occur is proportional to the data rate.
00073 *
00074 * **Polled Mode Operation**
00075 *
00076 * This driver does not provide a polled mode of operation primarily because
00077 * polled mode which is non-blocking is difficult with the amount of
00078 * interaction with the hardware that is necessary.
00079 *
00080 * **Interrupts**
00081 *
00082 * The device has many interrupts which allow IIC data transactions as well
00083 * as bus status processing to occur.
00084 *
00085 * The interrupts are divided into two types, data and status. Data interrupts
00086 * indicate data has been received or transmitted while the status interrupts
00087 * indicate the status of the IIC bus. Some of the interrupts, such as Not
00088 * Addressed As Slave and Bus Not Busy, are only used when these specific
00089 * events must be recognized as opposed to being enabled at all times.
00090 *
00091 * Many of the interrupts are not a single event in that they are continuously
00092 * present such that they must be disabled after recognition or when undesired.
00093 * Some of these interrupts, which are data related, may be acknowledged by the
00094 * software by reading or writing data to the appropriate register, or must
00095 * be disabled. The following interrupts can be continuous rather than single

```

00096 * events.
00097 *   - Data Transmit Register Empty/Transmit FIFO Empty
00098 *   - Data Receive Register Full/Receive FIFO
00099 *   - Transmit FIFO Half Empty
00100 *   - Bus Not Busy
00101 *   - Addressed As Slave
00102 *   - Not Addressed As Slave
00103 *
00104 * The following interrupts are not passed directly to the application thru the
00105 * status callback. These are only used internally for the driver processing
00106 * and may result in the receive and send handlers being called to indicate
00107 * completion of an operation. The following interrupts are data related
00108 * rather than status.
00109 *   - Data Transmit Register Empty/Transmit FIFO Empty
00110 *   - Data Receive Register Full/Receive FIFO
00111 *   - Transmit FIFO Half Empty
00112 *   - Slave Transmit Complete
00113 *
00114 * <b>Interrupt To Event Mapping</b>
00115 *
00116 * The following table provides a mapping of the interrupts to the events which
00117 * are passed to the status handler and the intended role (master or slave) for
00118 * the event. Some interrupts can cause multiple events which are combined
00119 * together into a single status event such as XII_MASTER_WRITE_EVENT and
00120 * XII_GENERAL_CALL_EVENT
00121 * <pre>
00122 * Interrupt                               Event(s)                               Role
00123 *
00124 * Arbitration Lost Interrupt             XII_ARB_LOST_EVENT                     Master
00125 * Transmit Error                         XII_SLAVE_NO_ACK_EVENT                 Master
00126 * IIC Bus Not Busy                       XII_BUS_NOT_BUSY_EVENT                 Master
00127 * Addressed As Slave                     XII_MASTER_READ_EVENT,                 Slave
00128 *                                         XII_MASTER_WRITE_EVENT,                 Slave
00129 *                                         XII_GENERAL_CALL_EVENT                 Slave
00130 * </pre>
00131 * <b>Not Addressed As Slave Interrupt</b>
00132 *
00133 * The Not Addressed As Slave interrupt is not passed directly to the
00134 * application thru the status callback. It is used to determine the end of
00135 * a message being received by a slave when there was no stop condition
00136 * (repeated start). It will cause the receive handler to be called to
00137 * indicate completion of the operation.
00138 *
00139 * <b>RTOS Independence</b>
00140 *
00141 * This driver is intended to be RTOS and processor independent. It works
00142 * with physical addresses only. Any needs for dynamic memory management,
00143 * threads or thread mutual exclusion, virtual memory, or cache control must
00144 * be satisfied by the layer above this driver.
00145 *
00146 * <pre>
00147 * MODIFICATION HISTORY:

```

```

00148 *
00149 * Ver    Who    Date      Changes
00150 * -----
00151 * 1.01a rfp   10/19/01 release
00152 * 1.01c ecm   12/05/02 new rev
00153 * </pre>
00154 *
00155 *****/
00156
00157 #ifndef XIIC_H /* prevent circular inclusions */
00158 #define XIIC_H /* by using protection macros */
00159
00160 /***** Include Files *****/
00161
00162 #include "xbasic_types.h"
00163 #include "xstatus.h"
00164 #include "xipif_v1_23_b.h"
00165 #include "xiic_l.h"
00166
00167 /***** Constant Definitions *****/
00168
00169 /** @name Configuration options
00170  *
00171  * The following options may be specified or retrieved for the device and
00172  * enable/disable additional features of the IIC bus. Each of the options
00173  * are bit fields such that more than one may be specified.
00174  * @{
00175  */
00176 /**
00177  * <pre>
00178  * XII_GENERAL_CALL_OPTION          The general call option allows an IIC slave to
00179  *                                  recognized the general call address. The status
00180  *                                  handler is called as usual indicating the
device
00181  *                                  has been addressed as a slave with a general
00182  *                                  call. It is the application's responsibility to
00183  *                                  perform any special processing for the general
00184  *                                  call.
00185  *
00186  * XII_REPEATED_START_OPTION        The repeated start option allows multiple
00187  *                                  messages to be sent/received on the IIC bus
00188  *                                  without rearbitrating for the bus. The
messages
00189  *                                  are sent as a series of messages such that the
00190  *                                  option must be enabled before the 1st message
of
00191  *                                  the series, to prevent an stop condition from
00192  *                                  being generated on the bus, and disabled before
00193  *                                  the last message of the series, to allow the
00194  *                                  stop condition to be generated.
00195  *
00196  * XII_SEND_10_BIT_OPTION           The send 10 bit option allows 10 bit addresses

```

```

00197 * to be sent on the bus when the device is a
00198 * master. The device can be configured to respond
00199 * as to 7 bit addresses even though it may be
00200 * communicating with other devices that support
10
00201 * bit addresses. When this option is not
enabled,
00202 * only 7 bit addresses are sent on the bus.
00203 *
00204 * </pre>
00205 */
00206 #define XII_GENERAL_CALL_OPTION 0x00000001
00207 #define XII_REPEATED_START_OPTION 0x00000002
00208 #define XII_SEND_10_BIT_OPTION 0x00000004
00209
00210 /*@}*/
00211
00212 /** @name Status events
00213 *
00214 * The following status events occur during IIC bus processing and are passed
00215 * to the status callback. Each event is only valid during the appropriate
00216 * processing of the IIC bus. Each of these events are bit fields such that
00217 * more than one may be specified.
00218 * @{
00219 */
00220 /**
00221 * <pre>
00222 * XII_BUS_NOT_BUSY_EVENT bus transitioned to not busy
00223 * XII_ARB_LOST_EVENT arbitration was lost
00224 * XII_SLAVE_NO_ACK_EVENT slave did not acknowledge data (had error)
00225 * XII_MASTER_READ_EVENT master reading from slave
00226 * XII_MASTER_WRITE_EVENT master writing to slave
00227 * XII_GENERAL_CALL_EVENT general call to all slaves
00228 * </pre>
00229 */
00230 #define XII_BUS_NOT_BUSY_EVENT 0x00000001
00231 #define XII_ARB_LOST_EVENT 0x00000002
00232 #define XII_SLAVE_NO_ACK_EVENT 0x00000004
00233 #define XII_MASTER_READ_EVENT 0x00000008
00234 #define XII_MASTER_WRITE_EVENT 0x00000010
00235 #define XII_GENERAL_CALL_EVENT 0x00000020
00236 /*@}*/
00237
00238
00239 /* The following address types are used when setting and getting the addresses
00240 * of the driver. These are mutually exclusive such that only one or the other
00241 * may be specified.
00242 */
00243 /** bus address of slave device */
00244 #define XII_ADDR_TO_SEND_TYPE 1
00245 /** this device's bus address when slave */

```

```

00246 #define XII_ADDR_TO_RESPOND_TYPE      2
00247
00248 /***** Type Definitions *****/
00249
00250 /**
00251  * This typedef contains configuration information for the device.
00252  */
00253 typedef struct
00254 {
00255     Xuint16 DeviceId;          /**< Unique ID  of device */
00256     Xuint32 BaseAddress;      /**< Device base address */
00257     Xboolean Has10BitAddr;    /**< does device have 10 bit address decoding */
00258 } XIic_Config;
00259
00260 /**
00261  * This callback function data type is defined to handle the asynchronous
00262  * processing of sent and received data of the IIC driver.  The application
00263  * using this driver is expected to define a handler of this type to support
00264  * interrupt driven mode.  The handlers are called in an interrupt context such
00265  * that minimal processing should be performed.  The handler data type is
00266  * utilized for both send and receive handlers.
00267  *
00268  * @param CallbackRef is a callback reference passed in by the upper layer when
00269  *         setting the callback functions, and passed back to the upper layer
00270  *         when the callback is invoked.  Its type is unimportant to the driver
00271  *         component, so it is a void pointer.
00272  *
00273  * @param ByteCount indicates the number of bytes remaining to be sent or
00274  *         received.  A value of zero indicates that the requested number of
00275  *         bytes were sent or received.
00276  */
00277 typedef void (*XIic_Handler)(void *CallbackRef, int ByteCount);
00278
00279 /**
00280  * This callback function data type is defined to handle the asynchronous
00281  * processing of status events of the IIC driver.  The application using
00282  * this driver is expected to define a handler of this type to support
00283  * interrupt driven mode.  The handler is called in an interrupt context such
00284  * that minimal processing should be performed.
00285  *
00286  * @param CallbackRef is a callback reference passed in by the upper layer when
00287  *         setting the callback functions, and passed back to the upper layer
00288  *         when the callback is invoked.  Its type is unimportant to the driver
00289  *         component, so it is a void pointer.
00290  *
00291  * @param StatusEvent indicates one or more status events that occurred.  See
00292  *         the definition of the status events above.
00293  */
00294 typedef void (*XIic_StatusHandler)(void *CallbackRef, XStatus StatusEvent);
00295
00296 /**
00297  * XIic statistics

```

```

00298  */
00299 typedef struct
00300 {
00301     Xuint8 ArbitrationLost;      /**< Number of times arbitration was lost */
00302     Xuint8 RepeatedStarts;      /**< Number of repeated starts */
00303     Xuint8 BusBusy;             /**< Number of times bus busy status returned */
00304     Xuint8 RecvBytes;           /**< Number of bytes received */
00305     Xuint8 RecvInterrupts;      /**< Number of receive interrupts */
00306     Xuint8 SendBytes;           /**< Number of transmit bytes received */
00307     Xuint8 SendInterrupts;      /**< Number of transmit interrupts */
00308     Xuint8 TxErrors;            /**< Number of transmit errors (no ack) */
00309     Xuint8 IicInterrupts;       /**< Number of IIC (device) interrupts */
00310 } XIicStats;
00311
00312
00313 /**
00314  * The XIic driver instance data. The user is required to allocate a
00315  * variable of this type for every IIC device in the system. A pointer
00316  * to a variable of this type is then passed to the driver API functions.
00317  */
00318 typedef struct
00319 {
00320     XIicStats Stats;            /* Statistics */
00321     Xuint32 BaseAddress;        /* Device base address */
00322     Xboolean Has10BitAddr;      /* XTRUE when 10 bit addressing in design */
00323     Xuint32 IsReady;            /* Device is initialized and ready */
00324     Xuint32 IsStarted;          /* Device has been started */
00325     int AddrOfSlave;           /* Slave addr writing to */
00326
00327     Xuint32 Options;            /* current operating options */
00328     Xuint8 *SendBufferPtr;      /* Buffer to send (state) */
00329     Xuint8 *RecvBufferPtr;      /* Buffer to receive (state) */
00330     Xuint8 TxAddrMode;          /* State of Tx Address transmission */
00331     int SendByteCount;          /* Number of data bytes in buffer (state) */
00332     int RecvByteCount;          /* Number of empty bytes in buffer (state) */
00333
00334     Xboolean BNBOOnly;          /* XTRUE when BNB interrupt needs to */
00335                                 /* call callback */
00336
00337     XIic_StatusHandler StatusHandler;
00338     void *StatusCallBackRef;    /* Callback reference for status handler */
00339     XIic_Handler RecvHandler;
00340     void *RecvCallBackRef;      /* Callback reference for recv handler */
00341     XIic_Handler SendHandler;
00342     void *SendCallBackRef;      /* Callback reference for send handler */
00343
00344 } XIic;
00345
00346

```



```
00347 /***** Macros (Inline Functions) Definitions *****/
00348
00349
00350 /***** Function Prototypes *****/
00351
00352 /*
00353  * Required functions in xiic.c
00354  */
00355 XStatus XIic_Initialize(XIic *InstancePtr, Xuint16 DeviceId);
00356
00357 XStatus XIic_Start(XIic *InstancePtr);
00358 XStatus XIic_Stop(XIic *InstancePtr);
00359
00360 void XIic_Reset(XIic *InstancePtr);
00361
00362 XStatus XIic_SetAddress(XIic *InstancePtr, int AddressType, int Address);
00363 Xuint16 XIic_GetAddress(XIic *InstancePtr, int AddressType);
00364
00365 XIic_Config *XIic_LookupConfig(Xuint16 DeviceId);
00366
00367 /*
00368  * Interrupt (currently required) functions in xiic_intr.c
00369  */
00370 void XIic_InterruptHandler(void *InstancePtr);
00371 void XIic_SetRecvHandler(XIic *InstancePtr, void *CallBackRef,
00372                          XIic_Handler FuncPtr);
00373 void XIic_SetSendHandler(XIic *InstancePtr, void *CallBackRef,
00374                          XIic_Handler FuncPtr);
00375 void XIic_SetStatusHandler(XIic *InstancePtr, void *CallBackRef,
00376                            XIic_StatusHandler FuncPtr);
00377 /*
00378  * Master send and receive functions in xiic_master.c
00379  */
00380 XStatus XIic_MasterRecv(XIic *InstancePtr, Xuint8 *RxMsgPtr, int ByteCount);
00381 XStatus XIic_MasterSend(XIic *InstancePtr, Xuint8 *TxMsgPtr, int ByteCount);
00382
00383 /*
00384  * Slave send and receive functions in xiic_slave.c
00385  */
00386 void XIic_SlaveInclude(void);
00387 XStatus XIic_SlaveRecv(XIic *InstancePtr, Xuint8 *RxMsgPtr, int ByteCount);
00388 XStatus XIic_SlaveSend(XIic *InstancePtr, Xuint8 *TxMsgPtr, int ByteCount);
00389
00390 /*
00391  * Statistics functions in xiic_stats.c
00392  */
00393 void XIic_GetStats(XIic *InstancePtr, XIicStats *StatsPtr);
00394 void XIic_ClearStats(XIic *InstancePtr);
00395
00396 /*
```

```
00397 * Self test functions in xiic_selftest.c
00398 */
00399 XStatus XIic_SelfTest(XIic *InstancePtr);
00400
00401 /*
00402 * Options functions in xiic_options.c
00403 */
00404 void XIic_SetOptions(XIic *InstancePtr, Xuint32 Options);
00405 Xuint32 XIic_GetOptions(XIic *InstancePtr);
00406
00407 /*
00408 * Multi-master functions in xiic_multi_master.c
00409 */
00410 void XIic_MultiMasterInclude(void);
00411
00412
00413 #endif                /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

iic/v1_01_c/src/xiic_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xiic.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00b	jhl	05/07/02	First release
1.01c	ecm	12/05/02	new rev

```
#include "xbasic\_types.h"
```

[Go to the source code of this file.](#)

Functions

unsigned [Xlic_Recv](#) ([Xuint32](#) BaseAddress, [Xuint8](#) Address, [Xuint8](#) *BufferPtr, unsigned ByteCount)
unsigned [Xlic_Send](#) ([Xuint32](#) BaseAddress, [Xuint8](#) Address, [Xuint8](#) *BufferPtr, unsigned ByteCount)

Function Documentation

```
unsigned XIic_Recv( Xuint32  BaseAddress,  
                  Xuint8   Address,  
                  Xuint8 * BufferPtr,  
                  unsigned ByteCount  
                  )
```

Receive data as a master on the IIC bus. This function receives the data using polled I/O and blocks until the data has been received. It only supports 7 bit addressing and non-repeated start modes of operation. The user is responsible for ensuring the bus is not busy if multiple masters are present on the bus.

Parameters:

BaseAddress contains the base address of the IIC device.

Address contains the 7 bit IIC address of the device to send the specified data to.

BufferPtr points to the data to be sent.

ByteCount is the number of bytes to be sent.

Returns:

The number of bytes received.

Note:

None

```
unsigned XIic_Send( Xuint32  BaseAddress,  
                  Xuint8   Address,  
                  Xuint8 * BufferPtr,  
                  unsigned ByteCount  
                  )
```

Send data as a master on the IIC bus. This function sends the data using polled I/O and blocks until the data has been sent. It only supports 7 bit addressing and non-repeated start modes of operation. The user is responsible for ensuring the bus is not busy if multiple masters are present on the bus.

Parameters:

BaseAddress contains the base address of the IIC device.

Address contains the 7 bit IIC address of the device to send the specified data to.

BufferPtr points to the data to be sent.

ByteCount is the number of bytes to be sent.

Returns:

The number of bytes sent.

Note:

None

Generated on 29 May 2003 for Xilinx Device Drivers

iic/v1_01_c/src/xiic_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xiic_l.h,v 1.2 2002/12/05 19:32:40 meinelte Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file iic/v1_01_c/src/xiic_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  High-level driver functions
00029 * are defined in xiic.h.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date        Changes
00035 * -----
00036 * 1.00b  jhl    05/07/02  First release
00037 * 1.01c  ecm    12/05/02  new rev
00038 * </pre>
00039 *
00040 *****/
00041
```

```

00042 #ifndef XIIC_L_H /* prevent circular inclusions */
00043 #define XIIC_L_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048
00049 /***** Constant Definitions *****/
00050
00051 #define XIIC_MSB_OFFSET          3
00052
00053 #define XIIC_REG_OFFSET 0x100 + XIIC_MSB_OFFSET
00054
00055 /*
00056  * Register offsets in bytes from RegisterBase. Three is added to the
00057  * base offset to access LSB (IBM style) of the word
00058  */
00059 #define XIIC_CR_REG_OFFSET      0x00+XIIC_REG_OFFSET /* Control Register */
00060 #define XIIC_SR_REG_OFFSET      0x04+XIIC_REG_OFFSET /* Status Register */
00061 #define XIIC_DTR_REG_OFFSET     0x08+XIIC_REG_OFFSET /* Data Tx Register */
00062 #define XIIC_DRR_REG_OFFSET     0x0C+XIIC_REG_OFFSET /* Data Rx Register */
00063 #define XIIC_ADR_REG_OFFSET     0x10+XIIC_REG_OFFSET /* Address Register */
00064 #define XIIC_TFO_REG_OFFSET     0x14+XIIC_REG_OFFSET /* Tx FIFO Occupancy */
00065 #define XIIC_RFO_REG_OFFSET     0x18+XIIC_REG_OFFSET /* Rx FIFO Occupancy */
00066 #define XIIC_TBA_REG_OFFSET     0x1C+XIIC_REG_OFFSET /* 10 Bit Address reg */
00067 #define XIIC_RFD_REG_OFFSET     0x20+XIIC_REG_OFFSET /* Rx FIFO Depth reg */
00068
00069 /* Control Register masks */
00070
00071 #define XIIC_CR_ENABLE_DEVICE_MASK      0x01 /* Device enable = 1 */
00072 #define XIIC_CR_TX_FIFO_RESET_MASK     0x02 /* Transmit FIFO reset=1 */
00073 #define XIIC_CR_MSMS_MASK              0x04 /* Master starts Txing=1 */
00074 #define XIIC_CR_DIR_IS_TX_MASK         0x08 /* Dir of tx. Txing=1 */
00075 #define XIIC_CR_NO_ACK_MASK            0x10 /* Tx Ack. NO ack = 1 */
00076 #define XIIC_CR_REPEATED_START_MASK    0x20 /* Repeated start = 1 */
00077 #define XIIC_CR_GENERAL_CALL_MASK      0x40 /* Gen Call enabled = 1 */
00078
00079 /* Status Register masks */
00080
00081 #define XIIC_SR_GEN_CALL_MASK           0x01 /* 1=a mstr issued a GC */
00082 #define XIIC_SR_ADDR_AS_SLAVE_MASK     0x02 /* 1=when addr as slave */
00083 #define XIIC_SR_BUS_BUSY_MASK          0x04 /* 1 = bus is busy */
00084 #define XIIC_SR_MSTR_RDING_SLAVE_MASK  0x08 /* 1=Dir: mstr <-- slave */
00085 #define XIIC_SR_TX_FIFO_FULL_MASK      0x10 /* 1 = Tx FIFO full */
00086 #define XIIC_SR_RX_FIFO_FULL_MASK      0x20 /* 1 = Rx FIFO full */
00087 #define XIIC_SR_RX_FIFO_EMPTY_MASK     0x40 /* 1 = Rx FIFO empty */
00088
00089 /* IPIF Interrupt Status Register masks      Interrupt occurs when... */
00090
00091 #define XIIC_INTR_ARB_LOST_MASK        0x01 /* 1 = arbitration lost */
00092 #define XIIC_INTR_TX_ERROR_MASK        0x02 /* 1=Tx error/msg complete*/
00093 #define XIIC_INTR_TX_EMPTY_MASK        0x04 /* 1 = Tx FIFO/reg empty */

```

```

00094 #define XIIC_INTR_RX_FULL_MASK          0x08 /* 1=Rx FIFO/reg=OCY level*/
00095 #define XIIC_INTR_BNB_MASK              0x10 /* 1 = Bus not busy */
00096 #define XIIC_INTR_AAS_MASK              0x20 /* 1 = when addr as slave */
00097 #define XIIC_INTR_NAAS_MASK             0x40 /* 1 = not addr as slave */
00098 #define XIIC_INTR_TX_HALF_MASK         0x80 /* 1 = TX FIFO half empty */
00099
00100 /* IPIF Device Interrupt Register masks */
00101
00102 #define XIIC_IPIF_IIC_MASK                0x00000004UL /* 1=inter enabled */
00103 #define XIIC_IPIF_ERROR_MASK             0x00000001UL /* 1=inter enabled */
00104 #define XIIC_IPIF_INTER_ENABLE_MASK     (XIIC_IPIF_IIC_MASK | \
00105                                         XIIC_IPIF_ERROR_MASK)
00106
00107 #define XIIC_TX_ADDR_SENT                 0x00
00108 #define XIIC_TX_ADDR_MSTR_RECV_MASK     0x02
00109
00110 /* The following constants specify the depth of the FIFOs */
00111
00112 #define IIC_RX_FIFO_DEPTH                 16 /* Rx fifo capacity */
00113 #define IIC_TX_FIFO_DEPTH                 16 /* Tx fifo capacity */
00114
00115 /* The following constants specify groups of interrupts that are typically
00116 * enabled or disables at the same time
00117 */
00118 #define XIIC_TX_INTERRUPTS                \
00119     (XIIC_INTR_TX_ERROR_MASK | XIIC_INTR_TX_EMPTY_MASK | \
00120     XIIC_INTR_TX_HALF_MASK)
00121
00122 #define XIIC_TX_RX_INTERRUPTS (XIIC_INTR_RX_FULL_MASK | XIIC_TX_INTERRUPTS)
00123
00124 /* The following constants are used with the following macros to specify the
00125 * operation, a read or write operation.
00126 */
00127 #define XIIC_READ_OPERATION               1
00128 #define XIIC_WRITE_OPERATION              0
00129
00130 /* The following constants are used with the transmit FIFO fill function to
00131 * specify the role which the IIC device is acting as, a master or a slave.
00132 */
00133 #define XIIC_MASTER_ROLE                  1
00134 #define XIIC_SLAVE_ROLE                   0
00135
00136 /***** Type Definitions *****/
00137
00138
00139 /***** Macros (Inline Functions) Definitions *****/
00140
00141
00142 /***** Function Prototypes *****/
00143
00144 unsigned Xiic_Recv(Xuint32 BaseAddress, Xuint8 Address,
00145                   Xuint8 *BufferPtr, unsigned ByteCount);

```



```
00146
00147 unsigned XIic_Send(Xuint32 BaseAddress, Xuint8 Address,
00148                     Xuint8 *BufferPtr, unsigned ByteCount);
00149
00150 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

iic/v1_01_c/src/xiic_1.c File Reference

Detailed Description

This file contains low-level driver functions that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01b	jhl	5/13/02	First release
1.01b	jhl	10/14/02	Corrected bug in the receive function, the setup of the interrupt status mask was not being done in the loop such that a read would sometimes fail on the last byte because the transmit error which should have been ignored was being used. This would leave an extra byte in the FIFO and the bus throttled such that the next operation would also fail. Also updated the receive function to not disable the device after the last byte until after the bus transitions to not busy which is more consistent with the expected behavior.
1.01c	ecm	12/05/02	new rev

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"  
#include "xiic_1.h"
```

Functions

unsigned **XIic_Recv** (**Xuint32** BaseAddress, **Xuint8** Address, **Xuint8** *BufferPtr, unsigned ByteCount)

unsigned **XIic_Send** (**Xuint32** BaseAddress, **Xuint8** Address, **Xuint8** *BufferPtr, unsigned ByteCount)

Function Documentation

```
unsigned XIic_Recv( Xuint32  BaseAddress,  
                  Xuint8  Address,  
                  Xuint8 * BufferPtr,  
                  unsigned  ByteCount  
                  )
```

Receive data as a master on the IIC bus. This function receives the data using polled I/O and blocks until the data has been received. It only supports 7 bit addressing and non-repeated start modes of operation. The user is responsible for ensuring the bus is not busy if multiple masters are present on the bus.

Parameters:

BaseAddress contains the base address of the IIC device.

Address contains the 7 bit IIC address of the device to send the specified data to.

BufferPtr points to the data to be sent.

ByteCount is the number of bytes to be sent.

Returns:

The number of bytes received.

Note:

None

```
unsigned XIic_Send( Xuint32  BaseAddress,  
                  Xuint8  Address,  
                  Xuint8 * BufferPtr,  
                  unsigned  ByteCount  
                  )
```

Send data as a master on the IIC bus. This function sends the data using polled I/O and blocks until the data has been sent. It only supports 7 bit addressing and non-repeated start modes of operation. The user is responsible for ensuring the bus is not busy if multiple masters are present on the bus.

Parameters:

BaseAddress contains the base address of the IIC device.

Address contains the 7 bit IIC address of the device to send the specified data to.

BufferPtr points to the data to be sent.

ByteCount is the number of bytes to be sent.

Returns:

The number of bytes sent.

Note:

None

Generated on 29 May 2003 for Xilinx Device Drivers

Xlic_Config Struct Reference

```
#include <xlic.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xboolean Has10BitAddr

Field Documentation

Xuint32 Xlic_Config::BaseAddress

Device base address

Xuint16 Xlic_Config::DeviceId

Unique ID of device

Xboolean Xlic_Config::Has10BitAddr

does device have 10 bit address decoding

The documentation for this struct was generated from the following file:

- [iic/v1_01_c/src/xiic.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XlicStats Struct Reference

```
#include <xiic.h>
```

Detailed Description

Xlic statistics

Data Fields

Xuint8 ArbitrationLost

Xuint8 RepeatedStarts

Xuint8 BusBusy

Xuint8 RecvBytes

Xuint8 RecvInterrupts

Xuint8 SendBytes

Xuint8 SendInterrupts

Xuint8 TxErrors

Xuint8 IicInterrupts

Field Documentation

Xuint8 XIicStats::ArbitrationLost

Number of times arbitration was lost

Xuint8 XIicStats::BusBusy

Number of times bus busy status returned

Xuint8 XIicStats::IicInterrupts

Number of IIC (device) interrupts

Xuint8 XIicStats::RecvBytes

Number of bytes received

Xuint8 XIicStats::RecvInterrupts

Number of receive interrupts

Xuint8 XIicStats::RepeatedStarts

Number of repeated starts

Xuint8 XIicStats::SendBytes

Number of transmit bytes received

Xuint8 XIicStats::SendInterrupts

Number of transmit interrupts

Xuint8 XIicStats::TxErrors

Number of transmit errors (no ack)

The documentation for this struct was generated from the following file:

- `iic/v1_01_c/src/xiic.h`

iic/v1_01_c/src/xiic.c File Reference

Detailed Description

Contains required functions for the **XIic** component. See [xiic.h](#) for more information on the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev
1.01c	rmm	05/14/03	Fixed diab compiler warnings relating to asserts.

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"  
#include "xparameters.h"
```

Functions

XStatus XIic_Initialize (**XIic** *InstancePtr, **Xuint16** DeviceId)

XStatus XIic_Start (**XIic** *InstancePtr)

XStatus XIic_Stop (**XIic** *InstancePtr)

void **XIic_Reset** (**XIic** *InstancePtr)

XStatus XIic_SetAddress (**XIic** *InstancePtr, int AddressType, int Address)

Xuint16 XIic_GetAddress (**XIic** *InstancePtr, int AddressType)

Xboolean XIic_IsSlave (**XIic** *InstancePtr)

void **XIic_SetRecvHandler** (**XIic** *InstancePtr, void *CallBackRef, **XIic_Handler** FuncPtr)

void **XIic_SetSendHandler** (**XIic** *InstancePtr, void *CallBackRef, **XIic_Handler** FuncPtr)

void **XIic_SetStatusHandler** (**XIic** *InstancePtr, void *CallBackRef, **XIic_StatusHandler** FuncPtr)

XIic_Config * **XIic_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
Xuint16 XIic_GetAddress( XIic * InstancePtr,  
                        int AddressType  
                        )
```

This function gets the addresses for the IIC device driver. The addresses include the device address that the device responds to as a slave, or the slave address to communicate with on the bus. The address returned has the same format whether 7 or 10 bits.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

AddressType indicates which address, the address which this responds to on the IIC bus as a slave, or the slave address to communicate with when this device is a master. One of the following values must be contained in this argument.

<code>XII_ADDRESS_TO_SEND_TYPE</code>	slave being addressed as a master
<code>XII_ADDRESS_TO_RESPOND_TYPE</code>	slave address to respond to as a slave

If neither of the two valid arguments are used, the function returns the address of the slave device

Returns:

The address retrieved.

Note:

None.

```
XStatus XIic_Initialize( XIic * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initializes a specific **XIic** instance. The initialization entails:

- Check the device has an entry in the configuration table.
- Initialize the driver to allow access to the device registers and initialize other subcomponents necessary for the operation of the device.
- Default options to:
 - 7-bit slave addressing
 - Send messages as a slave device
 - Repeated start off
 - General call recognition disabled
- Clear messaging and error statistics

The **XIic_Start()** function must be called after this function before the device is ready to send and receive data on the IIC bus.

Before **XIic_Start()** is called, the interrupt control must connect the ISR routine to the interrupt handler. This is done by the user, and not **XIic_Start()** to allow the user to use an interrupt controller of their choice.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XIic** instance. Passing in a device id associates the generic **XIic** instance to a specific device, as chosen by the caller or application developer.

Returns:

- `XST_SUCCESS` when successful
- `XST_DEVICE_IS_STARTED` indicates the device is started (i.e. interrupts enabled and messaging is possible). Must stop before re-initialization is allowed.

Note:

None.

Xboolean XIic_IsSlave(XIic * InstancePtr)

A function to determine if the device is currently addressed as a slave

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

XTRUE if the device is addressed as slave, and XFALSE otherwise.

Note:

None.

XIic_Config* XIic_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table IicConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for

Returns:

A pointer to the configuration data of the device, or XNULL if no match is found.

Note:

None.

void XIic_Reset(XIic * InstancePtr)

Resets the IIC device. Reset must only be called after the driver has been initialized. The configuration after this reset is as follows:

- Repeated start is disabled
- General call is disabled

The upper layer software is responsible for initializing and re-configuring (if necessary) and restarting the IIC device after the reset.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

None.

Note:

None.

```

XStatus XIic_SetAddress( XIic * InstancePtr,
                        int   AddressType,
                        int   Address
                        )

```

This function sets the bus addresses. The addresses include the device address that the device responds to as a slave, or the slave address to communicate with on the bus. The IIC device hardware is built to allow either 7 or 10 bit slave addressing only at build time rather than at run time. When this device is a master, slave addressing can be selected at run time to match addressing modes for other bus devices.

Addresses are represented as hex values with no adjustment for the data direction bit as the software manages address bit placement. Example: For a 7 address written to the device of 1010 011X where X is the transfer direction (send/rcv), the address parameter for this function needs to be 01010011 or 0x53 where the correct bit allignment will be handled for 7 as well as 10 bit devices. This is especially important as the bit placement is not handled the same depending on which options are used such as repeated start.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

AddressType indicates which address is being modified; the address which this device responds to on the IIC bus as a slave, or the slave address to communicate with when this device is a master. One of the following values must be contained in this argument.

XII_ADDRESS_TO_SEND	Slave being addressed by a this master
XII_ADDRESS_TO_RESPOND	Address to respond to as a slave device

Address contains the address to be set; 7 bit or 10 bit address. A ten bit address must be within the range: 0 - 1023 and a 7 bit address must be within the range 0 - 127.

Returns:

XST_SUCCESS is returned if the address was successfully set, otherwise one of the following errors is returned.

- o XST_IIC_NO_10_BIT_ADDRESSING indicates only 7 bit addressing supported.
- o XST_INVALID_PARAM indicates an invalid parameter was specified.

Note:

Upper bits of 10-bit address is written only when current device is built as a ten bit device.

```

void XIic_SetRecvHandler( XIic * InstancePtr,
                          void * CallbackRef,
                          XIic_Handler FuncPtr
                          )

```

Sets the receive callback function, the receive handler, which the driver calls when it finishes receiving data. The number of bytes used to signal when the receive is complete is the number of bytes set in the XIic_Recv function.

The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context.

The number of bytes received is passed to the handler as an argument.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
void Xlic_SetSendHandler( Xlic *      InstancePtr,  
                        void *      CallbackRef,  
                        Xlic_Handler FuncPtr  
                        )
```

Sets the send callback function, the send handler, which the driver calls when it receives confirmation of sent data. The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context.

Parameters:

InstancePtr the pointer to the **Xlic** instance to be worked on.

CallbackRef the upper layer callback reference passed back when the callback function is invoked.

FuncPtr the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
void Xlic_SetStatusHandler( Xlic *      InstancePtr,  
                          void *      CallbackRef,  
                          Xlic_StatusHandler FuncPtr  
                          )
```

Sets the status callback function, the status handler, which the driver calls when it encounters conditions which are not data related. The handler executes in an interrupt context such that it must minimize the amount of processing performed such as transferring data to a thread context. The status events that can be returned are described in [xiic.h](#).

Parameters:

InstancePtr points to the **Xlic** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context ...

```
XStatus Xlic_Start( Xlic * InstancePtr)
```

This function starts the IIC device and driver by enabling the proper interrupts such that data may be sent and received on the IIC bus. This function must be called before the functions to send and receive data.

Before [XIic_Start\(\)](#) is called, the interrupt control must connect the ISR routine to the interrupt handler. This is done by the user, and not [XIic_Start\(\)](#) to allow the user to use an interrupt controller of their choice.

Start enables:

- IIC device
- Interrupts:
 - Addressed as slave to allow messages from another master
 - Arbitration Lost to detect Tx arbitration errors
 - Global IIC interrupt within the IPIF interface

Parameters:

InstancePtr is a pointer to the [XIic](#) instance to be worked on.

Returns:

XST_SUCCESS always

Note:

The device interrupt is connected to the interrupt controller, but no "messaging" interrupts are enabled. Addressed as Slave is enabled to reception of messages when this devices address is written to the bus. The correct messaging interrupts are enabled when sending or receiving via the [IicSend\(\)](#) and [IicRecv\(\)](#) functions. No action is required by the user to control any IIC interrupts as the driver completely manages all 8 interrupts. Start and Stop control the ability to use the device. Stopping the device completely stops all device interrupts from the processor.

XStatus XIic_Stop(XIic * InstancePtr)

This function stops the IIC device and driver such that data is no longer sent or received on the IIC bus. This function stops the device by disabling interrupts. This function only disables interrupts within the device such that the caller is responsible for disconnecting the interrupt handler of the device from the interrupt source and disabling interrupts at other levels.

Due to bus throttling that could hold the bus between messages when using repeated start option, stop will not occur when the device is actively sending or receiving data from the IIC bus or the bus is being throttled by this device, but instead return XST_IIC_BUS_BUSY.

Parameters:

InstancePtr is a pointer to the [XIic](#) instance to be worked on.

Returns:

- XST_SUCCESS indicates all IIC interrupts are disabled. No messages can be received or transmitted until [XIic_Start\(\)](#) is called.
- XST_IIC_BUS_BUSY indicates this device is currently engaged in message traffic and cannot be stopped.

Note:

None.

iic/v1_01_c/src/xiic_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xiic_i.h,v 1.2 2002/12/05 19:32:40 meinelte Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file iic/v1_01_c/src/xiic_i.h
00026 *
00027 * This header file contains internal identifiers, which are those shared
00028 * between XIic components.  The identifiers in this file are not intended for
00029 * use external to the driver.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date        Changes
00035 * -----
00036 * 1.01a  rfp    10/19/01  release
00037 * 1.01c  ecm    12/05/02  new rev
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XIIC_I_H /* prevent circular inclusions */
```

```

00043 #define XIIC_I_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048 #include "xstatus.h"
00049
00050 /***** Constant Definitions *****/
00051
00052
00053 /***** Type Definitions *****/
00054
00055
00056 /***** Macros (Inline Functions) Definitions *****/
00057
00058
00059 /*****
00060 *
00061 * This macro sends the first byte of the address for a 10 bit address during
00062 * both read and write operations. It takes care of the details to format the
00063 * address correctly.
00064 *
00065 * address = 1111_0xxD    xx = address MSBits
00066 *                       D = Tx direction = 0 = write
00067 *
00068 * @param    SlaveAddress contains the address of the slave to send to.
00069 *
00070 * @param    Operation indicates XIIC_READ_OPERATION or XIIC_WRITE_OPERATION
00071 *
00072 * @return
00073 *
00074 * None.
00075 *
00076 * @note
00077 *
00078 * Signature: void XIic_mSend10BitAddrByte1(Xuint16 SlaveAddress, Xuint8
Operation);
00079 *
00080 *****/
00081 #define XIic_mSend10BitAddrByte1(SlaveAddress, Operation) \
00082 { \
00083     Xuint8 LocalAddr = (Xuint8)((SlaveAddress) >> 7); \
00084     LocalAddr = (LocalAddr & 0xF6) | 0xF0 | (Operation); \
00085     XIo_Out8(InstancePtr->BaseAddress + XIIC_DTR_REG_OFFSET, LocalAddr); \
00086 }
00087
00088 /*****
00089 *
00090 * This macro sends the second byte of the address for a 10 bit address during
00091 * both read and write operations. It takes care of the details to format the
00092 * address correctly.
00093 *

```



```

00094 * @param SlaveAddress contains the address of the slave to send to.
00095 *
00096 * @return
00097 *
00098 * None.
00099 *
00100 * @note
00101 *
00102 * Signature: void XIic_mSend10BitAddrByte2(Xuint16 SlaveAddress,
00103 *                                           Xuint8 Operation);
00104 *
00105 *****/
00106 #define XIic_mSend10BitAddrByte2(SlaveAddress) \
00107     XIo_Out8(InstancePtr->BaseAddress + XIIC_DTR_REG_OFFSET, \
00108             (Xuint8)(SlaveAddress));
00109
00110 /*****
00111 *
00112 * This macro sends the address for a 7 bit address during both read and write
00113 * operations. It takes care of the details to format the address correctly.
00114 *
00115 * @param SlaveAddress contains the address of the slave to send to.
00116 *
00117 * @param Operation indicates XIIC_READ_OPERATION or XIIC_WRITE_OPERATION
00118 *
00119 * @return
00120 *
00121 * None.
00122 *
00123 * @note
00124 *
00125 * Signature: void XIic_mSend7BitAddr(Xuint16 SlaveAddress, Xuint8 Operation);
00126 *
00127 *****/
00128 #define XIic_mSend7BitAddr(SlaveAddress, Operation) \
00129 { \
00130     Xuint8 LocalAddr = (Xuint8)(SlaveAddress << 1); \
00131     LocalAddr = (LocalAddr & 0xFE) | (Operation); \
00132     XIo_Out8(InstancePtr->BaseAddress + XIIC_DTR_REG_OFFSET, LocalAddr); \
00133 }
00134
00135 /*****
00136 *
00137 * This macro disables the specified interrupts in the IPIF interrupt enable
00138 * register. It is non-destructive in that the register is read and only the
00139 * interrupts specified is changed.
00140 *
00141 * @param BaseAddress contains the IPIF registers base address.
00142 *
00143 * @param InterruptMask contains the interrupts to be disabled
00144 *
00145 * @return
00146 *

```

```

00147 * None.
00148 *
00149 * @note
00150 *
00151 * Signature: void XIic_mDisableIntr(Xuint32 BaseAddress,
00152 *                               Xuint32 InterruptMask);
00153 *
00154 *****/
00155 #define XIic_mDisableIntr(BaseAddress, InterruptMask) \
00156     XIIF_V123B_WRITE_IIER((BaseAddress), \
00157     XIIF_V123B_READ_IIER(BaseAddress) & ~(InterruptMask))
00158
00159 /*****
00160 *
00161 * This macro enables the specified interrupts in the IPIF interrupt enable
00162 * register. It is non-destructive in that the register is read and only the
00163 * interrupts specified is changed.
00164 *
00165 * @param BaseAddress contains the IPIF registers base address.
00166 *
00167 * @param InterruptMask contains the interrupts to be disabled
00168 *
00169 * @return
00170 *
00171 * None.
00172 *
00173 * @note
00174 *
00175 * Signature: void XIic_mEnableIntr(Xuint32 BaseAddress,
00176 *                               Xuint32 InterruptMask);
00177 *
00178 *****/
00179 #define XIic_mEnableIntr(BaseAddress, InterruptMask) \
00180     XIIF_V123B_WRITE_IIER((BaseAddress), \
00181     XIIF_V123B_READ_IIER(BaseAddress) | (InterruptMask))
00182
00183 /*****
00184 *
00185 * This macro clears the specified interrupt in the IPIF interrupt status
00186 * register. It is non-destructive in that the register is read and only the
00187 * interrupt specified is cleared. Clearing an interrupt acknowledges it.
00188 *
00189 * @param BaseAddress contains the IPIF registers base address.
00190 *
00191 * @param InterruptMask contains the interrupts to be disabled
00192 *
00193 * @return
00194 *
00195 * None.
00196 *
00197 * @note
00198 *

```

```

00199 * Signature: void XIic_mClearIntr(Xuint32 BaseAddress,
00200 *                               Xuint32 InterruptMask);
00201 *
00202 *****/
00203 #define XIic_mClearIntr(BaseAddress, InterruptMask) \
00204     XIIF_V123B_WRITE_IISR((BaseAddress), \
00205     XIIF_V123B_READ_IISR(BaseAddress) & (InterruptMask))
00206
00207 *****/
00208 *
00209 * This macro clears and enables the specified interrupt in the IPIF interrupt
00210 * status and enable registers. It is non-destructive in that the registers are
00211 * read and only the interrupt specified is modified.
00212 * Clearing an interrupt acknowledges it.
00213 *
00214 * @param BaseAddress contains the IPIF registers base address.
00215 *
00216 * @param InterruptMask contains the interrupts to be cleared and enabled
00217 *
00218 * @return
00219 *
00220 * None.
00221 *
00222 * @note
00223 *
00224 * Signature: void XIic_mClearEnableIntr(Xuint32 BaseAddress,
00225 *                                       Xuint32 InterruptMask);
00226 *
00227 *****/
00228 #define XIic_mClearEnableIntr(BaseAddress, InterruptMask) \
00229 { \
00230     XIIF_V123B_WRITE_IISR(BaseAddress, \
00231     (XIIF_V123B_READ_IISR(BaseAddress) & (InterruptMask))); \
00232     XIIF_V123B_WRITE_IIER(BaseAddress, \
00233     (XIIF_V123B_READ_IIER(BaseAddress) | (InterruptMask))); \
00234 }
00235
00236
00237 *****/
00238 *
00239 * This macro flushes the receive FIFO such that all bytes contained within it
00240 * are discarded.
00241 *
00242 * @param InstancePtr is a pointer to the IIC instance containing the FIFO
00243 * to be flushed.
00244 *
00245 * @return
00246 *
00247 * None.
00248 *
00249 * @note
00250 *
00251 * Signature: void XIic_mFlushRxFifo(XIic *InstancePtr);

```

```

00252 *
00253 *****/
00254 #define XIic_mFlushRxFifo(InstancePtr) \
00255 { \
00256     int LoopCnt; \
00257     Xuint8 Temp; \
00258     Xuint8 BytesToRead = XIo_In8(InstancePtr->BaseAddress + \
00259                                 XIIC_RFO_REG_OFFSET) + 1; \
00260     for(LoopCnt = 0; LoopCnt < BytesToRead; LoopCnt++) \
00261     { \
00262         Temp = XIo_In8(InstancePtr->BaseAddress + XIIC_DRR_REG_OFFSET); \
00263     } \
00264 }
00265
00266 *****/
00267 *
00268 * This macro flushes the transmit FIFO such that all bytes contained within it
00269 * are discarded.
00270 *
00271 * @param InstancePtr is a pointer to the IIC instance containing the FIFO
00272 * to be flushed.
00273 *
00274 * @return
00275 *
00276 * None.
00277 *
00278 * @note
00279 *
00280 * Signature: void XIic_mFlushTxFifo(XIic *InstancePtr);
00281 *
00282 *****/
00283 #define XIic_mFlushTxFifo(InstancePtr); \
00284 { \
00285     Xuint8 CntlReg = XIo_In8(InstancePtr->BaseAddress + \
00286                             XIIC_CR_REG_OFFSET); \
00287     XIo_Out8(InstancePtr->BaseAddress + XIIC_CR_REG_OFFSET, \
00288             CntlReg | XIIC_CR_TX_FIFO_RESET_MASK); \
00289     XIo_Out8(InstancePtr->BaseAddress + XIIC_CR_REG_OFFSET, CntlReg); \
00290 }
00291
00292 *****/
00293 *
00294 * This macro reads the next available received byte from the receive FIFO
00295 * and updates all the data structures to reflect it.
00296 *
00297 * @param InstancePtr is a pointer to the IIC instance to be operated on.
00298 *
00299 * @return
00300 *
00301 * None.
00302 *
00303 * @note

```

```

00304 *
00305 * Signature: void XIic_mReadRecvByte(XIic *InstancePtr);
00306 *
00307 *****/
00308 #define XIic_mReadRecvByte(InstancePtr) \
00309 { \
00310     *InstancePtr->RecvBufferPtr++ = \
00311         XIo_In8(InstancePtr->BaseAddress + XIIC_DRR_REG_OFFSET); \
00312     InstancePtr->RecvByteCount--; \
00313     InstancePtr->Stats.RecvBytes++; \
00314 }
00315
00316 *****/
00317 *
00318 * This macro writes the next byte to be sent to the transmit FIFO
00319 * and updates all the data structures to reflect it.
00320 *
00321 * @param InstancePtr is a pointer to the IIC instance to be operated on.
00322 *
00323 * @return
00324 *
00325 * None.
00326 *
00327 * @note
00328 *
00329 * Signature: void XIic_mWriteSendByte(XIic *InstancePtr);
00330 *
00331 *****/
00332 #define XIic_mWriteSendByte(InstancePtr) \
00333 { \
00334     XIo_Out8(InstancePtr->BaseAddress + XIIC_DTR_REG_OFFSET, \
00335         *InstancePtr->SendBufferPtr++); \
00336     InstancePtr->SendByteCount--; \
00337     InstancePtr->Stats.SendBytes++; \
00338 }
00339
00340 *****/
00341 *
00342 * This macro sets up the control register for a master receive operation.
00343 * A write is necessary if a 10 bit operation is being performed.
00344 *
00345 * @param InstancePtr is a pointer to the IIC instance to be operated on.
00346 *
00347 * @param ControlRegister contains the contents of the IIC device control
00348 * register
00349 *
00350 * @param ByteCount contains the number of bytes to be received for the
00351 * master receive operation
00352 *
00353 * @return
00354 *
00355 * None.
00356 *

```

```

00357 * @note
00358 *
00359 * Signature: void XIic_mSetControlRegister(XIic *InstancePtr,
00360 *                                         Xuint8 ControlRegister,
00361 *                                         int ByteCount);
00362 *
00363 *****/
00364 #define XIic_mSetControlRegister(InstancePtr, ControlRegister, ByteCount) \
00365 { \
00366     (ControlRegister) &= ~(XIIC_CR_NO_ACK_MASK | XIIC_CR_DIR_IS_TX_MASK); \
00367     if (InstancePtr->Options & XII_SEND_10_BIT_OPTION) \
00368     { \
00369         (ControlRegister) |= XIIC_CR_DIR_IS_TX_MASK; \
00370     } \
00371     else \
00372     { \
00373         if ((ByteCount) == 1) \
00374         { \
00375             (ControlRegister) |= XIIC_CR_NO_ACK_MASK; \
00376         } \
00377     } \
00378 }
00379
00380 /*****
00381 *
00382 * This macro enters a critical region by disabling the global interrupt bit
00383 * in the IPIF.
00384 *
00385 * @param    BaseAddress contains the IPIF registers base address.
00386 *
00387 * @return
00388 *
00389 * None.
00390 *
00391 * @note
00392 *
00393 * Signature: void XIic_mEnterCriticalRegion(Xuint32 BaseAddress)
00394 *
00395 *****/
00396 #define XIic_mEnterCriticalRegion(BaseAddress) \
00397     XIIF_V123B_GINTR_DISABLE(BaseAddress)
00398
00399 /*****
00400 *
00401 * This macro exits a critical region by enabling the global interrupt bit
00402 * in the IPIF.
00403 *
00404 * @param    BaseAddress contains the IPIF registers base address.
00405 *
00406 * @return
00407 *
00408 * None.

```

```

00409 *
00410 * @note
00411 *
00412 * Signature: void XIic_mExitCriticalRegion(Xuint32 BaseAddress)
00413 *
00414 *****/
00415 #define XIic_mExitCriticalRegion(BaseAddress) \
00416     XIIF_V123B_GINTR_ENABLE(BaseAddress)
00417
00418 /*****
00419 *
00420 * This macro clears the statistics of an instance such that it can be common
00421 * such that some parts of the driver may be optional.
00422 *
00423 * @param InstancePtr is a pointer to the IIC instance to be operated on.
00424 *
00425 * @return
00426 *
00427 * None.
00428 *
00429 * @note
00430 *
00431 * Signature: void XIIC_CLEAR_STATS(XIic *InstancePtr)
00432 *
00433 *****/
00434 #define XIIC_CLEAR_STATS(InstancePtr) \
00435 { \
00436     Xuint8 NumBytes; \
00437     Xuint8 *DestPtr; \
00438 \
00439     DestPtr = (Xuint8 *)&InstancePtr->Stats; \
00440     for (NumBytes = 0; NumBytes < sizeof(XIicStats); NumBytes++) \
00441     { \
00442         *DestPtr++ = 0; \
00443     } \
00444 }
00445
00446 /***** Function Prototypes *****/
00447
00448 extern XIic_Config XIic_ConfigTable[];
00449
00450 /* The following variables are shared across files of the driver and
00451 * are function pointers that are necessary to break dependencies allowing
00452 * optional parts of the driver to be used without condition compilation
00453 */
00454 extern void (*XIic_AddrAsSlaveFuncPtr)(XIic *InstancePtr);
00455 extern void (*XIic_NotAddrAsSlaveFuncPtr)(XIic *InstancePtr);
00456 extern void (*XIic_RecvSlaveFuncPtr)(XIic *InstancePtr);
00457 extern void (*XIic_SendSlaveFuncPtr)(XIic *InstancePtr);
00458 extern void (*XIic_RecvMasterFuncPtr)(XIic *InstancePtr);
00459 extern void (*XIic_SendMasterFuncPtr)(XIic *InstancePtr);
00460 extern void (*XIic_ArbLostFuncPtr)(XIic *InstancePtr);

```

```
00461 extern void (*XIic_BusNotBusyFuncPtr)(XIic *InstancePtr);
00462
00463 void XIic_TransmitFifoFill(XIic *InstancePtr, int Role);
00464
00465 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

iic/v1_01_c/src/xiic_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between **XIic** components. The identifiers in this file are not intended for use external to the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

[Go to the source code of this file.](#)

Variables

[XIic_Config XIic_ConfigTable \[\]](#)

Variable Documentation

[XIic_Config XIic_ConfigTable\[\]\(\)](#)

The IIC configuration table, sized by the number of instances defined in [xparameters.h](#).

Generated on 29 May 2003 for Xilinx Device Drivers

iic/v1_01_c/src/xiic_intr.c File Reference

Detailed Description

Contains interrupt functions of the **XIic** driver. This file is required for the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev
1.01c	rmm	05/14/03	Fixed diab compiler warnings relating to asserts.

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

void **XIic_InterruptHandler** (void *InstancePtr)

Function Documentation

```
void XIic_InterruptHandler( void * InstancePtr)
```

This function is the interrupt handler for the **XIic** driver. This function should be connected to the interrupt system.

Only one interrupt source is handled for each interrupt allowing higher priority system interrupts quicker response time.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

None.

iic/v1_01_c/src/xiic_master.c File Reference

Detailed Description

Contains master functions for the **XIic** component. This file is necessary to send or receive as a master on the IIC bus.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	---	-----	-----
1.01b	jhl	3/27/02	Repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

XStatus XIic_MasterSend (**XIic** *InstancePtr, **Xuint8** *TxMsgPtr, int ByteCount)

XStatus XIic_MasterRecv (**XIic** *InstancePtr, **Xuint8** *RxMsgPtr, int ByteCount)

Function Documentation

```
XStatus XIic_MasterRecv( XIic * InstancePtr,  
                        Xuint8 * RxMsgPtr,  
                        int ByteCount  
                        )
```

This function receives data as a master from a slave device on the IIC bus. If the bus is busy, it will indicate so and then enable an interrupt such that the status handler will be called when the bus is no longer busy. The slave address which has been set with the [XIic_SetAddress\(\)](#) function is the address from which data is received. Receiving data on the bus performs a read operation.

Parameters:

InstancePtr is a pointer to the Iic instance to be worked on.

RxMsgPtr is a pointer to the data to be transmitted

ByteCount is the number of message bytes to be sent

Returns:

- XST_SUCCESS indicates the message reception processes has been initiated.
- XST_IIC_BUS_BUSY indicates the bus was in use and that the BusNotBusy interrupt is enabled which will update the EventStatus when the bus is no longer busy.
- XST_IIC_GENERAL_CALL_ADDRESS indicates the slave address is set to the the general call address. This is not allowed for Master receive mode.

```
XStatus XIic_MasterSend( XIic * InstancePtr,  
                        Xuint8 * TxMsgPtr,  
                        int ByteCount  
                        )
```

This function sends data as a master on the IIC bus. If the bus is busy, it will indicate so and then enable an interrupt such that the status handler will be called when the bus is no longer busy. The slave address which has been set with the [XIic_SetAddress\(\)](#) function is the address to which the specific data is sent. Sending data on the bus performs a write operation.

Parameters:

InstancePtr points to the Iic instance to be worked on.

TxMsgPtr points to the data to be transmitted

ByteCount is the number of message bytes to be sent

Returns:

- XST_SUCCESS indicates the message transmission has been initiated.
- XST_IIC_BUS_BUSY indicates the bus was in use and that the BusNotBusy interrupt

is enabled which will update the EventStatus when the bus is no longer busy.

Note:

None

Generated on 29 May 2003 for Xilinx Device Drivers

iic/v1_01_c/src/xiic_stats.c File Reference

Detailed Description

Contains statistics functions for the **XIic** component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01b	jhl	3/26/02	repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

```
void XIic_GetStats (XIic *InstancePtr, XIicStats *StatsPtr)  
void XIic_ClearStats (XIic *InstancePtr)
```

Function Documentation


```
void Xlic_ClearStats( Xlic * InstancePtr)
```

Clears the statistics for the IIC device by zeroing all counts.

Parameters:

InstancePtr is a pointer to the **Xlic** instance to be worked on.

Returns:

None.

Note:

None.

```
void Xlic_GetStats( Xlic * InstancePtr,  
                  XlicStats * StatsPtr  
                  )
```

Gets a copy of the statistics for an IIC device.

Parameters:

InstancePtr is a pointer to the **Xlic** instance to be worked on.

StatsPtr is a pointer to a **XlicStats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

iic/v1_01_c/src/xiic_selftest.c File Reference

Detailed Description

Contains selftest functions for the **XIic** component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.01b	jhl	3/26/02	repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

XStatus XIic_SelfTest (XIic *InstancePtr)

Function Documentation

XStatus XIic_SelfTest(XIic * InstancePtr)

Runs a limited self-test on the driver/device. The self-test is destructive in that a reset of the device is performed in order to check the reset values of the registers and to get the device into a known state. There is no loopback capabilities for the device such that this test does not send or receive data.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

XST_SUCCESS	No errors found
XST_IIC_STAND_REG_ERROR	One or more IIC regular registers did not zero on reset or read back correctly based on what was written to it
XST_IIC_TX_FIFO_REG_ERROR	One or more IIC parametrizable TX FIFO registers did not zero on reset or read back correctly based on what was written to it
XST_IIC_RX_FIFO_REG_ERROR	One or more IIC parametrizable RX FIFO registers did not zero on reset or read back correctly based on what was written to it
XST_IIC_STAND_REG_RESET_ERROR	A non parameterizable reg value after reset not valid
XST_IIC_TX_FIFO_REG_RESET_ERROR	Tx fifo, included in design, value after reset not valid
XST_IIC_RX_FIFO_REG_RESET_ERROR	Rx fifo, included in design, value after reset not valid
XST_IIC_TBA_REG_RESET_ERROR	10 bit addr, incl in design, value after reset not valid
XST_IIC_CR_READBACK_ERROR	Read of the control register didn't return value written
XST_IIC_DTR_READBACK_ERROR	Read of the data Tx reg didn't return value written
XST_IIC_DRR_READBACK_ERROR	Read of the data Receive reg didn't return value written
XST_IIC_ADR_READBACK_ERROR	Read of the data Tx reg didn't return value written
XST_IIC_TBA_READBACK_ERROR	Read of the 10 bit addr reg didn't return written value

Note:

Only the registers that have be included into the hardware design are tested, such as, 10-bit vs 7-bit addressing.

iic/v1_01_c/src/xiic_options.c File Reference

Detailed Description

Contains options functions for the **XIic** component. This file is not required unless the functions in this file are called.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	---	-----	-----
1.01b	jhl	3/26/02	repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

```
void XIic_SetOptions (XIic *InstancePtr, Xuint32 NewOptions)  
Xuint32 XIic_GetOptions (XIic *InstancePtr)
```

Function Documentation

Xuint32 `XIic_GetOptions(XIic * InstancePtr)`

This function gets the current options for the IIC device. Options control the how the device behaves on the IIC bus. See `SetOptions` for more information on options.

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

Returns:

The options of the IIC device. See [xiic.h](#) for a list of available options.

Note:

Options enabled will have a 1 in its appropriate bit position.

```
void XIic_SetOptions( XIic * InstancePtr,  
                    Xuint32 NewOptions  
                    )
```

This function sets the options for the IIC device driver. The options control how the device behaves relative to the IIC bus. If an option applies to how messages are sent or received on the IIC bus, it must be set prior to calling functions which send or receive data.

To set multiple options, the values must be ORed together. To not change existing options, read/modify/write with the current options using [XIic_GetOptions\(\)](#).

USAGE EXAMPLE:

Read/modify/write to enable repeated start:

```
Xuint8 Options;  
Options = XIic_GetOptions(&Iic);  
XIic_SetOptions(&Iic, Options | XII_REPEATED_START_OPTION);
```

Disabling General Call:

```
Options = XIic_GetOptions(&Iic);  
XIic_SetOptions(&Iic, Options &= ~XII_GENERAL_CALL_OPTION);
```

Parameters:

InstancePtr is a pointer to the **XIic** instance to be worked on.

NewOptions are the options to be set. See **xiic.h** for a list of the available options.

Returns:

None.

Note:

Sending or receiving messages with repeated start enabled, and then disabling repeated start, will not take effect until another master transaction is completed. i.e. After using repeated start, the bus will continue to be throttled after repeated start is disabled until a master transaction occurs allowing the IIC to release the bus.

Options enabled will have a 1 in its appropriate bit position.

iic/v1_01_c/src/xiic_multi_master.c File Reference

Detailed Description

Contains multi-master functions for the **XIic** component. This file is necessary if multiple masters are on the IIC bus such that arbitration can be lost or the bus can be busy.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	---	-----	-----
1.01b	jhl	3/27/02	Repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

Functions

```
void XIic_MultiMasterInclude ()
```

Function Documentation

```
void XIic_MultiMasterInclude( void )
```

This function includes multi-master code such that multi-master events are handled properly. Multi-master events include a loss of arbitration and the bus transitioning from busy to not busy. This function allows the multi-master processing to be optional. This function must be called prior to allowing any multi-master events to occur, such as after the driver is initialized.

Note:

None

iic/v1_01_c/src/xiic_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of IIC devices in the system. Each IIC device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.01a	rfp	10/19/01	release
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xparameters.h"
```

Variables

[XIic_Config XIic_ConfigTable](#) [XPAR_XIIC_NUM_INSTANCES]

Variable Documentation

[XIic_Config XIic_ConfigTable](#)[XPAR_XIIC_NUM_INSTANCES]

The IIC configuration table, sized by the number of instances defined in [xparameters.h](#).

intc/v1_00_b/src/xintc.h File Reference

Detailed Description

The Xilinx interrupt controller driver component. This component supports the Xilinx interrupt controller. A more detailed description of the API for the component can be found in the [xintc.c](#) file.

The interrupt controller driver uses the idea of priority for the various handlers. Priority is an integer within the range of 0 and 31 inclusive with 0 being the highest priority interrupt source.

The Xilinx interrupt controller supports the following features:

- specific individual interrupt enabling/disabling
- specific individual interrupt acknowledging
- attaching specific callback function to handle interrupt source
- master enable/disable
- single callback per interrupt or all pending interrupts handled for each interrupt of the processor

The acknowledgement of the interrupt within the interrupt controller is selectable, either prior to the device's handler being called or after the handler is called. This is necessary to support interrupt signal inputs which are either edge or level signals. Edge driven interrupt signals require that the interrupt is acknowledged prior to the interrupt being serviced in order to prevent the loss of interrupts which are occurring extremely close together. A level driven interrupt input signal requires the interrupt to be acknowledged after servicing the interrupt to ensure that the interrupt only generates a single interrupt condition.

Details about connecting the interrupt handler of the driver are contained in the source file specific to interrupt processing, [xintc_intr.c](#).

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY :

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/16/01	First release

- 1.00a rpm 01/09/02 Removed the AckLocation argument from `XIntc_Connect()`.
This information is now internal in `xintc_g.c`.
- 1.00b jhl 02/13/02 Repartitioned the driver for smaller files
- 1.00b jhl 04/24/02 Made LookupConfig function global and relocated config data type

```
#include "xbasic_types.h"  
#include "xparameters.h"  
#include "xstatus.h"  
#include "xintc_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XIntc  
struct XIntc_Config
```

Configuration options

These options are used in `XIntc_SetOptions()` to configure the device.

```
#define XIN_SVC_SGL_ISR_OPTION  
#define XIN_SVC_ALL_ISRS_OPTION
```

Start modes

One of these values is passed to `XIntc_Start()` to start the device.

```
#define XIN_SIMULATION_MODE  
#define XIN_REAL_MODE
```

Functions

```
XStatus XIntc_Initialize (XIntc *InstancePtr, Xuint16 DeviceId)  
XStatus XIntc_Start (XIntc *InstancePtr, Xuint8 Mode)
```

```

void XIntc_Stop (XIntc *InstancePtr)
XStatus XIntc_Connect (XIntc *InstancePtr, Xuint8 Id, XInterruptHandler Handler, void
    *CallbackRef)
void XIntc_Disconnect (XIntc *InstancePtr, Xuint8 Id)
void XIntc_Enable (XIntc *InstancePtr, Xuint8 Id)
void XIntc_Disable (XIntc *InstancePtr, Xuint8 Id)
void XIntc_Acknowledge (XIntc *InstancePtr, Xuint8 Id)
XIntc_Config * XIntc_LookupConfig (Xuint16 DeviceId)
void XIntc_VoidInterruptHandler ()
void XIntc_InterruptHandler (XIntc *InstancePtr)
XStatus XIntc_SetOptions (XIntc *InstancePtr, Xuint32 Options)
Xuint32 XIntc_GetOptions (XIntc *InstancePtr)
XStatus XIntc_SelfTest (XIntc *InstancePtr)
XStatus XIntc_SimulateIntr (XIntc *InstancePtr, Xuint8 Id)

```

Define Documentation

#define XIN_REAL_MODE

Real mode, no simulation allowed, hardware interrupts recognized

#define XIN_SIMULATION_MODE

Simulation only mode, no hardware interrupts recognized

#define XIN_SVC_ALL_ISRS_OPTION

<code>XIN_SVC_SGL_ISR_OPTION</code>	Service the highest priority pending interrupt and then return.
<code>XIN_SVC_ALL_ISRS_OPTION</code>	Service all of the pending interrupts and then return.

#define XIN_SVC_SGL_ISR_OPTION

<code>XIN_SVC_SGL_ISR_OPTION</code>	Service the highest priority pending interrupt and then return.
<code>XIN_SVC_ALL_ISRS_OPTION</code>	Service all of the pending interrupts and then return.

Function Documentation

```
void XIntc_Acknowledge( XIntc * InstancePtr,  
                      Xuint8 Id  
                      )
```

Acknowledges the interrupt source provided as the argument *Id*. When the interrupt is acknowledged, it causes the interrupt controller to clear its interrupt condition.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Returns:

None.

Note:

None.

```
XStatus XIntc_Connect( XIntc * InstancePtr,  
                     Xuint8 Id,  
                     XInterruptHandler Handler,  
                     void * CallbackRef  
                     )
```

Makes the connection between the *Id* of the interrupt source and the associated handler that is to run when the interrupt is recognized. The argument provided in this call as the *Callbackref* is used as the argument for the handler when it is called.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Handler to the handler for that interrupt.

CallbackRef is the callback reference, usually the instance pointer of the connecting driver.

Returns:

- XST_SUCCESS if the handler was connected correctly.
- XST_INTC_CONNECT_ERROR if the handler is already in use. Must disconnect existing handler assignment prior to calling connect again.

Note:

None.

```
void XIntc_Disable( XIntc * InstancePtr,  
                  Xuint8 Id  
                  )
```

Disables the interrupt source provided as the argument *Id* such that the interrupt controller will not cause interrupts for the specified *Id*. The interrupt controller will continue to hold an interrupt condition for the *Id*, but will not cause an interrupt.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Returns:

None.

Note:

None.

```
void XIntc_Disconnect( XIntc * InstancePtr,  
                     Xuint8 Id  
                     )
```

Updates the interrupt table with the Null Handler and XNULL arguments at the location pointed at by the *Id*. This effectively disconnects that interrupt source from any handler. The interrupt is disabled also.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Returns:

None.

Note:

None.

```
void XIntc_Enable( XIntc * InstancePtr,  
                  Xuint8 Id  
                  )
```

Enables the interrupt source provided as the argument *Id*. Any pending interrupt condition for the specified *Id* will occur after this function is called.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Returns:

None.

Note:

None.

```
Xuint32 XIntc_GetOptions( XIntc * InstancePtr)
```

Return the currently set options.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

The currently set options. The options are described in [xintc.h](#).

Note:

None.

```
XStatus XIntc_Initialize( XIntc * InstancePtr,  
                          Xuint16 DeviceId  
                          )
```


Initialize a specific interrupt controller instance/driver. The initialization entails:

- Initialize fields of the **XIntc** structure
- Initial vector table with stub function calls
- All interrupt sources are disabled
- Interrupt output is disabled

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XIntc** instance. Passing in a device id associates the generic **XIntc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

```
void XIntc_InterruptHandler( XIntc * InstancePtr)
```

The interrupt handler for the driver. This function determines the pending interrupts and calls the appropriate handlers in the priority based order.

This specific function allows multiple interrupt controller instances to be handled. The user must connect this function to the interrupt system such that it is called whenever the devices which are connected to it cause an interrupt.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

None.

Note:

None.

```
XIntc_Config* XIntc_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique identifier for a device.

Returns:

A pointer to the **XIntc** configuration structure for the specified device, or XNULL if the device was not found.

Note:

None.

```
XStatus XIntc_SelfTest( XIntc * InstancePtr)
```

Run a self-test on the driver/device. This is a destructive test.

This involves forcing interrupts into the controller and verifying that they are recognized and can be acknowledged. This test will not succeed if the interrupt controller has been started in real mode such that interrupts cannot be forced.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

- XST_SUCCESS if self-test is successful.
- XST_INTC_FAIL_SELFTEST if the Interrupt controller fails the self-test. It will fail the self test if the device has previously been started in real mode.

Note:

None.

```
XStatus XIntc_SetOptions( XIntc * InstancePtr,  
                        Xuint32 Options  
                        )
```


Starts the interrupt controller by enabling the output from the controller to the processor. Interrupts may be generated by the interrupt controller after this function is called.

It is necessary for the caller to connect the interrupt handler of this component to the proper interrupt source.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Mode determines if software is allowed to simulate interrupts or real interrupts are allowed to occur. Note that these modes are mutually exclusive. The interrupt controller hardware resets in a mode that allows software to simulate interrupts until this mode is exited. It cannot be reentered once it has been exited. One of the following values should be used for the mode.

- XIN_SIMULATION_MODE enables simulation of interrupts only
- XIN_REAL_MODE enables hardware interrupts only

Returns:

- XST_SUCCESS if the device was started successfully
- XST_FAILURE if simulation mode was specified and it could not be set because real mode has already been entered.

Note:

Must be called after **XIntc** initialization is completed.

```
void XIntc_Stop( XIntc * InstancePtr)
```

Stops the interrupt controller by disabling the output from the controller so that no interrupts will be caused by the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

None.

Note:

None.

```
void XIntc_VoidInterruptHandler( )
```

Interrupt handler for the driver. This function determines the pending interrupts and calls the appropriate handlers in the priority based order.

This specific function does not support multiple interrupt controller instances to be handled. The user must connect this function to the interrupt system such that it is called whenever the devices which are connected to it cause an interrupt.

Returns:

None.

Note:

None.

intc/v1_00_b/src/xintc.c File Reference

Detailed Description

Contains required functions for the **XIntc** driver for the Xilinx Interrupt Controller. See [xintc.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/16/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files
1.00b	jhl	04/24/02	Made LookupConfig global and compressed ack before table in the configuration into a bit mask

```
#include "xbasic_types.h"  
#include "xintc.h"  
#include "xintc_l.h"  
#include "xintc_i.h"
```

Functions

XStatus XIntc_Initialize (**XIntc** *InstancePtr, **Xuint16** DeviceId)
XStatus XIntc_Start (**XIntc** *InstancePtr, **Xuint8** Mode)
void **XIntc_Stop** (**XIntc** *InstancePtr)
XStatus XIntc_Connect (**XIntc** *InstancePtr, **Xuint8** Id, **XInterruptHandler** Handler, void *CallbackRef)
void **XIntc_Disconnect** (**XIntc** *InstancePtr, **Xuint8** Id)
void **XIntc_Enable** (**XIntc** *InstancePtr, **Xuint8** Id)
void **XIntc_Disable** (**XIntc** *InstancePtr, **Xuint8** Id)
void **XIntc_Acknowledge** (**XIntc** *InstancePtr, **Xuint8** Id)

Function Documentation

```
void XIntc_Acknowledge( XIntc * InstancePtr,  
                      Xuint8 Id  
                      )
```

Acknowledges the interrupt source provided as the argument *Id*. When the interrupt is acknowledged, it causes the interrupt controller to clear its interrupt condition.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Returns:

None.

Note:

None.

```
XStatus XIntc_Connect( XIntc * InstancePtr,  
                      Xuint8 Id,  
                      XInterruptHandler Handler,  
                      void * CallbackRef  
                      )
```

Makes the connection between the *Id* of the interrupt source and the associated handler that is to run when the interrupt is recognized. The argument provided in this call as the *CallbackRef* is used as the argument for the handler when it is called.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Handler to the handler for that interrupt.

CallbackRef is the callback reference, usually the instance pointer of the connecting driver.

Returns:

- XST_SUCCESS if the handler was connected correctly.

- XST_INTC_CONNECT_ERROR if the handler is already in use. Must disconnect existing handler assignment prior to calling connect again.

Note:

None.

```
void XIntc_Disable( XIntc * InstancePtr,  
                  Xuint8 Id  
                  )
```

Disables the interrupt source provided as the argument *Id* such that the interrupt controller will not cause interrupts for the specified *Id*. The interrupt controller will continue to hold an interrupt condition for the *Id*, but will not cause an interrupt.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Returns:

None.

Note:

None.

```
void XIntc_Disconnect( XIntc * InstancePtr,  
                     Xuint8 Id  
                     )
```

Updates the interrupt table with the Null Handler and XNULL arguments at the location pointed at by the *Id*. This effectively disconnects that interrupt source from any handler. The interrupt is disabled also.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Returns:

None.

Note:

None.


```
void XIntc_Enable( XIntc * InstancePtr,  
                  Xuint8 Id  
                  )
```

Enables the interrupt source provided as the argument *Id*. Any pending interrupt condition for the specified *Id* will occur after this function is called.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id contains the ID of the interrupt source and should be in the range of 0 to XPAR_INTC_MAX_NUM_INTR_INPUTS - 1 with 0 being the highest priority interrupt.

Returns:

None.

Note:

None.

```
XStatus XIntc_Initialize( XIntc * InstancePtr,  
                          Xuint16 DeviceId  
                          )
```

Initialize a specific interrupt controller instance/driver. The initialization entails:

- Initialize fields of the **XIntc** structure
- Initial vector table with stub function calls
- All interrupt sources are disabled
- Interrupt output is disabled

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XIntc** instance. Passing in a device id associates the generic **XIntc** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if device configuration information was not found for a device with the supplied device ID.

Note:

None.

XIntc_Config* XIntc_LookupConfig(**Xuint16** *DeviceId*)

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique identifier for a device.

Returns:

A pointer to the **XIntc** configuration structure for the specified device, or XNULL if the device was not found.

Note:

None.

XStatus XIntc_Start(**XIntc *** *InstancePtr*, **Xuint8** *Mode*)

Starts the interrupt controller by enabling the output from the controller to the processor. Interrupts may be generated by the interrupt controller after this function is called.

It is necessary for the caller to connect the interrupt handler of this component to the proper interrupt source.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Mode determines if software is allowed to simulate interrupts or real interrupts are allowed to occur. Note that these modes are mutually exclusive. The interrupt controller hardware resets in a mode that allows software to simulate interrupts until this mode is exited. It cannot be reentered once it has been exited. One of the following values should be used for the mode.

- XIN_SIMULATION_MODE enables simulation of interrupts only
- XIN_REAL_MODE enables hardware interrupts only

Returns:

- XST_SUCCESS if the device was started successfully
- XST_FAILURE if simulation mode was specified and it could not be set because real mode has already been entered.

Note:

Must be called after **XIntc** initialization is completed.

```
void XIntc_Stop( XIntc * InstancePtr)
```

Stops the interrupt controller by disabling the output from the controller so that no interrupts will be caused by the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

None.

Note:

None.

XIntc Struct Reference

```
#include <xintc.h>
```

Detailed Description

The XIntc driver instance data. The user is required to allocate a variable of this type for every intc device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `intc/v1_00_b/src/xintc.h`
-

intc/v1_00_b/src/xintc.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xintc.h,v 1.9 2002/07/26 20:36:59 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file intc/v1_00_b/src/xintc.h
00026 *
00027 * The Xilinx interrupt controller driver component. This component supports the
00028 * Xilinx interrupt controller. A more detailed description of the API for the
00029 * component can be found in the xintc.c file.
00030 *
00031 * The interrupt controller driver uses the idea of priority for the various
00032 * handlers. Priority is an integer within the range of 0 and 31 inclusive with
00033 * 0 being the highest priority interrupt source.
00034 *
00035 * The Xilinx interrupt controller supports the following features:
00036 *
00037 * - specific individual interrupt enabling/disabling
00038 * - specific individual interrupt acknowledging
00039 * - attaching specific callback function to handle interrupt source
00040 * - master enable/disable
00041 * - single callback per interrupt or all pending interrupts handled for
00042 * each interrupt of the processor
```

```

00043 *
00044 * The acknowledgement of the interrupt within the interrupt controller is
00045 * selectable, either prior to the device's handler being called or after
00046 * the handler is called. This is necessary to support interrupt signal inputs
00047 * which are either edge or level signals. Edge driven interrupt signals
00048 * require that the interrupt is acknowledged prior to the interrupt being
00049 * serviced in order to prevent the loss of interrupts which are occurring
00050 * extremely close together. A level driven interrupt input signal requires
00051 * the interrupt to be acknowledged after servicing the interrupt to ensure that
00052 * the interrupt only generates a single interrupt condition.
00053 *
00054 * Details about connecting the interrupt handler of the driver are contained
00055 * in the source file specific to interrupt processing, xintc_intr.c.
00056 *
00057 * This driver is intended to be RTOS and processor independent. It works with
00058 * physical addresses only. Any needs for dynamic memory management, threads
00059 * or thread mutual exclusion, virtual memory, or cache control must be
00060 * satisfied by the layer above this driver.
00061 *
00062 * <pre>
00063 * MODIFICATION HISTORY:
00064 *
00065 * Ver    Who    Date      Changes
00066 * -----
00067 * 1.00a  ecm    08/16/01  First release
00068 * 1.00a  rpm    01/09/02  Removed the AckLocation argument from XIntc_Connect().
00069 *                This information is now internal in xintc_g.c.
00070 * 1.00b  jhl    02/13/02  Repartitioned the driver for smaller files
00071 * 1.00b  jhl    04/24/02  Made LookupConfig function global and relocated config
00072 *                data type
00073 * </pre>
00074 *
00075 *****/
00076
00077 #ifndef XINTC_H /* prevent circular inclusions */
00078 #define XINTC_H /* by using protection macros */
00079
00080 /***** Include Files *****/
00081
00082 #include "xbasic_types.h"
00083 #include "xparameters.h"
00084 #include "xstatus.h"
00085 #include "xintc_l.h"
00086
00087 /***** Constant Definitions *****/
00088
00089 /**
00090  * @name Configuration options
00091  * These options are used in XIntc_SetOptions() to configure the device.
00092  * @{
00093  */
00094 /**

```

```

00095 * <pre>
00096 * XIN_SVC_SGL_ISR_OPTION      Service the highest priority pending interrupt
00097 *                               and then return.
00098 * XIN_SVC_ALL_ISRS_OPTION     Service all of the pending interrupts and then
00099 *                               return.
00100 * </pre>
00101 */
00102 #define XIN_SVC_SGL_ISR_OPTION  1UL
00103 #define XIN_SVC_ALL_ISRS_OPTION 2UL
00104 /*@}*/
00105
00106 /**
00107  * @name Start modes
00108  * One of these values is passed to XIntc_Start() to start the device.
00109  * @{
00110  */
00111 /** Simulation only mode, no hardware interrupts recognized */
00112 #define XIN_SIMULATION_MODE      0
00113 /** Real mode, no simulation allowed, hardware interrupts recognized */
00114 #define XIN_REAL_MODE            1
00115 /*@}*/
00116
00117 /*
00118  * A constant to allow each table in the component to be sized properly
00119  * since the max interrupt id is a zero based number
00120  */
00121 #define XIN_TABLE_SIZE          (XPAR_INTC_MAX_ID + 1)
00122
00123 /***** Type Definitions *****/
00124
00125 /**
00126  * This typedef contains configuration information for the device.
00127  */
00128 typedef struct
00129 {
00130     Xuint16 DeviceId;           /**< Unique ID  of device */
00131     Xuint32 BaseAddress;        /**< Register base address */
00132     Xuint32 AckBeforeService;   /**< Ack location per interrupt */
00133 } XIntc_Config;
00134
00135 /**
00136  * The XIntc driver instance data. The user is required to allocate a
00137  * variable of this type for every intc device in the system. A pointer
00138  * to a variable of this type is then passed to the driver API functions.
00139  */
00140 typedef struct
00141 {
00142     Xuint32 UnhandledInterrupts; /* Intc Statistics */
00143     Xuint32 BaseAddress;         /* Base address of registers */
00144     Xuint32 IsReady;            /* Device is initialized and ready */
00145     Xuint32 IsStarted;          /* Device has been started */

```

```

00146     Xuint32 Options;                /* Device options */
00147
00148     /* the following instance variable controls when each interrupt
00149     * is acknowledged, before servicing it (edge) or after (level)
00150     * each bit controls an interrupt id with a 1 indicating to ack
00151     * the interrupt before service and a 0 indicating ack after
00152     * the bit number correlates to the interrupt id (bit 0 = lsb)
00153     */
00154     Xuint32 AckBeforeService;
00155
00156     /* the following data is a table of handlers to be called when an
00157     * interrupt signal is active
00158     */
00159     XIntc_VectorTableEntry HandlerTable[XPAR_INTC_MAX_NUM_INTR_INPUTS];
00160
00161 } XIntc;
00162
00163 /***** Macros (Inline Functions) Definitions *****/
00164
00165 /***** Function Prototypes *****/
00166
00167 /*
00168  * Required functions in xintc.c
00169  */
00170
00171 XStatus XIntc_Initialize(XIntc *InstancePtr, Xuint16 DeviceId);
00172
00173 XStatus XIntc_Start(XIntc *InstancePtr, Xuint8 Mode);
00174 void XIntc_Stop(XIntc *InstancePtr);
00175
00176 XStatus XIntc_Connect(XIntc *InstancePtr, Xuint8 Id,
00177                      XInterruptHandler Handler, void *CallBackRef);
00178 void XIntc_Disconnect(XIntc *InstancePtr, Xuint8 Id);
00179
00180 void XIntc_Enable(XIntc *InstancePtr, Xuint8 Id);
00181 void XIntc_Disable(XIntc *InstancePtr, Xuint8 Id);
00182
00183 void XIntc_Acknowledge(XIntc *InstancePtr, Xuint8 Id);
00184
00185 XIntc_Config *XIntc_LookupConfig(Xuint16 DeviceId);
00186
00187 /*
00188  * Interrupt functions in xintr_intr.c
00189  */
00190 void XIntc_VoidInterruptHandler();
00191 void XIntc_InterruptHandler(XIntc *InstancePtr);
00192
00193 /*
00194  * Options functions in xintc_options.c
00195  */
00196 XStatus XIntc_SetOptions(XIntc *InstancePtr, Xuint32 Options);

```



```
00197 Xuint32 XIntc_GetOptions(XIntc *InstancePtr);
00198
00199 /*
00200  * Self-test functions in xintc_selftest.c
00201  */
00202 XStatus XIntc_SelfTest(XIntc *InstancePtr);
00203 XStatus XIntc_SimulateIntr(XIntc *InstancePtr, Xuint8 Id);
00204
00205 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

intc/v1_00_b/src/xintc_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xintc.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xparameters.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XIntc_VectorTableEntry
```

Defines

```
#define XIntc_mMasterEnable(BaseAddress)
```

```
#define XIntc_mMasterDisable(BaseAddress)
#define XIntc_mEnableIntr(BaseAddress, EnableMask)
#define XIntc_mDisableIntr(BaseAddress, DisableMask)
#define XIntc_mAckIntr(BaseAddress, AckMask)
#define XIntc_mGetIntrStatus(BaseAddress)
```

Functions

```
void XIntc_DefaultHandler (void *Input)
void XIntc_LowLevelInterruptHandler (void)
```

Define Documentation

```
#define XIntc_mAckIntr( BaseAddress,  
                          AckMask     )
```

Acknowledge specific interrupt(s) in the interrupt controller.

Parameters:

BaseAddress is the base address of the device

AckMask is the 32-bit value to write to the acknowledge register. Each bit of the mask corresponds to an interrupt input signal that is connected to the interrupt controller (INT0 = LSB). Only the bits which are set in the mask will acknowledge interrupts.

Returns:

None.

```
#define XIntc_mDisableIntr( BaseAddress,  
                          DisableMask )
```

Disable specific interrupt(s) in the interrupt controller.

Parameters:

BaseAddress is the base address of the device

DisableMask is the 32-bit value to write to the enable register. Each bit of the mask corresponds to an interrupt input signal that is connected to the interrupt controller (INT0 = LSB). Only the bits which are set in the mask will disable interrupts.

Returns:

None.

```
#define XIntc_mEnableIntr( BaseAddress,  
                          EnableMask )
```

Enable specific interrupt(s) in the interrupt controller.

Parameters:

BaseAddress is the base address of the device

EnableMask is the 32-bit value to write to the enable register. Each bit of the mask corresponds to an interrupt input signal that is connected to the interrupt controller (INT0 = LSB). Only the bits which are set in the mask will enable interrupts.

Returns:

None.

```
#define XIntc_mGetIntrStatus( BaseAddress )
```

Get the interrupt status from the interrupt controller which indicates which interrupts are active and enabled.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit contents of the interrupt status register. Each bit corresponds to an interrupt input signal that is connected to the interrupt controller (INT0 = LSB). Bits which are set indicate an active interrupt which is also enabled.

```
#define XIntc_mMasterDisable( BaseAddress )
```

Disable all interrupts in the Master Enable register of the interrupt controller.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

```
#define XIntc_mMasterEnable( BaseAddress )
```

Enable all interrupts in the Master Enable register of the interrupt controller. The interrupt controller defaults to all interrupts disabled from reset such that this macro must be used to enable interrupts.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

Function Documentation

```
void XIntc_DefaultHandler( void * UnusedInput )
```

This function is an default interrupt handler for the low level driver of the interrupt controller. It allows the interrupt vector table to be initialized to this function so that unexpected interrupts don't result in a system crash.

Parameters:

UnusedInput is an unused input that is necessary for this function to have the signature of an input handler.

Returns:

None.

Note:

None.

void XIntc_LowLevelInterruptHandler(void)

This function is an interrupt handler for the low level driver of the interrupt controller. It must be connected to the interrupt source such that is called when an interrupt of the interrupt controller is active. It will resolve which interrupts are active and enabled and call the appropriate interrupt handler. It acknowledges the interrupt after it has been serviced by the interrupt handler.

This function assumes that an interrupt vector table has been previously initialized by the user. It does not verify that entries in the table are valid before calling an interrupt handler.

Returns:

None.

Note:

The constants `XPAR_INTC_SINGLE_BASEADDR` & `XPAR_INTC_MAX_ID` must be setup for this to compile. Interrupt IDs range from 0 - 31 and correspond to the interrupt input signals for the interrupt controller. `XPAR_INTC_MAX_ID` specifies the highest numbered interrupt input signal that is used.

intc/v1_00_b/src/xintc_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xintc_l.h,v 1.5 2002/08/01 14:24:01 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file intc/v1_00_b/src/xintc_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  The user should refer to the
00029 * hardware device specification for more details of the device operation.
00030 * High-level driver functions are defined in xintc.h.
00031 *
00032 * <pre>
00033 * MODIFICATION HISTORY:
00034 *
00035 * Ver   Who   Date     Changes
00036 * ----  ---  -
00037 * 1.00b jhl   04/24/02 First release
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XINTC_L_H /* prevent circular inclusions */
```

```

00043 #define XINTC_L_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048 #include "xparameters.h"
00049 #include "xio.h"
00050
00051 /*      uncomment when the libgen functionality arrives
00052 #ifndef XPAR_XINTC_USE_DCR
00053 #error "XPAR_XINTC_USE_DCR not defined"
00054 #endif
00055 */
00056
00057 #if (XPAR_XINTC_USE_DCR != 0)
00058 #include "xio_dcr.h"
00059 #endif
00060
00061 /***** Constant Definitions *****/
00062
00063 /* define the offsets from the base address for all the registers of the
00064 * interrupt controller, some registers may be optional in the hardware device
00065 */
00066 #if (XPAR_XINTC_USE_DCR != 0)
00067
00068 #define XIN_ISR_OFFSET      0      /* Interrupt Status Register */
00069 #define XIN_IPR_OFFSET     1      /* Interrupt Pending Register */
00070 #define XIN_IER_OFFSET     2      /* Interrupt Enable Register */
00071 #define XIN_IAR_OFFSET     3      /* Interrupt Acknowledge Register */
00072 #define XIN_SIE_OFFSET     4      /* Set Interrupt Enable Register */
00073 #define XIN_CIE_OFFSET     5      /* Clear Interrupt Enable Register */
00074 #define XIN_IVR_OFFSET     6      /* Interrupt Vector Register */
00075 #define XIN_MER_OFFSET     7      /* Master Enable Register */
00076
00077 #else /*(XPAR_XINTC_USE_DCR != 0)*/
00078
00079 #define XIN_ISR_OFFSET      0      /* Interrupt Status Register */
00080 #define XIN_IPR_OFFSET     4      /* Interrupt Pending Register */
00081 #define XIN_IER_OFFSET     8      /* Interrupt Enable Register */
00082 #define XIN_IAR_OFFSET    12      /* Interrupt Acknowledge Register */
00083 #define XIN_SIE_OFFSET    16      /* Set Interrupt Enable Register */
00084 #define XIN_CIE_OFFSET    20      /* Clear Interrupt Enable Register */
00085 #define XIN_IVR_OFFSET    24      /* Interrupt Vector Register */
00086 #define XIN_MER_OFFSET    28      /* Master Enable Register */
00087
00088 #endif /*(XPAR_XINTC_USE_DCR != 0)*/
00089
00090 /* Bit definitions for the bits of the MER register */
00091
00092 #define XIN_INT_MASTER_ENABLE_MASK      0x1UL
00093 #define XIN_INT_HARDWARE_ENABLE_MASK   0x2UL /* once set cannot be cleared */
00094

```



```

00095 /***** Type Definitions *****/
00096
00097 /* The following data type defines each entry in an interrupt vector table,
00098  * the callback reference is the base address of the interrupting device
00099  * for the low level driver and an instance pointer for the high level driver
00100  */
00101 typedef struct
00102 {
00103     XInterruptHandler Handler;
00104     void *CallbackRef;
00105 } XIntc_VectorTableEntry;
00106
00107 /***** Macros (Inline Functions) Definitions *****/
00108
00109 /*
00110  * Define the appropriate I/O access method to memory mapped I/O or DCR.
00111  */
00112 #if (XPAR_XINTC_USE_DCR != 0)
00113
00114 #define XIntc_In32    XIo_DcrIn
00115 #define XIntc_Out32  XIo_DcrOut
00116
00117 #else
00118
00119 #define XIntc_In32    XIo_In32
00120 #define XIntc_Out32  XIo_Out32
00121
00122 #endif
00123
00124 /*****
00125  *
00126  * Low-level driver macros.  The list below provides signatures to help the
00127  * user use the macros.
00128  *
00129  * void XIntc_mMasterEnable(BaseAddress)
00130  * void XIntc_mMasterDisable(BaseAddress)
00131  *
00132  * void XIntc_mEnableIntr(BaseAddress, EnableMask)
00133  * void XIntc_mDisableIntr(BaseAddress, DisableMask)
00134  * void XIntc_mAckIntr(BaseAddress, AckMask)
00135  * Xuint32 XIntc_mGetIntrStatus(BaseAddress)
00136  *
00137  *****/
00138
00139 /*****
00140  **
00141  *
00142  * Enable all interrupts in the Master Enable register of the interrupt
00143  * controller.  The interrupt controller defaults to all interrupts disabled
00144  * from reset such that this macro must be used to enable interrupts.
00145  *
00146  * @param    BaseAddress is the base address of the device.

```

```

00147 *
00148 * @return    None.
00149 *
00150 *****/
00151 #define XIntc_mMasterEnable(BaseAddress) \
00152     XIntc_Out32((BaseAddress) + XIN_MER_OFFSET, \
00153                 XIN_INT_MASTER_ENABLE_MASK | XIN_INT_HARDWARE_ENABLE_MASK)
00154
00155 *****/
00156 /**
00157 *
00158 * Disable all interrupts in the Master Enable register of the interrupt
00159 * controller.
00160 *
00161 * @param    BaseAddress is the base address of the device.
00162 *
00163 * @return    None.
00164 *
00165 *****/
00166 #define XIntc_mMasterDisable(BaseAddress) \
00167     XIntc_Out32((BaseAddress) + XIN_MER_OFFSET, 0)
00168
00169 *****/
00170 /**
00171 *
00172 * Enable specific interrupt(s) in the interrupt controller.
00173 *
00174 * @param    BaseAddress is the base address of the device
00175 * @param    EnableMask is the 32-bit value to write to the enable register.
00176 *           Each bit of the mask corresponds to an interrupt input signal that
00177 *           is connected to the interrupt controller (INT0 = LSB). Only the
00178 *           bits which are set in the mask will enable interrupts.
00179 *
00180 * @return    None.
00181 *
00182 *****/
00183 #define XIntc_mEnableIntr(BaseAddress, EnableMask) \
00184     XIntc_Out32((BaseAddress) + XIN_IER_OFFSET, (EnableMask))
00185
00186 *****/
00187 /**
00188 *
00189 * Disable specific interrupt(s) in the interrupt controller.
00190 *
00191 * @param    BaseAddress is the base address of the device
00192 * @param    DisableMask is the 32-bit value to write to the enable register.
00193 *           Each bit of the mask corresponds to an interrupt input signal that
00194 *           is connected to the interrupt controller (INT0 = LSB). Only the
00195 *           bits which are set in the mask will disable interrupts.
00196 *
00197 * @return    None.
00198 *

```

```

00199 *****/
00200 #define XIntc_mDisableIntr(BaseAddress, DisableMask) \
00201     XIntc_Out32((BaseAddress) + XIN_IER_OFFSET, ~(DisableMask))
00202
00203 *****/
00204 /**
00205 *
00206 * Acknowledge specific interrupt(s) in the interrupt controller.
00207 *
00208 * @param   BaseAddress is the base address of the device
00209 * @param   AckMask is the 32-bit value to write to the acknowledge register.
00210 *          Each bit of the mask corresponds to an interrupt input signal that
00211 *          is connected to the interrupt controller (INT0 = LSB). Only the
00212 *          bits which are set in the mask will acknowledge interrupts.
00213 *
00214 * @return  None.
00215 *
00216 *****/
00217 #define XIntc_mAckIntr(BaseAddress, AckMask) \
00218     XIntc_Out32((BaseAddress) + XIN_IAR_OFFSET, (AckMask))
00219
00220 *****/
00221 /**
00222 *
00223 * Get the interrupt status from the interrupt controller which indicates
00224 * which interrupts are active and enabled.
00225 *
00226 * @param   BaseAddress is the base address of the device
00227 *
00228 * @return  The 32-bit contents of the interrupt status register. Each bit
00229 *          corresponds to an interrupt input signal that is connected to the
00230 *          interrupt controller (INT0 = LSB). Bits which are set indicate an
00231 *          active interrupt which is also enabled.
00232 *
00233 *****/
00234 #define XIntc_mGetIntrStatus(BaseAddress) \
00235     (XIntc_In32((BaseAddress) + XIN_ISR_OFFSET) & \
00236     XIntc_In32((BaseAddress) + XIN_IER_OFFSET))
00237
00238 *****/ Function Prototypes *****/
00239
00240 void XIntc_DefaultHandler(void *Input);
00241
00242 void XIntc_LowLevelInterruptHandler(void);
00243
00244 *****/ Variable Definitions *****/
00245
00246 extern XIntc_VectorTableEntry XIntc_InterruptVectorTable[];
00247
00248 extern Xuint32 XIntc_AckBeforeService;
00249
00250 #endif          /* end of protection macro */

```

Generated on 29 May 2003 for Xilinx Device Drivers

cpu_ppc405/v1_00_a/src/xio_dcr.h File Reference

Detailed Description

The DCR I/O access functions.

Note:

These access functions are specific to the PPC405 CPU. Changes might be necessary for other members of the IBM PPC Family.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	10/18/01	First release

```
#include "xbasic_types.h"
```

[Go to the source code of this file.](#)

Functions

```
void XIo_DcrOut (Xuint32 DcrRegister, Xuint32 Data)
```

```
Xuint32 XIo_DcrIn (Xuint32 DcrRegister)
```

Function Documentation

```
Xuint32 XIo_DcrIn( Xuint32 DcrRegister )
```

Reads value from specified register.

Parameters:

DcrRegister is the intended source DCR register

Returns:

Contents of the specified DCR register.

Note:

None.

```
void XIo_DcrOut( Xuint32 DcrRegister,  
                Xuint32 Data  
                )
```

Outputs value provided to specified register defined in the header file.

Parameters:

DcrRegister is the intended destination DCR register

Data is the value to be placed into the specified DCR register

Returns:

None.

Note:

None.

cpu_ppc405/v1_00_a/src/xio_dcr.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xio_dcr.h,v 1.7 2002/07/26 20:27:56 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file cpu_ppc405/v1_00_a/src/xio_dcr.h
00026 *
00027 * The DCR I/O access functions.
00028 *
00029 * @note
00030 *
00031 * These access functions are specific to the PPC405 CPU. Changes might be
00032 * necessary for other members of the IBM PPC Family.
00033 *
00034 * <pre>
00035 * MODIFICATION HISTORY:
00036 *
00037 * Ver   Who   Date       Changes
00038 * ----  ---  -
00039 * 1.00a ecm   10/18/01  First release
00040 * </pre>
00041 *
00042 * @internal
```

```

00043 *
00044 * The C functions which subsequently call into either the assembly code
00045 * or into the provided table of functions are required since the registers
00046 * assigned to the calling and return from functions are strictly defined
00047 * in the ABI and that definition is used in the low-level functions
00048 * directly. The use of macros is not recommended since the temporary
00049 * registers in the ABI are defined but there is no way to force
00050 * the compiler to use a specific register in a block of code.
00051 *
00052 *****/
00053
00054 #ifndef XDCRIO_H /* prevent circular inclusions */
00055 #define XDCRIO_H /* by using protection macros */
00056
00057 /***** Include Files *****/
00058 #include "xbasic_types.h"
00059
00060 /***** Constant Definitions *****/
00061
00062
00063 #define MAX_DCR_REGISTERS          1024
00064 #define MAX_DCR_REGISTER          MAX_DCR_REGISTERS - 1
00065 #define MIN_DCR_REGISTER          0
00066
00067 /***** Type Definitions *****/
00068
00069
00070 /***** Macros (Inline Functions) Definitions *****/
00071
00072
00073 /***** Function Prototypes *****/
00074
00075 /*
00076  * Access Functions
00077  */
00078 void XIo_DcrOut(Xuint32 DcrRegister, Xuint32 Data);
00079 Xuint32 XIo_DcrIn(Xuint32 DcrRegister);
00080
00081
00082 #endif /* end of protection macro */

```


cpu_ppc405/v1_00_a/src/xio_dcr.c File Reference

Detailed Description

The implementation of the XDcrIo interface. See [xio_dcr.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	10/18/01	First release
	ecm	01/29/03	Cleaned up the table and made it more readable. The DCR_REG_512 was 17 and the table used DCR_REG_32+15 which is actually the same as what DCR_REG_512 should have been,16. I removed the blocks using DCR_REG_32+15 and added blocks using DCR_REG_512+14 to make the table correct when DCR_REG_512 = 16. This is functionally exactly the same, just more readable hence no rev bump.

```
#include "xstatus.h"  
#include "xbasic_types.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Data Structures

struct **DcrFunctionTableEntryTag**

Functions

void **XIo_DcrOut** (**Xuint32** DcrRegister, **Xuint32** Data)

Xuint32 **XIo_DcrIn** (**Xuint32** DcrRegister)

Function Documentation

Xuint32 **XIo_DcrIn**(**Xuint32** *DcrRegister*)

Reads value from specified register.

Parameters:

DcrRegister is the intended source DCR register

Returns:

Contents of the specified DCR register.

Note:

None.

```
void XIo_DcrOut( Xuint32 DcrRegister,  
                Xuint32 Data  
                )
```

Outputs value provided to specified register defined in the header file.

Parameters:

DcrRegister is the intended destination DCR register

Data is the value to be placed into the specified DCR register

Returns:

None.

Note:

None.

intc/v1_00_b/src/xintc_l.c File Reference

Detailed Description

This file contains low-level driver functions that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	04/24/02	First release

```
#include "xparameters.h"  
#include "xbasic_types.h"  
#include "xintc_l.h"
```

Functions

void [XIntc_DefaultHandler](#) (void *UnusedInput)

void [XIntc_LowLevelInterruptHandler](#) (void)

Function Documentation

```
void XIntc_DefaultHandler( void * UnusedInput)
```

This function is an default interrupt handler for the low level driver of the interrupt controller. It allows the interrupt vector table to be initialized to this function so that unexpected interrupts don't result in a system crash.

Parameters:

UnusedInput is an unused input that is necessary for this function to have the signature of an input handler.

Returns:

None.

Note:

None.

void XIntc_LowLevelInterruptHandler(void)

This function is an interrupt handler for the low level driver of the interrupt controller. It must be connected to the interrupt source such that is called when an interrupt of the interrupt controller is active. It will resolve which interrupts are active and enabled and call the appropriate interrupt handler. It acknowledges the interrupt after it has been serviced by the interrupt handler.

This function assumes that an interrupt vector table has been previously initialized by the user. It does not verify that entries in the table are valid before calling an interrupt handler.

Returns:

None.

Note:

The constants `XPAR_INTC_SINGLE_BASEADDR` & `XPAR_INTC_MAX_ID` must be setup for this to compile. Interrupt IDs range from 0 - 31 and correspond to the interrupt input signals for the interrupt controller. `XPAR_INTC_MAX_ID` specifies the highest numbered interrupt input signal that is used.

XIntc_Config Struct Reference

```
#include <xintc.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xuint32 AckBeforeService

Field Documentation

Xuint32 XIntc_Config::AckBeforeService

Ack location per interrupt

Xuint32 XIntc_Config::BaseAddress

Register base address

Xuint16 XIntc_Config::DeviceId

Unique ID of device

The documentation for this struct was generated from the following file:

- [intc/v1_00_b/src/xintc.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

intc/v1_00_b/src/xintc_intr.c File Reference

Detailed Description

This file contains the interrupt processing for the **XIntc** component which is the driver for the Xilinx Interrupt Controller. The interrupt processing is partitioned separately such that users are not required to use the provided interrupt processing. This file requires other files of the driver to be linked in also.

Two different interrupt handlers are provided for this driver such that the user must select the appropriate handler for the application. The first interrupt handler, `XIntc_VoidInterruptHandler`, is provided for systems which use only a single interrupt controller or for systems that cannot otherwise provide an argument to the **XIntc** interrupt handler (e.g., the RTOS interrupt vector handler may not provide such a facility). The second interrupt handler, `XIntc_InterruptHandler`, uses an input argument which is an instance pointer to an interrupt controller driver such that multiple interrupt controllers can be supported. This handler requires the calling function to pass it the appropriate argument, so another level of indirection may be required.

The interrupt processing may be used by connecting one of the interrupt handlers to the interrupt system. These handlers do not save and restore the processor context but only handle the processing of the Interrupt Controller. The two handlers are provided as working examples. The user is encouraged to supply their own interrupt handler when performance tuning is deemed necessary.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/13/02	First release

```
#include "xbasic_types.h"  
#include "xintc_i.h"
```



```
#include "xintc.h"
```

Functions

```
void XIntc_VoidInterruptHandler ()
```

```
void XIntc_InterruptHandler (XIntc *InstancePtr)
```

Function Documentation

```
void XIntc_InterruptHandler( XIntc * InstancePtr)
```

The interrupt handler for the driver. This function determines the pending interrupts and calls the appropriate handlers in the priority based order.

This specific function allows multiple interrupt controller instances to be handled. The user must connect this function to the interrupt system such that it is called whenever the devices which are connected to it cause an interrupt.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

None.

Note:

None.

```
void XIntc_VoidInterruptHandler( )
```

Interrupt handler for the driver. This function determines the pending interrupts and calls the appropriate handlers in the priority based order.

This specific function does not support multiple interrupt controller instances to be handled. The user must connect this function to the interrupt system such that it is called whenever the devices which are connected to it cause an interrupt.

Returns:

None.

Note:

None.

intc/v1_00_b/src/xintc_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xintc_i.h,v 1.4 2002/05/02 20:35:24 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file intc/v1_00_b/src/xintc_i.h
00026 *
00027 * This file contains data which is shared between files and internal to the
00028 * XIntc component. It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date      Changes
00034 * -----
00035 * 1.00b  jhl    02/06/02  First release
00036 * 1.00b  jhl    04/24/02  Moved register definitions to xintc_l.h
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XINTC_I_H /* prevent circular inclusions */
00042 #define XINTC_I_H /* by using protection macros */
```

```
00043
00044 /***** Include Files *****/
00045
00046 #include "xbasic_types.h"
00047 #include "xintc.h"
00048
00049 /***** Constant Definitions *****/
00050
00051
00052 /***** Type Definitions *****/
00053
00054
00055 /***** Macros (Inline Functions) Definitions *****/
00056
00057
00058 /***** Function Prototypes *****/
00059
00060
00061 /***** Variable Definitions *****/
00062
00063 extern Xuint32 XIntc_BitPosMask[];
00064
00065 extern XIntc *XIntc_InstancePtr;
00066
00067 extern XIntc_Config XIntc_ConfigTable[];
00068
```

intc/v1_00_b/src/xintc_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XIntc** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	02/06/02	First release
1.00b	jhl	04/24/02	Moved register definitions to xintc_1.h

```
#include "xbasic_types.h"  
#include "xintc.h"
```

[Go to the source code of this file.](#)

Variables

[XIntc_Config](#) [XIntc_ConfigTable](#) []

Variable Documentation

[XIntc_Config](#) [XIntc_ConfigTable](#)[()]

This table contains configuration information for each intc device in the system. The **XIntc** driver must know when to acknowledge the interrupt. The entry which specifies this is a bit mask where each bit corresponds to a specific interrupt. A bit set indicates to ack it before servicing it. Generally, acknowledge before service is used when the interrupt signal is edge-sensitive, and after when the signal is level-sensitive.

Generated on 29 May 2003 for Xilinx Device Drivers

intc/v1_00_b/src/xintc_options.c File Reference

Detailed Description

Contains option functions for the **XIntc** driver. These functions allow the user to configure an instance of the **XIntc** driver. This file requires other files of the component to be linked in also.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	02/21/02	First release

```
#include "xbasic_types.h"  
#include "xintc.h"
```

Functions

XStatus **XIntc_SetOptions** (**XIntc** *InstancePtr, **Xuint32** Options)

Xuint32 **XIntc_GetOptions** (**XIntc** *InstancePtr)

Function Documentation

Xuint32 **XIntc_GetOptions**(**XIntc** * *InstancePtr*)

Return the currently set options.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

The currently set options. The options are described in **xintc.h**.

Note:

None.

```
XStatus XIntc_SetOptions( XIntc * InstancePtr,  
                          Xuint32 Options  
                          )
```

Set the options for the interrupt controller driver.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Options to be set. The available options are described in **xintc.h**.

Returns:

- XST_SUCCESS if the options were set successfully
- XST_INVALID_PARAM if the specified option was not valid

Note:

None.

intc/v1_00_b/src/xintc_selftest.c File Reference

Detailed Description

Contains diagnostic self-test functions for the **XIntc** component. This file requires other files of the component to be linked in also.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00b	jhl	02/21/02	First release

```
#include "xbasic_types.h"  
#include "xintc.h"  
#include "xintc_i.h"
```

Functions

XStatus **XIntc_SelfTest** (**XIntc** *InstancePtr)

XStatus **XIntc_SimulateIntr** (**XIntc** *InstancePtr, **Xuint8** Id)

Function Documentation

XStatus **XIntc_SelfTest**(**XIntc** * *InstancePtr*)

Run a self-test on the driver/device. This is a destructive test.

This involves forcing interrupts into the controller and verifying that they are recognized and can be acknowledged. This test will not succeed if the interrupt controller has been started in real mode such that interrupts cannot be forced.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Returns:

- XST_SUCCESS if self-test is successful.
- XST_INTC_FAIL_SELFTEST if the Interrupt controller fails the self-test. It will fail the self test if the device has previously been started in real mode.

Note:

None.

XStatus **XIntc_SimulateIntr**(**XIntc** * *InstancePtr*, **Xuint8** *Id*)

Allows software to simulate an interrupt in the interrupt controller. This function will only be successful when the interrupt controller has been started in simulation mode. Once it has been started in real mode, interrupts cannot be simulated. A simulated interrupt allows the interrupt controller to be tested without any device to drive an interrupt input signal into it.

Parameters:

InstancePtr is a pointer to the **XIntc** instance to be worked on.

Id is the interrupt ID for which to simulate an interrupt.

Returns:

XST_SUCCESS if successful, or XST_FAILURE if the interrupt could not be simulated because the interrupt controller is or has previously been in real mode.

Note:

None.

intc/v1_00_b/src/xintc_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of interrupt controller devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	01/09/02	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files
1.00b	jhl	04/24/02	Compressed the ack table into a bit mask.

```
#include "xintc.h"  
#include "xparameters.h"
```

Variables

[XIntc_Config](#) [XIntc_ConfigTable](#) [XPAR_XINTC_NUM_INSTANCES]

Variable Documentation

[XIntc_Config](#) [XIntc_ConfigTable](#)[XPAR_XINTC_NUM_INSTANCES]

This table contains configuration information for each intc device in the system. The **XIntc** driver must know when to acknowledge the interrupt. The entry which specifies this is a bit mask where each bit corresponds to a specific interrupt. A bit set indicates to ack it before servicing it. Generally, acknowledge before service is used when the interrupt signal is edge-sensitive, and after when the signal is level-sensitive.

Generated on 29 May 2003 for Xilinx Device Drivers

opb2plb/v1_00_a/src/xopb2plb.h File Reference

Detailed Description

This component contains the implementation of the **XOpb2Plb** component. It is the driver for the OPB to PLB Bridge. The bridge converts OPB bus transactions to PLB bus transactions. The hardware acts as a slave on the OPB side and as a master on the PLB side. This interface is necessary for the peripherals to access PLB based memory.

This driver allows the user to access the Bridge registers to support the handling of bus errors and other access errors and determine an appropriate solution if possible.

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

The Xilinx OPB to PLB Bridge is a soft IP core designed for Xilinx FPGAs and contains the following features:

- 64-bit PLB Master interface.
- Communicates with 32- or 64-bit PLB slaves.
- Non-burst transfers of 1-8 bytes.
- Burst transfers, including word and double-word bursts of fixed lengths, up to the depth of the burst buffer. Buffer depth configurable via a design parameter.
- Cacheline transactions of 4, 8, and 16 words.
- Programmable address boundaries that encompass the inverse of the PLB to OPB Bridge address space.
- Translates OPB data bursts to either cacheline or fixed length PLB burst transfers.
- Performing only CPU subset transactions (cachelines) can reduce system logic utilization and improve timing through the simplification of PLB slave IP.

- Using PLB burst transfers yields better bus cycle efficiency but may increase logic utilization and degrade timing in the system OPB Slave interface.
- 32-bit OPB Slave interface with byte enable transfers. *Note:* Does not support dynamic bus sizing or non-byte enable transactions. PLB and OPB clocks can have an asynchronous relationship (OPB clock frequency must be less than or equal to the PLB clock frequency).
- Bus Error Status Register (BESR) and Bus Error Address Register (BEAR) to report errors.
- DCR Slave interface provides access to BESR and BEAR.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. The current configuration data is contained in the [xopb2plb_g.c](#). A table is defined where each entry contains configuration information for a device. This information includes such things as the base address of the DCR mapped device.

Register Access

The bridge registers reside on the DCR address bus which is a parameter that can be selected in the hardware. If the DCR is not used, the registers reside in the OPB address space. A restriction of this driver is that if more than one bridge exists in the system, all must be configured the same way. That is, all must use DCR or all must use OPB.

Note:

This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	01/22/02	First release
1.00a	rpm	05/14/02	Made configuration table/data public

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xopb2plb_1.h"
#include "xio.h"
#include "xio_dcr.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XOpb2Plb  
struct XOpb2Plb\_Config
```

OPB-PLB bridge error status values

```
#define XO2P\_READ\_ERROR  
#define XO2P\_WRITE\_ERROR  
#define XO2P\_NO\_ERROR
```

Functions

```
XStatus XOpb2Plb\_Initialize (XOpb2Plb *InstancePtr, Xuint16 DeviceId)  
void XOpb2Plb\_Reset (XOpb2Plb *InstancePtr)  
XOpb2Plb\_Config * XOpb2Plb\_LookupConfig (Xuint16 DeviceId)  
Xboolean XOpb2Plb\_IsError (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_ClearErrors (XOpb2Plb *InstancePtr)  
Xuint32 XOpb2Plb\_GetErrorStatus (XOpb2Plb *InstancePtr)  
Xuint32 XOpb2Plb\_GetErrorAddress (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_EnableInterrupt (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_DisableInterrupt (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_EnableLock (XOpb2Plb *InstancePtr)  
void XOpb2Plb\_DisableLock (XOpb2Plb *InstancePtr)  
XStatus XOpb2Plb\_SelfTest (XOpb2Plb *InstancePtr, Xuint32 TestAddress)
```

Define Documentation

```
#define XO2P\_NO\_ERROR
```


XO2P_READ_ERROR	A read error occurred
XO2P_WRITE_ERROR	A write error occurred
XO2P_NO_ERROR	There is no error

#define XO2P_READ_ERROR

XO2P_READ_ERROR	A read error occurred
XO2P_WRITE_ERROR	A write error occurred
XO2P_NO_ERROR	There is no error

#define XO2P_WRITE_ERROR

XO2P_READ_ERROR	A read error occurred
XO2P_WRITE_ERROR	A write error occurred
XO2P_NO_ERROR	There is no error

Function Documentation

void XOpb2Plb_ClearErrors(XOpb2Plb * InstancePtr)

Clears the errors. If the lock bit is set, this allows subsequent errors to be recognized.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

void XOpb2Plb_DisableInterrupt(XOpb2Plb * InstancePtr)

Disables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void XOpb2Plb_DisableLock( XOpb2Plb * InstancePtr)
```

Disables the locking of the status on error for the bridge. This 'unlocks' the status and address registers allowing subsequent errors to overwrite the current values when an error occurs.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

```
void XOpb2Plb_EnableInterrupt( XOpb2Plb * InstancePtr)
```

Enables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void XOpb2Plb_EnableLock( XOpb2Plb * InstancePtr)
```

Enables the locking of the status on error for the bridge. This 'locks' the status and address register values when an error occurs, preventing subsequent errors from overwriting the values. Clearing the error allows the status and address registers to update with the next error that occurs

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 XOpb2Plb_GetErrorAddress( XOpb2Plb * InstancePtr)
```

Returns the PLB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XOpb2Plb_IsError()** is recommended to confirm that an error has occurred prior to calling **XOpb2Plb_GetErrorAddress()** to ensure that the data in the error address register is relevant.

```
Xuint32 XOpb2Plb_GetErrorStatus( XOpb2Plb * InstancePtr )
```

Returns the error status indicating the type of error that has occurred.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

The current error status for the OPB to PLB bridge. The possible return values are described in **xopb2plb.h**.

Note:

None.

```
XStatus XOpb2Plb_Initialize( XOpb2Plb * InstancePtr,  
                             Xuint16      DeviceId  
                             )
```

Initializes a specific **XOpb2Plb** instance. Looks up the configuration data for the given device ID and then initializes instance data.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XOpb2Plb** component. Passing in a device id associates the generic **XOpb2Plb** component to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

Xboolean XOpb2Plb_IsError(XOpb2Plb * InstancePtr)

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

XOpb2Plb_Config* XOpb2Plb_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table `OpbPlbConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device to look for.

Returns:

A pointer to the configuration data of the given device, or XNULL if no match is found.

Note:

None.

```
void XOpb2Plb_Reset( XOpb2Plb * InstancePtr)
```

Forces a software-induced reset to occur in the bridge and disables interrupts and the locking functionality in the process.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

Disables interrupts and the locking functionality in the process.

```
XStatus XOpb2Plb_SelfTest( XOpb2Plb * InstancePtr,  
                          Xuint32 TestAddress  
                          )
```

Runs a self-test on the driver/device.

This tests reads the provided `TestAddress` which is intended to cause an error. Then the `XOpb2Plb_IsError()` routine is called and if there is an error, the address is checked against the provided location and if they match `XST_SUCCESS` is returned and all errors are then cleared.

If the `XOpb2Plb_IsError()` is called and no error is indicated, `XST_FAILURE` is returned.

Parameters:

InstancePtr is a pointer to the `XOpb2Plb` instance to be worked on.

TestAddress is a location that should cause an error on read.

Returns:

`XST_SUCCESS` if successful, or `XST_FAILURE` if the driver fails the self-test.

Note:

This test assumes that the bus error interrupts to the processor are not enabled.

XOpb2Plb Struct Reference

```
#include <xopb2plb.h>
```

Detailed Description

The XOpb2Plb driver instance data. The user is required to allocate a variable of this type for every OPB-to-PLB bridge device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [opb2plb/v1_00_a/src/xopb2plb.h](#)

opb2plb/v1_00_a/src/xopb2plb.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xopb2plb.h,v 1.3 2002/07/26 20:46:38 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file opb2plb/v1_00_a/src/xopb2plb.h
00026 *
00027 * This component contains the implementation of the XOpb2Plb component. It is
00028 * the driver for the OPB to PLB Bridge. The bridge converts OPB bus
transactions
00029 * to PLB bus transactions. The hardware acts as a slave on the OPB side and as
00030 * a master on the PLB side. This interface is necessary for the peripherals to
00031 * access PLB based memory.
00032 *
00033 * This driver allows the user to access the Bridge registers to support the
00034 * handling of bus errors and other access errors and determine an appropriate
00035 * solution if possible.
00036 *
00037 * The bridge hardware generates interrupts in error conditions. These
interrupts
00038 * are not handled by the driver directly. It is the application's
responsibility
00039 * to attach to the appropriate interrupt with a handler which then calls
```

00040 * functions provided by this driver to determine the cause of the error and
take
00041 * the necessary actions to correct the situation.
00042 *
00043 * The Xilinx OPB to PLB Bridge is a soft IP core designed for Xilinx FPGAs
00044 * and contains the following features:
00045 * - 64-bit PLB Master interface.
00046 * - Communicates with 32- or 64-bit PLB slaves.
00047 * - Non-burst transfers of 1-8 bytes.
00048 * - Burst transfers, including word and double-word bursts of fixed lengths,
00049 * up to the depth of the burst buffer. Buffer depth configurable via a
00050 * design parameter.
00051 * - Cacheline transactions of 4, 8, and 16 words.
00052 * - Programmable address boundaries that encompass the inverse of the PLB to
00053 * OPB Bridge address space.
00054 * - Translates OPB data bursts to either cacheline or fixed length PLB burst
00055 * transfers.
00056 * - Performing only CPU subset transactions (cachelines) can reduce system
00057 * logic utilization and improve timing through the simplification of PLB
00058 * slave IP.
00059 * - Using PLB burst transfers yields better bus cycle efficiency but may
00060 * increase logic utilization and degrade timing in the system
00061 * OPB Slave interface.
00062 * - 32-bit OPB Slave interface with byte enable transfers.
00063 * *Note*: Does not support dynamic bus sizing or non-byte enable
00064 * transactions. PLB and OPB clocks can have an asynchronous relationship
00065 * (OPB clock frequency must be less than or equal to the PLB clock
frequency).
00066 * - Bus Error Status Register (BESR) and Bus Error Address Register (BEAR) to
00067 * report errors.
00068 * - DCR Slave interface provides access to BESR and BEAR.
00069 *
00070 * **Device Configuration**
00071 *
00072 * The device can be configured in various ways during the FPGA implementation
00073 * process. The current configuration data is contained in the xopb2plb_g.c.
00074 * A table is defined where each entry contains configuration information for a
00075 * device. This information includes such things as the base address of the DCR
00076 * mapped device.
00077 *
00078 * **Register Access**
00079 *
00080 * The bridge registers reside on the DCR address bus which is a parameter that
00081 * can be selected in the hardware. If the DCR is not used, the registers
00082 * reside in the OPB address space. A restriction of this driver is that if
00083 * more than one bridge exists in the system, all must be configured the same
00084 * way. That is, all must use DCR or all must use OPB.
00085 *
00086 * @note
00087 *
00088 * This driver is not thread-safe. Thread safety must be guaranteed by the layer
00089 * above this driver if there is a need to access the device from multiple
00090 * threads.

```

00091 *
00092 * <pre>
00093 * MODIFICATION HISTORY:
00094 *
00095 * Ver    Who    Date      Changes
00096 * -----
00097 * 1.00a ecm    01/22/02 First release
00098 * 1.00a rpm    05/14/02 Made configuration table/data public
00099 * </pre>
00100 *
00101 *****/
00102
00103 #ifndef XOPB2PLB_H /* prevent circular inclusions */
00104 #define XOPB2PLB_H /* by using protection macros */
00105
00106 /***** Include Files *****/
00107 #include "xbasic_types.h"
00108 #include "xstatus.h"
00109 #include "xopb2plb_1.h"
00110 #include "xio.h"
00111 #include "xio_dcr.h"
00112
00113 /***** Constant Definitions *****/
00114
00115
00116 /** @name OPB-PLB bridge error status values
00117  * @{
00118  */
00119 /**
00120  * <pre>
00121  * XO2P_READ_ERROR      A read error occurred
00122  * XO2P_WRITE_ERROR     A write error occurred
00123  * XO2P_NO_ERROR        There is no error
00124  * </pre>
00125  */
00126 #define XO2P_READ_ERROR      2
00127 #define XO2P_WRITE_ERROR     1
00128 #define XO2P_NO_ERROR        0
00129 /*@}*/
00130
00131 /***** Type Definitions *****/
00132
00133 /**
00134  * This typedef contains configuration information for the device.
00135  */
00136 typedef struct
00137 {
00138     Xuint16 DeviceId;      /**< Unique ID of device */
00139     Xuint32 BaseAddress;   /**< Register base address */
00140 } XOpb2Plb_Config;
00141

```

```

00142 /**
00143  * The XOpb2Plb driver instance data. The user is required to allocate a
00144  * variable of this type for every OPB-to-PLB bridge device in the system. A
00145  * pointer to a variable of this type is then passed to the driver API
00146  * functions.
00147  */
00148 typedef struct
00149 {
00150     Xuint32 BaseAddress;      /* Base address of registers */
00151     Xuint32 IsReady;         /* Device is initialized and ready */
00152 } XOpb2Plb;
00153
00154
00155
00156 /***** Macros (Inline Functions) Definitions *****/
00157
00158
00159 /***** Function Prototypes *****/
00160
00161 /* required functions in xopb2plb.c */
00162
00163 /*
00164  * Initialization Functions
00165  */
00166 XStatus XOpb2Plb_Initialize(XOpb2Plb *InstancePtr, Xuint16 DeviceId);
00167 void XOpb2Plb_Reset(XOpb2Plb *InstancePtr);
00168 XOpb2Plb_Config *XOpb2Plb_LookupConfig(Xuint16 DeviceId);
00169
00170 /*
00171  * Access Functions
00172  */
00173
00174 Xboolean XOpb2Plb_IsError(XOpb2Plb *InstancePtr);
00175 void XOpb2Plb_ClearErrors(XOpb2Plb *InstancePtr);
00176
00177 Xuint32 XOpb2Plb_GetErrorStatus(XOpb2Plb *InstancePtr);
00178 Xuint32 XOpb2Plb_GetErrorAddress(XOpb2Plb *InstancePtr);
00179
00180 /*
00181  * Configuration
00182  */
00183 void XOpb2Plb_EnableInterrupt(XOpb2Plb *InstancePtr);
00184 void XOpb2Plb_DisableInterrupt(XOpb2Plb *InstancePtr);
00185 void XOpb2Plb_EnableLock(XOpb2Plb *InstancePtr);
00186 void XOpb2Plb_DisableLock(XOpb2Plb *InstancePtr);
00187
00188 /* contained in xopb2plb_selftest.c */
00189
00190 XStatus XOpb2Plb_SelfTest(XOpb2Plb *InstancePtr, Xuint32 TestAddress);
00191
00192 #endif          /* end of protection macro */

```

Generated on 29 May 2003 for Xilinx Device Drivers

opb2plb/v1_00_a/src/xopb2plb_l.h File Reference

Detailed Description

This file contains identifiers and low-level macros that can be used to access the device directly.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	05/14/02	First release

```
#include "xparameters.h"  
#include "xbasic_types.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XOpb2Plb_mGetBesrReg(BaseAddress)  
#define XOpb2Plb_mGetBearReg(BaseAddress)  
#define XOpb2Plb_mSetControlReg(BaseAddress, Mask)  
#define XOpb2Plb_mGetControlReg(BaseAddress)
```

Define Documentation

```
#define XOpb2Plb_mGetBearReg( BaseAddress )
```

Get the bus error address register (BEAR).

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the error address register.

Note:

None.

```
#define XOpb2Plb_mGetBesrReg( BaseAddress )
```

Get the bus error status register (BESR).

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the error status register.

Note:

None.

```
#define XOpb2Plb_mGetControlReg( BaseAddress )
```

Get the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the control register.

Note:

None.

```
#define XOpb2Plb_mSetControlReg( BaseAddress,  
                                Mask          )
```

Set the control register to the given value.

Parameters:

BaseAddress is the base address of the device

Mask is the value to write to the control register.

Returns:

None.

Note:

None.

opb2plb/v1_00_a/src/xopb2plb_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xopb2plb_l.h,v 1.4 2002/07/26 21:07:57 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file opb2plb/v1_00_a/src/xopb2plb_l.h
00026 *
00027 * This file contains identifiers and low-level macros that can be used to
00028 * access the device directly.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date        Changes
00034 * -----
00035 * 1.00a rpm    05/14/02 First release
00036 * </pre>
00037 *
00038 *****/
00039
00040 #ifndef XOPB2PLB_L_H /* prevent circular inclusions */
00041 #define XOPB2PLB_L_H /* by using protection macros */
00042
```

```

00043 /***** Include Files *****/
00044 #include "xparameters.h"
00045 #include "xbasic_types.h"
00046 #include "xio.h"
00047 #include "xio_dcr.h"
00048
00049 /* generic from MPD file is C_OPB_REG_INTFC */
00050 /* uncomment when the functionality is in libgen
00051 #ifndef XPAR_XOPB2PLB_OPB_REG_INTFC
00052 #error "XPAR_XOPB2PLB_OPB_REG_INTFC not defined"
00053 #endif
00054 */
00055
00056 /***** Constant Definitions *****/
00057
00058 #if (XPAR_XOPB2PLB_OPB_REG_INTFC == 0)
00059 /*
00060  * OPB-PLB Bridge Register Addresses - DCR bus
00061  */
00062 #define XO2P_BESR_OFFSET          0x00 /* error register */
00063 #define XO2P_BEAR_ADDR_OFFSET    0x01 /* address where error occurred */
00064 #define XO2P_BCR_OFFSET          0x02 /* control and status register */
00065
00066 #else
00067 /*
00068  * OPB-PLB Bridge Register Addresses - OPB bus
00069  */
00070 #define XO2P_BESR_OFFSET          0x00 /* error register */
00071 #define XO2P_BEAR_ADDR_OFFSET    0x04 /* address where error occurred */
00072 #define XO2P_BCR_OFFSET          0x08 /* control and status register */
00073
00074 #endif
00075
00076
00077 /* BESR Register masks */
00078 #define XO2P_BESR_ERROR_MASK      0x80000000 /* error has occurred */
00079 #define XO2P_BESR_READ_ERROR_MASK 0x40000000 /* read error occurred */
00080 #define XO2P_BESR_BYTE_EN_MASK   0x0000FF00 /* PLB byte enables */
00081
00082 /* BCR Register masks */
00083 #define XO2P_BCR_CLEAR_ERROR_MASK 0x80000000 /* clear the error (write only)
*/
00084 #define XO2P_BCR_ENABLE_LOCK_MASK 0x40000000 /* enable lock on error */
00085 #define XO2P_BCR_ENABLE_INTR_MASK 0x20000000 /* enable interrupts */
00086 #define XO2P_BCR_RESET_MASK      0x10000000 /* force reset */
00087
00088
00089 /***** Type Definitions *****/
00090
00091 /***** Macros (Inline Functions) Definitions *****/
00092
00093 /* Define the appropriate I/O access method for the bridge which may be

```

```

00094 * memory mapped or DCR
00095 */
00096 #if (XPAR_XOPB2PLB_OPB_REG_INTFC == 0)
00097
00098 #define XOpb2Plb_In32    XIo_DcrIn
00099 #define XOpb2Plb_Out32  XIo_DcrOut
00100
00101 #else
00102
00103 #define XOpb2Plb_In32    XIo_In32
00104 #define XOpb2Plb_Out32  XIo_Out32
00105
00106 #endif
00107
00108
00109 /*****
00110 *
00111 * Low-level driver macros and functions. The list below provides signatures
00112 * to help the user use the macros.
00113 *
00114 * Xuint32 XOpb2Plb_mGetBesrReg(Xuint32 BaseAddress)
00115 * Xuint32 XOpb2Plb_mGetBearReg(Xuint32 BaseAddress)
00116 *
00117 * void XOpb2Plb_mSetControlReg(Xuint32 BaseAddress, Xuint32 Mask)
00118 * Xuint32 XOpb2Plb_mGetControlReg(Xuint32 BaseAddress)
00119 *
00120 *****/
00121
00122 /*****/
00123 /**
00124 *
00125 * Get the bus error status register (BESR).
00126 *
00127 * @param    BaseAddress is the base address of the device
00128 *
00129 * @return   The 32-bit value of the error status register.
00130 *
00131 * @note    None.
00132 *
00133 *****/
00134 #define XOpb2Plb_mGetBesrReg(BaseAddress) \
00135         XOpb2Plb_In32((BaseAddress) + XO2P_BESR_OFFSET)
00136
00137
00138 /*****/
00139 /**
00140 *
00141 * Get the bus error address register (BEAR).
00142 *
00143 * @param    BaseAddress is the base address of the device
00144 *
00145 * @return   The 32-bit value of the error address register.

```

```

00146 *
00147 * @note      None.
00148 *
00149 *****/
00150 #define XOpb2Plb_mGetBearReg(BaseAddress) \
00151         XOpb2Plb_In32((BaseAddress) + XO2P_BEAR_ADDR_OFFSET)
00152
00153 *****/
00154 /**
00155 *
00156 * Set the control register to the given value.
00157 *
00158 * @param      BaseAddress is the base address of the device
00159 * @param      Mask is the value to write to the control register.
00160 *
00161 * @return     None.
00162 *
00163 * @note      None.
00164 *
00165 *****/
00166 #define XOpb2Plb_mSetControlReg(BaseAddress, Mask) \
00167         XOpb2Plb_Out32((BaseAddress) + XO2P_BCR_OFFSET, (Mask))
00168
00169
00170 *****/
00171 /**
00172 *
00173 * Get the contents of the control register.
00174 *
00175 * @param      BaseAddress is the base address of the device
00176 *
00177 * @return     The 32-bit value of the control register.
00178 *
00179 * @note      None.
00180 *
00181 *****/
00182 #define XOpb2Plb_mGetControlReg(BaseAddress) \
00183         XOpb2Plb_In32((BaseAddress) + XO2P_BCR_OFFSET)
00184
00185
00186 *****/ Function Prototypes *****/
00187
00188 *****/ Variables *****/
00189
00190
00191 #endif          /* end of protection macro */

```

XOpb2Plb_Config Struct Reference

```
#include <xopb2plb.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Field Documentation

Xuint32 XOpb2Plb_Config::BaseAddress

Register base address

Xuint16 XOpb2Plb_Config::DeviceId

Unique ID of device

The documentation for this struct was generated from the following file:

- opb2plb/v1_00_a/src/xopb2plb.h
-

opb2plb/v1_00_a/src/xopb2plb.c File Reference

Detailed Description

Contains required functions for the **XOpb2Plb** component. See [xopb2plb.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	01/22/02	First release

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xopb2plb.h"  
#include "xopb2plb_i.h"
```

Functions

```
XStatus XOpb2Plb_Initialize (XOpb2Plb *InstancePtr, Xuint16 DeviceId)  
void XOpb2Plb_Reset (XOpb2Plb *InstancePtr)  
Xboolean XOpb2Plb_IsError (XOpb2Plb *InstancePtr)  
void XOpb2Plb_ClearErrors (XOpb2Plb *InstancePtr)  
Xuint32 XOpb2Plb_GetErrorStatus (XOpb2Plb *InstancePtr)  
Xuint32 XOpb2Plb_GetErrorAddress (XOpb2Plb *InstancePtr)
```

void **XOpb2Plb_EnableInterrupt** (**XOpb2Plb** *InstancePtr)

void **XOpb2Plb_DisableInterrupt** (**XOpb2Plb** *InstancePtr)

void **XOpb2Plb_EnableLock** (**XOpb2Plb** *InstancePtr)

void **XOpb2Plb_DisableLock** (**XOpb2Plb** *InstancePtr)

XOpb2Plb_Config * **XOpb2Plb_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

void XOpb2Plb_ClearErrors(**XOpb2Plb** * *InstancePtr*)

Clears the errors. If the lock bit is set, this allows subsequent errors to be recognized.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

void XOpb2Plb_DisableInterrupt(**XOpb2Plb** * *InstancePtr*)

Disables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

void XOpb2Plb_DisableLock(XOpb2Plb * InstancePtr)

Disables the locking of the status on error for the bridge. This 'unlocks' the status and address registers allowing subsequent errors to overwrite the current values when an error occurs.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

void XOpb2Plb_EnableInterrupt(XOpb2Plb * InstancePtr)

Enables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

void XOpb2Plb_EnableLock(XOpb2Plb * InstancePtr)

Enables the locking of the status on error for the bridge. This 'locks' the status and address register values when an error occurs, preventing subsequent errors from overwriting the values. Clearing the error allows the status and address registers to update with the next error that occurs

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

None.

Xuint32 XOpb2Plb_GetErrorAddress(**XOpb2Plb** * *InstancePtr*)

Returns the PLB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XOpb2Plb_IsError()** is recommended to confirm that an error has occurred prior to calling **XOpb2Plb_GetErrorAddress()** to ensure that the data in the error address register is relevant.

Xuint32 XOpb2Plb_GetErrorStatus(**XOpb2Plb** * *InstancePtr*)

Returns the error status indicating the type of error that has occurred.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

The current error status for the OPB to PLB bridge. The possible return values are described in **xopb2plb.h**.

Note:

None.

```
XStatus XOpb2Plb_Initialize( XOpb2Plb * InstancePtr,  
                             Xuint16   DeviceId  
                             )
```

Initializes a specific **XOpb2Plb** instance. Looks up the configuration data for the given device ID and then initializes instance data.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XOpb2Plb** component. Passing in a device id associates the generic **XOpb2Plb** component to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
Xboolean XOpb2Plb_IsError( XOpb2Plb * InstancePtr)
```

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

XOpb2Plb_Config* XOpb2Plb_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table OpbPlbConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID of the device to look for.

Returns:

A pointer to the configuration data of the given device, or XNULL if no match is found.

Note:

None.

void XOpb2Plb_Reset(XOpb2Plb * InstancePtr)

Forces a software-induced reset to occur in the bridge and disables interrupts and the locking functionality in the process.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

Returns:

None.

Note:

Disables interrupts and the locking functionality in the process.

Generated on 29 May 2003 for Xilinx Device Drivers

opb2plb/v1_00_a/src/xopb2plb_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xopb2plb_i.h,v 1.1 2002/06/26 19:05:15 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file opb2plb/v1_00_a/src/xopb2plb_i.h
00026 *
00027 * This file contains data which is shared between files and internal to the
00028 * XOpb2Plb component. It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date      Changes
00034 * -----
00035 * 1.00a  ecm    02/28/02  First release
00036 * 1.00a  rpm    05/14/02  Moved identifiers to xopb2plb_l.h
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XOPB2PLB_I_H /* prevent circular inclusions */
00042 #define XOPB2PLB_I_H /* by using protection macros */
```

```
00043
00044 /***** Include Files *****/
00045 #include "xopb2plb_1.h"
00046
00047 /***** Constant Definitions *****/
00048
00049 /***** Type Definitions *****/
00050
00051 /***** Macros (Inline Functions) Definitions *****/
00052
00053 /***** Function Prototypes *****/
00054
00055 /***** Variables *****/
00056
00057 extern XOpb2Plb_Config XOpb2Plb_ConfigTable[];
00058
00059
00060 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

opb2plb/v1_00_a/src/xopb2plb_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XOpb2Plb** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	ecm	02/28/02	First release
1.00a	rpm	05/14/02	Moved identifiers to xopb2plb_1.h

```
#include "xopb2plb\_1.h"
```

[Go to the source code of this file.](#)

Variables

[XOpb2Plb_Config](#) [XOpb2Plb_ConfigTable](#) []

Variable Documentation

XOpb2Plb_Config XOpb2Plb_ConfigTable[]()

The OPB-to-PLB bridge configuration table, sized by the number of instances defined in **xparameters.h**.

Generated on 29 May 2003 for Xilinx Device Drivers

opb2plb/v1_00_a/src/xopb2plb_selftest.c

File Reference

Detailed Description

Contains diagnostic self-test functions for the **XOpb2Plb** component. See [xopb2plb.h](#) for more information about the component.

This functionality assumes that the initialize function has been called prior to calling the self-test function.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	11/16/01	First release

```
#include "xstatus.h"  
#include "xopb2plb.h"  
#include "xopb2plb_i.h"
```

Functions

[XStatus XOpb2Plb_SelfTest](#) ([XOpb2Plb](#) *InstancePtr, [Xuint32](#) TestAddress)

Function Documentation

```
XStatus XOpb2Plb_SelfTest( XOpb2Plb * InstancePtr,  
                           Xuint32   TestAddress  
                           )
```

Runs a self-test on the driver/device.

This tests reads the provided *TestAddress* which is intended to cause an error. Then the **XOpb2Plb_IsError()** routine is called and if there is an error, the address is checked against the provided location and if they match **XST_SUCCESS** is returned and all errors are then cleared.

If the **XOpb2Plb_IsError()** is called and no error is indicated, **XST_FAILURE** is returned.

Parameters:

InstancePtr is a pointer to the **XOpb2Plb** instance to be worked on.

TestAddress is a location that should cause an error on read.

Returns:

XST_SUCCESS if successful, or **XST_FAILURE** if the driver fails the self-test.

Note:

This test assumes that the bus error interrupts to the processor are not enabled.

opb2plb/v1_00_a/src/xopb2plb_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of OPB-to-PLB bridge devices in the system. Each bridge device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	01/22/02	First release
1.00a	rpm	05/14/02	Made configuration table/data public

```
#include "xopb2plb.h"  
#include "xparameters.h"
```

Variables

[XOpb2Plb_Config](#) [XOpb2Plb_ConfigTable](#) [XPAR_XOPB2PLB_NUM_INSTANCES]

Variable Documentation

[XOpb2Plb_Config](#) [XOpb2Plb_ConfigTable](#)[XPAR_XOPB2PLB_NUM_INSTANCES]

The OPB-to-PLB bridge configuration table, sized by the number of instances defined in [xparameters.h](#).

Generated on 29 May 2003 for Xilinx Device Drivers

opbarb/v1_02_a/src/xopbarb.h File Reference

Detailed Description

This component contains the implementation of the **XOpbArb** component. It is the driver for the On-chip Peripheral Bus (OPB) Arbiter. The arbiter performs bus arbitration for devices on the OPB

Priority Arbitration

By default, the arbiter prioritizes masters on the bus based on their master IDs. A master's ID corresponds to the signals with which it connects to the arbiter. Master 0 is the highest priority, master 1 the next highest, and so on. The device driver allows an application to modify this default behavior.

There are two modes of priority arbitration, dynamic and fixed. The device can be parameterized in either of these modes. Fixed priority arbitration makes use of priority levels that can be configured by software. There is one level for each master on the bus. Priority level 0 is assigned the master ID of the highest priority master, priority level 1 is assigned the master ID of the next highest priority master, and so on.

Dynamic priority arbitration utilizes a Least Recently Used(LRU) algorithm to determine the priorities of masters on the bus. Once a master is granted the bus, it falls to the lowest priority master. A master that is not granted the bus moves up in priority until it becomes the highest priority master based on the fact that it has been the least recently used master. The arbiter uses the currently assigned priority levels as its starting configuration for the LRU algorithm. Software can modify this starting configuration by assigning master IDs to the priority levels.

When configuring priority levels (i.e., assigning master IDs to priority levels), the application must suspend use of the priority levels by the device. Every master must be represented by one and only one priority level. The device driver enforces this by making the application suspend use of the priority levels by the device during the time it takes to correctly configure the levels. Once the levels are configured, the application must explicitly resume use of the priority levels by the device. During the time priority levels

are suspended, the device reverts to its default behavior of assigning priorities based on master IDs.

Bus Parking

By default, bus parking is disabled. The device driver allows an application to enable bus parking, which forces the arbiter to always assert the grant signal for a specific master when no other masters are requesting the bus. The master chosen for parking is either the master that was last granted the bus, or the master configured by the SetParkId function. When bus parking is enabled but no park ID has been set, bus parking defaults to the master that was last granted the bus.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in `xopbarb_g.c`. A table is defined where each entry contains configuration information for a device. This information includes such things as the base address of the memory-mapped device, the number of masters on the bus, and the priority arbitration scheme.

When the device is parameterized with only 1 master, or if the device is parameterized without a slave interface, there are no registers accessible to software and no configuration entry is available. In these configurations it is likely that the driver will not be loaded or used by the application. But in the off-chance that it is, it is assumed that no configuration information for the arbiter is entered in the `xopbarb_g.c` table. If config information were entered for the device, there will be nothing to prevent the driver's use, and any use of the driver under these circumstances will result in undefined behavior.

Note:

This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.02a	rpm	08/13/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xopbarb_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XOpbArb  
struct XOpbArb_Config
```

Configuration options

```
#define XOA_DYNAMIC_PRIORITY_OPTION  
#define XOA_PARK_ENABLE_OPTION  
#define XOA_PARK_BY_ID_OPTION
```

Functions

```
XStatus XOpbArb_Initialize (XOpbArb *InstancePtr, Xuint16 DeviceId)  
XOpbArb_Config * XOpbArb_LookupConfig (Xuint16 DeviceId)  
XStatus XOpbArb_SelfTest (XOpbArb *InstancePtr)  
XStatus XOpbArb_SetOptions (XOpbArb *InstancePtr, Xuint32 Options)  
Xuint32 XOpbArb_GetOptions (XOpbArb *InstancePtr)  
XStatus XOpbArb_SetPriorityLevel (XOpbArb *InstancePtr, Xuint8 Level, Xuint8  
    MasterId)  
Xuint8 XOpbArb_GetPriorityLevel (XOpbArb *InstancePtr, Xuint8 Level)  
void XOpbArb_SuspendPriorityLevels (XOpbArb *InstancePtr)  
XStatus XOpbArb_ResumePriorityLevels (XOpbArb *InstancePtr)  
XStatus XOpbArb_SetParkId (XOpbArb *InstancePtr, Xuint8 MasterId)  
XStatus XOpbArb_GetParkId (XOpbArb *InstancePtr, Xuint8 *MasterIdPtr)
```

Define Documentation

```
#define XOA_DYNAMIC_PRIORITY_OPTION
```


The options enable or disable additional features of the OPB Arbiter. Each of the options are bit fields such that more than one may be specified.

- **XOA_DYNAMIC_PRIORITY_OPTION**

The Dynamic Priority option configures the device for dynamic priority arbitration, which uses a Least Recently Used (LRU) algorithm to determine the priorities of OPB masters. This option is not applicable if the device is parameterized for fixed priority arbitration. When the device is parameterized for dynamic priority arbitration, it can still use a fixed priority arbitration by turning this option off. Fixed priority arbitration uses the priority levels as written by software to determine the priorities of OPB masters. The default is fixed priority arbitration.

- **XOA_PARK_ENABLE_OPTION**

The Park Enable option enables bus parking, which forces the arbiter to always assert the grant signal for a specific master when no other masters are requesting the bus. The master chosen for parking is either the master that was last granted the bus, or the master configured by the SetParkId function.

- **XOA_PARK_BY_ID_OPTION**

The Park By ID option enables bus parking based on the specific master ID. The master ID defaults to master 0, and can be changed using [XOpbArb_SetParkId\(\)](#). When this option is disabled, bus parking defaults to the master that was last granted the bus. The park enable option must be set for this option to take effect.

#define XOA_PARK_BY_ID_OPTION

The options enable or disable additional features of the OPB Arbiter. Each of the options are bit fields such that more than one may be specified.

- **XOA_DYNAMIC_PRIORITY_OPTION**

The Dynamic Priority option configures the device for dynamic priority arbitration, which uses a Least Recently Used (LRU) algorithm to determine the priorities of OPB masters. This option is not applicable if the device is parameterized for fixed priority arbitration. When the device is parameterized for dynamic priority arbitration, it can still use a fixed priority arbitration by turning this option off. Fixed priority arbitration uses the priority levels as written by software to determine the priorities of OPB masters. The default is fixed priority arbitration.

- **XOA_PARK_ENABLE_OPTION**

The Park Enable option enables bus parking, which forces the arbiter to always assert the grant signal for a specific master when no other masters are requesting the bus. The master chosen

for parking is either the master that was last granted the bus, or the master configured by the `SetParkId` function.

- **XOA_PARK_BY_ID_OPTION**

The Park By ID option enables bus parking based on the specific master ID. The master ID defaults to master 0, and can be changed using `XOpbArb_SetParkId()`. When this option is disabled, bus parking defaults to the master that was last granted the bus. The park enable option must be set for this option to take effect.

#define XOA_PARK_ENABLE_OPTION

The options enable or disable additional features of the OPB Arbiter. Each of the options are bit fields such that more than one may be specified.

- **XOA_DYNAMIC_PRIORITY_OPTION**

The Dynamic Priority option configures the device for dynamic priority arbitration, which uses a Least Recently Used (LRU) algorithm to determine the priorities of OPB masters. This option is not applicable if the device is parameterized for fixed priority arbitration. When the device is parameterized for dynamic priority arbitration, it can still use a fixed priority arbitration by turning this option off. Fixed priority arbitration uses the priority levels as written by software to determine the priorities of OPB masters. The default is fixed priority arbitration.

- **XOA_PARK_ENABLE_OPTION**

The Park Enable option enables bus parking, which forces the arbiter to always assert the grant signal for a specific master when no other masters are requesting the bus. The master chosen for parking is either the master that was last granted the bus, or the master configured by the `SetParkId` function.

- **XOA_PARK_BY_ID_OPTION**

The Park By ID option enables bus parking based on the specific master ID. The master ID defaults to master 0, and can be changed using `XOpbArb_SetParkId()`. When this option is disabled, bus parking defaults to the master that was last granted the bus. The park enable option must be set for this option to take effect.

Xuint32 XOpbArb_GetOptions(**XOpbArb** * *InstancePtr*)

Gets the options for the arbiter. The options control how the device grants the bus to requesting masters.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

The options of the device. This is a bit mask where a 1 means the option is on, and a 0 means the option is off. See [xopbarb.h](#) for a description of the options.

Note:

None.

XStatus XOpbArb_GetParkId(**XOpbArb** * *InstancePtr*, **Xuint8** * *MasterIdPtr*)

Gets the master ID currently used for bus parking.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

MasterIdPtr is a pointer to a byte that will hold the master ID currently used for bus parking. This is an output parameter. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

XST_SUCCESS if the park ID is successfully retrieved, or XST_NO_FEATURE if bus parking is not supported by the device.

Note:

None.

Xuint8 XOpbArb_GetPriorityLevel(**XOpbArb** * *InstancePtr*, **Xuint8** *Level*)

Get the master ID at the given priority level.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Level is the priority level being retrieved. The level can range from 0 (highest) to N (lowest), where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

The master ID assigned to the given priority level. The ID can range from 0 to N, where N is the number of masters minus one.

Note:

If the arbiter is operating in dynamic priority mode, the value returned from this function may not be predictable because the arbiter changes the values on the fly.

```
XStatus XOpbArb_Initialize( XOpbArb * InstancePtr,  
                           Xuint16   DeviceId  
                           )
```

Initializes a specific **XOpbArb** instance. The driver is initialized to allow access to the device registers. In addition, the configuration information is retrieved for the device. Currently, configuration information is stored in **xopbarb_g.c**.

The state of the device after initialization is:

- Fixed or dynamic priority arbitration based on hardware parameter
- Bus parking is disabled

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XOpbArb** component. Passing in a device id associates the generic **XOpbArb** component to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful or XST_DEVICE_NOT_FOUND if no configuration data was found for this device.

Note:

None.

XOpbArb_Config* XOpbArb_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table OpbArbConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to match on.

Returns:

A pointer to the configuration information for the matching device instance, or XNULL if no match is found.

Note:

None.

XStatus XOpbArb_ResumePriorityLevels(XOpbArb * InstancePtr)

Resumes use of the priority levels by the device. This function is typically called sometime after a call to SuspendPriorityLevels. The application must resume use of priority levels by the device when all modifications are done. If no call is made to this function after use of the priority levels has been suspended, the device will remain in its default priority arbitration mode of assigning priorities based on master IDs. A call to this function has no effect if no prior call was made to suspend the use of priority levels.

Every master must be represented by one and only one fixed priority level before the use of priority levels can be resumed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

- XST_SUCCESS if the slave is selected successfully.
- XST_OPBARB_INVALID_PRIORITY if there is either a master that is not assigned a priority level, or a master that is assigned two or more priority levels.

Note:

None.

```
XStatus XOpbArb_SelfTest( XOpbArb * InstancePtr)
```

Runs a self-test on the driver/device. The self-test simply verifies that the arbiter's registers can be read and written. This is an intrusive test in that the arbiter will not be using the priority registers while the test is being performed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

XST_SUCCESS if successful, or XST_REGISTER_ERROR if a register did not read or write correctly

Note:

The priority level registers are restored after testing them in order to prevent problems with the registers being the same value after the test.

If the arbiter is in dynamic priority mode, this test changes the mode to fixed to ensure that the priority registers aren't changed by the arbiter during this test. The mode is restored to its entry value on exit.

```
XStatus XOpbArb_SetOptions( XOpbArb * InstancePtr,  
                           Xuint32   Options  
                           )
```

Sets the options for the OPB arbiter. The options control how the device grants the bus to requesting masters.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. See [xopbarb.h](#) for a description of the options.

Returns:

- XST_SUCCESS if options are successfully set.
- XST_NO_FEATURE if an attempt was made to enable dynamic priority arbitration

when the device is configured only for fixed priority arbitration, or an attempt was made to enable parking when bus parking is not supported by the device.

- XST_OPBARB_PARK_NOT_ENABLED if bus parking by park ID was enabled but bus parking itself was not enabled.

Note:

None.

```
XStatus XOpbArb_SetParkId( XOpbArb * InstancePtr,  
                           Xuint8      MasterId  
                           )
```

Sets the master ID used for bus parking. Bus parking must be enabled and the option to use bus parking by park ID must be set for this park ID to take effect (see the SetOptions function). If the option to use bus parking by park ID is set but this function is not called, bus parking defaults to master 0.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

MasterId is the ID of the master that will be parked if bus parking is enabled. This master's grant signal remains asserted as long as no other master requests the bus. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

XST_SUCCESS if the park ID is successfully set, or XST_NO_FEATURE if bus parking is not supported by the device.

Note:

None.

```
XStatus XOpbArb_SetPriorityLevel( XOpbArb * InstancePtr,  
                                  Xuint8      Level,  
                                  Xuint8      MasterId  
                                  )
```

Assigns a master ID to the given priority level. The use of priority levels by the device must be suspended before calling this function. Every master ID must be assigned to one and only one priority level. The driver enforces this before allowing use of priority levels by the device to be resumed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Level is the priority level being set. The level can range from 0 (highest) to N (lowest), where N is the number of masters minus one. The device currently supports up to 16 masters.

MasterId is the ID of the master being assigned to the priority level. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

- XST_SUCCESS if the slave is selected successfully.
- XST_OPBARB_NOT_SUSPENDED if priority levels have not been suspended. Before modifying the priority levels, use of priority levels by the device must be suspended.
- XST_OPBARB_NOT_FIXED_PRIORITY if the arbiter is in dynamic mode. It must be in fixed mode to modify the priority levels.

Note:

None.

```
void XOpbArb_SuspendPriorityLevels( XOpbArb * InstancePtr)
```

Suspends use of the priority levels by the device. Before modifying priority levels, the application must first suspend use of the levels by the device. This is to prevent possible OPB problems if no master is assigned a priority during the modification of priority levels. The application must resume use of priority levels by the device when all modifications are done. During the time priority levels are suspended, the device reverts to its default behavior of assigning priorities based on master IDs.

This function can be used when the device is configured for either fixed priority arbitration or dynamic priority arbitration. When used during dynamic priority arbitration, the application can configure the priority levels as a starting point for the LRU algorithm.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

None.

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

XOpbArb Struct Reference

```
#include <xopbarb.h>
```

Detailed Description

The XOpbArb driver instance data. The user is required to allocate a variable of this type for every OPB arbiter device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `opbarb/v1_02_a/src/xopbarb.h`
-

opbarb/v1_02_a/src/xopbarb.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xopbarb.h,v 1.1 2002/06/26 17:36:23 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file opbarb/v1_02_a/src/xopbarb.h
00026 *
00027 * This component contains the implementation of the XOpbArb component. It is
the
00028 * driver for the On-chip Peripheral Bus (OPB) Arbiter. The arbiter performs bus
00029 * arbitration for devices on the OPB
00030 *
00031 * <b>Priority Arbitration</b>
00032 *
00033 * By default, the arbiter prioritizes masters on the bus based on their master
00034 * IDs. A master's ID corresponds to the signals with which it connects to the
00035 * arbiter. Master 0 is the highest priority, master 1 the next highest, and so
00036 * on. The device driver allows an application to modify this default behavior.
00037 *
00038 * There are two modes of priority arbitration, dynamic and fixed. The device
00039 * can be parameterized in either of these modes. Fixed priority arbitration
00040 * makes use of priority levels that can be configured by software. There is
00041 * one level for each master on the bus. Priority level 0 is assigned the master
```

00042 * ID of the highest priority master, priority level 1 is assigned the master ID
00043 * of the next highest priority master, and so on.
00044 *
00045 * Dynamic priority arbitration utilizes a Least Recently Used(LRU) algorithm to
00046 * determine the priorities of masters on the bus. Once a master is granted the
00047 * bus, it falls to the lowest priority master. A master that is not granted the
00048 * bus moves up in priority until it becomes the highest priority master based
on
00049 * the fact that it has been the least recently used master. The arbiter uses
the
00050 * currently assigned priority levels as its starting configuration for the LRU
00051 * algorithm. Software can modify this starting configuration by assigning
00052 * master IDs to the priority levels.
00053 *
00054 * When configuring priority levels (i.e., assigning master IDs to priority
00055 * levels), the application must suspend use of the priority levels by the
00056 * device. Every master must be represented by one and only one priority level.
00057 * The device driver enforces this by making the application suspend use of the
00058 * priority levels by the device during the time it takes to correctly configure
00059 * the levels. Once the levels are configured, the application must explicitly
00060 * resume use of the priority levels by the device. During the time priority
00061 * levels are suspended, the device reverts to its default behavior of assigning
00062 * priorities based on master IDs.
00063 *
00064 * **Bus Parking**
00065 *
00066 * By default, bus parking is disabled. The device driver allows an application
00067 * to enable bus parking, which forces the arbiter to always assert the grant
00068 * signal for a specific master when no other masters are requesting the bus.
00069 * The master chosen for parking is either the master that was last granted the
00070 * bus, or the master configured by the SetParkId function. When bus parking
00071 * is enabled but no park ID has been set, bus parking defaults to the master
00072 * that was last granted the bus.
00073 *
00074 * **Device Configuration**
00075 *
00076 * The device can be configured in various ways during the FPGA implementation
00077 * process. Configuration parameters are stored in xopbarb_g.c. A table is
00078 * defined where each entry contains configuration information for a device.
00079 * This information includes such things as the base address of the
00080 * memory-mapped device, the number of masters on the bus, and the priority
00081 * arbitration scheme.
00082 *
00083 * When the device is parameterized with only 1 master, or if the device is
00084 * parameterized without a slave interface, there are no registers accessible to
00085 * software and no configuration entry is available. In these configurations it
00086 * is likely that the driver will not be loaded or used by the application.
00087 * But in the off-chance that it is, it is assumed that no configuration
00088 * information for the arbiter is entered in the xopbarb_g.c table. If config
00089 * information were entered for the device, there will be nothing to prevent
00090 * the driver's use, and any use of the driver under these circumstances will
00091 * result in undefined behavior.
00092 *

```

00093 * @note
00094 *
00095 * This driver is not thread-safe. Thread safety must be guaranteed by the layer
00096 * above this driver if there is a need to access the device from multiple
00097 * threads.
00098 *
00099 * <pre>
00100 * MODIFICATION HISTORY:
00101 *
00102 * Ver    Who    Date      Changes
00103 * -----
00104 * 1.02a rpm  08/13/01 First release
00105 * </pre>
00106 *
00107 *****/
00108
00109 #ifndef XOPBARB_H /* prevent circular inclusions */
00110 #define XOPBARB_H /* by using protection macros */
00111
00112 /****** Include Files *****/
00113
00114 #include "xbasic_types.h"
00115 #include "xstatus.h"
00116 #include "xopbarb_l.h"
00117
00118 /****** Constant Definitions *****/
00119
00120 /** @name Configuration options
00121  * @{
00122  */
00123 /**
00124  * The options enable or disable additional features of the OPB Arbiter. Each
00125  * of the options are bit fields such that more than one may be specified.
00126  *
00127  * - XOA_DYNAMIC_PRIORITY_OPTION
00128  * <br>
00129  * The Dynamic Priority option configures the device for dynamic priority
00130  * arbitration, which uses a Least Recently Used (LRU) algorithm to determine
00131  * the priorities of OPB masters. This option is not applicable if the device
00132  * is parameterized for fixed priority arbitration. When the device is
00133  * parameterized for dynamic priority arbitration, it can still use a fixed
00134  * priority arbitration by turning this option off. Fixed priority arbitration
00135  * uses the priority levels as written by software to determine the priorities
00136  * of OPB masters. The default is fixed priority arbitration.
00137  *
00138  * - XOA_PARK_ENABLE_OPTION
00139  * <br>
00140  * The Park Enable option enables bus parking, which forces the arbiter to
00141  * always assert the grant signal for a specific master when no other masters
00142  * are requesting the bus. The master chosen for parking is either the master
00143  * that was last granted the bus, or the master configured by the SetParkId
00144  * function.

```

```

00145 *
00146 * - XOA_PARK_BY_ID_OPTION
00147 * <br>
00148 * The Park By ID option enables bus parking based on the specific master ID.
00149 * The master ID defaults to master 0, and can be changed using
00150 * XOpbArb_SetParkId(). When this option is disabled, bus parking defaults to
00151 * the master that was last granted the bus. The park enable option must be set
00152 * for this option to take effect.
00153 */
00154 #define XOA_DYNAMIC_PRIORITY_OPTION    0x1
00155 #define XOA_PARK_ENABLE_OPTION        0x2
00156 #define XOA_PARK_BY_ID_OPTION         0x4
00157 /*@}*/
00158
00159 /***** Type Definitions *****/
00160
00161 /**
00162  * This typedef contains configuration information for the device.
00163  */
00164 typedef struct
00165 {
00166     Xuint16 DeviceId;           /**< Unique ID of device */
00167     Xuint32 BaseAddress;       /**< Register base address */
00168     Xuint8 NumMasters;         /**< Number of masters on the bus */
00169 } XOpbArb_Config;
00170
00171 /**
00172  * The XOpbArb driver instance data. The user is required to allocate a
00173  * variable of this type for every OPB arbiter device in the system. A pointer
00174  * to a variable of this type is then passed to the driver API functions.
00175  */
00176 typedef struct
00177 {
00178     Xuint32 BaseAddress;       /* Base address of registers */
00179     Xuint32 IsReady;           /* Device is initialized and ready */
00180     Xuint8 NumMasters;         /* Number of masters on this bus */
00181 } XOpbArb;
00182
00183 /***** Macros (Inline Functions) Definitions *****/
00184
00185
00186 /***** Function Prototypes *****/
00187
00188 /*
00189  * Initialization
00190  */
00191 XStatus XOpbArb_Initialize(XOpbArb *InstancePtr, Xuint16 DeviceId);
00192 XOpbArb_Config *XOpbArb_LookupConfig(Xuint16 DeviceId);
00193
00194 /*
00195  * Diagnostics

```

```
00196  */
00197 XStatus XOpbArb_SelfTest(XOpbArb *InstancePtr);
00198
00199 /*
00200  * Configuration
00201  */
00202 XStatus XOpbArb_SetOptions(XOpbArb *InstancePtr, Xuint32 Options);
00203 Xuint32 XOpbArb_GetOptions(XOpbArb *InstancePtr);
00204 XStatus XOpbArb_SetPriorityLevel(XOpbArb *InstancePtr, Xuint8 Level,
00205                                 Xuint8 MasterId);
00206 Xuint8 XOpbArb_GetPriorityLevel(XOpbArb *InstancePtr, Xuint8 Level);
00207
00208 void XOpbArb_SuspendPriorityLevels(XOpbArb *InstancePtr);
00209 XStatus XOpbArb_ResumePriorityLevels(XOpbArb *InstancePtr);
00210
00211 XStatus XOpbArb_SetParkId(XOpbArb *InstancePtr, Xuint8 MasterId);
00212 XStatus XOpbArb_GetParkId(XOpbArb *InstancePtr, Xuint8 *MasterIdPtr);
00213
00214
00215 #endif /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

opbarb/v1_02_a/src/xopbarb_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xopbarb.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XOpbArb_mSetControlReg(BaseAddress, RegisterValue)  
#define XOpbArb_mGetControlReg(BaseAddress)  
#define XOpbArb_mEnableDynamic(BaseAddress)  
#define XOpbArb_mDisableDynamic(BaseAddress)  
#define XOpbArb_mEnableParking(BaseAddress)
```



```
#define XOpbArb_mDisableParking(BaseAddress)
#define XOpbArb_mSetParkMasterNot(BaseAddress)
#define XOpbArb_mClearParkMasterNot(BaseAddress)
#define XOpbArb_mSetPriorityRegsValid(BaseAddress)
#define XOpbArb_mClearPriorityRegsValid(BaseAddress)
#define XOpbArb_mSetParkedMasterId(BaseAddress, ParkedMasterId)
#define XOpbArb_mSetPriorityReg(BaseAddress, Level, MasterId)
#define XOpbArb_mGetPriorityReg(BaseAddress, Level)
```

Define Documentation

```
#define XOpbArb_mClearParkMasterNot( BaseAddress )
```

Clear park on master not last (park on a specific master ID) in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mClearPriorityRegsValid( BaseAddress )
```

Clear the priority registers valid in the Control Register of the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

#define XOpbArb_mDisableDynamic(BaseAddress)

Disable dynamic priority arbitration in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

#define XOpbArb_mDisableParking(BaseAddress)

Disable parking in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

#define XOpbArb_mEnableDynamic(BaseAddress)

Enable dynamic priority arbitration in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mEnableParking( BaseAddress )
```

Enable parking in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mGetControlReg( BaseAddress )
```

Get the Control Register of the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XOpbArb_mGetPriorityReg( BaseAddress,  
                                Level          )
```

Get the priority register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Level contain the priority level of the register to get (0 - 15).

Returns:

The contents of the specified priority register, a master ID (0 - 15).

Note:

None.

```
#define XOpbArb_mSetControlReg( BaseAddress,  
                               RegisterValue )
```

Set the Control Register of the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

RegisterValue contains the value to be written to the register.

Returns:

None.

Note:

None.

```
#define XOpbArb_mSetParkedMasterId( BaseAddress,  
                                     ParkedMasterId )
```

Set the parked master ID in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

ParkedMasterId contains the ID of the master to park on (0 - 15).

Returns:

None.

Note:

None.

```
#define XOpbArb_mSetParkMasterNot( BaseAddress )
```

Set park on master not last (park on a specific master ID) in the Control Register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XOpbArb_mSetPriorityReg( BaseAddress,  
                                Level,  
                                MasterId )
```

Set the priority register in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Level contain the priority level of the register to set (0 - 15).

MasterId contains the value to be written to the register (0 - 15).

Returns:

None.

Note:

None.

```
#define XOpbArb_mSetPriorityRegsValid( BaseAddress )
```

Set the priority registers valid in the OPB Arbiter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

opbarb/v1_02_a/src/xopbarb_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xopbarb_l.h,v 1.1 2002/06/26 17:36:23 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file opbarb/v1_02_a/src/xopbarb_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  The user should refer to the
00029 * hardware device specification for more details of the device operation.
00030 * High-level driver functions are defined in xopbarb.h.
00031 *
00032 * <pre>
00033 * MODIFICATION HISTORY:
00034 *
00035 * Ver   Who   Date       Changes
00036 * ----  ---  -
00037 * 1.00b jhl   04/24/02  First release
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XOPBARB_L_H /* prevent circular inclusions */
```

```

00043 #define XOPBARB_L_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048 #include "xio.h"
00049
00050 /***** Constant Definitions *****/
00051
00052 /*
00053  * Register offsets for the OPB Arbiter
00054  */
00055 #define XOA_CR_OFFSET      0x100    /* (CR) Control Register */
00056 #define XOA_LVLX_OFFSET   0x104    /* Start of priority level registers */
00057 #define XOA_LVLX_SIZE     4        /* Size of priority level registers */
00058
00059 /*
00060  * OPB Arbiter Control Register (CR) masks
00061  */
00062 #define XOA_CR_DYNAMIC_ENABLE_MASK  0x80000000 /* Dynamic priority enable */
00063 #define XOA_CR_DYNAMIC_RW_MASK     0x40000000 /* Dynamic priority enable
00064                                           * read/write */
00065 #define XOA_CR_PARK_ENABLE_MASK     0x20000000 /* Park enable */
00066 #define XOA_CR_PARK_RW_MASK         0x10000000 /* Park enable read/write */
00067 #define XOA_CR_PARK_ON_ID_MASK      0x08000000 /* Park on park ID */
00068 #define XOA_CR_PRIORITY_VALID_MASK  0x04000000 /* Priority registers valid */
00069 #define XOA_CR_PARK_ID_MASK         0x0000000F /* Park ID */
00070
00071
00072 /***** Type Definitions *****/
00073
00074
00075 /***** Macros (Inline Functions) Definitions *****/
00076
00077
00078 /*****
00079  *
00080  * Low-level driver macros.  The list below provides signatures to help the
00081  * user use the macros.
00082  *
00083  * void XOpbArb_mSetControlReg(Xuint32 BaseAddress, Xuint32 RegisterValue)
00084  * Xuint32 XOpbArb_mGetControlReg(Xuint32 BaseAddress)
00085  *
00086  * void XOpbArb_mEnableDynamic(Xuint32 BaseAddress)
00087  * void XOpbArb_mDisableDynamic(Xuint32 BaseAddress)
00088  *
00089  * void XOpbArb_mEnableParking(Xuint32 BaseAddress)
00090  * void XOpbArb_mDisableParking(Xuint32 BaseAddress)
00091  *
00092  * void XOpbArb_mSetParkMasterNot(Xuint32 BaseAddress)
00093  * void XOpbArb_mClearParkOnMaster(Xuint32 BaseAddress)
00094  *

```



```

00095 * void XOpbArb_mSetPriorityRegsValid(Xuint32 BaseAddress)
00096 * void XOpbArb_mClearPriorityRegsValid(Xuint32 BaseAddress)
00097 *
00098 * void XOpbArb_mSetParkedMasterId(Xuint32 BaseAddress, Xuint8 ParkedMasterId)
00099 * void XOpbArb_mSetPriorityReg(Xuint32 BaseAddress, Xuint8 Level,
00100 *                               Xuint32 MasterId)
00101 * Xuint32 XOpbArb_mGetPriorityReg(Xuint32 BaseAddress, Xuint8 Level)
00102 *
00103 *****/
00104
00105 /***/
00106 /**
00107 * Set the Control Register of the OPB Arbiter.
00108 *
00109 * @param BaseAddress contains the base address of the device.
00110 * @param RegisterValue contains the value to be written to the register.
00111 *
00112 * @return None.
00113 *
00114 * @note None.
00115 *
00116 *****/
00117 #define XOpbArb_mSetControlReg(BaseAddress, RegisterValue) \
00118     XIo_Out32((BaseAddress) + XOA_CR_OFFSET, (RegisterValue))
00119
00120 *****/
00121 /**
00122 * Get the Control Register of the OPB Arbiter.
00123 *
00124 * @param BaseAddress contains the base address of the device.
00125 *
00126 * @return The value read from the register.
00127 *
00128 * @note None.
00129 *
00130 *****/
00131 #define XOpbArb_mGetControlReg(BaseAddress) \
00132     XIo_In32((BaseAddress) + XOA_CR_OFFSET)
00133
00134 *****/
00135 /**
00136 * Enable dynamic priority arbitration in the Control Register in the OPB
00137 * Arbiter.
00138 *
00139 * @param BaseAddress contains the base address of the device.
00140 *
00141 * @return None.
00142 *
00143 * @note None.
00144 *
00145 *****/
00146 #define XOpbArb_mEnableDynamic(BaseAddress) \

```

```

00147     XIo_Out32((BaseAddress) + XOA_CR_OFFSET,          \
00148             (XIo_In32((BaseAddress) + XOA_CR_OFFSET) |          \
00149             XOA_CR_DYNAMIC_ENABLE_MASK))
00150
00151 /*(*****/
00152 /**
00153 * Disable dynamic priority arbitration in the Control Register in the OPB
00154 * Arbiter.
00155 *
00156 * @param   BaseAddress contains the base address of the device.
00157 *
00158 * @return  None.
00159 *
00160 * @note    None.
00161 *
00162 *****/
00163 #define XOpbArb_mDisableDynamic(BaseAddress)          \
00164     XIo_Out32((BaseAddress) + XOA_CR_OFFSET,          \
00165             (XIo_In32((BaseAddress) + XOA_CR_OFFSET) &          \
00166             ~XOA_CR_DYNAMIC_ENABLE_MASK))
00167
00168 /*(*****/
00169 /**
00170 * Enable parking in the Control Register in the OPB Arbiter.
00171 *
00172 * @param   BaseAddress contains the base address of the device.
00173 *
00174 * @return  None.
00175 *
00176 * @note    None.
00177 *
00178 *****/
00179 #define XOpbArb_mEnableParking(BaseAddress)          \
00180     XIo_Out32((BaseAddress) + XOA_CR_OFFSET,          \
00181             (XIo_In32((BaseAddress) + XOA_CR_OFFSET) |          \
00182             XOA_CR_PARK_ENABLE_MASK))
00183
00184 /*(*****/
00185 /**
00186 * Disable parking in the Control Register in the OPB Arbiter.
00187 *
00188 * @param   BaseAddress contains the base address of the device.
00189 *
00190 * @return  None.
00191 *
00192 * @note    None.
00193 *
00194 *****/
00195 #define XOpbArb_mDisableParking(BaseAddress)          \
00196     XIo_Out32((BaseAddress) + XOA_CR_OFFSET,          \
00197             (XIo_In32((BaseAddress) + XOA_CR_OFFSET) &          \
00198             ~XOA_CR_PARK_ENABLE_MASK))

```

```

00199
00200 /*****
00201 /**
00202 * Set park on master not last (park on a specific master ID) in the Control
00203 * Register in the OPB Arbiter.
00204 *
00205 * @param   BaseAddress contains the base address of the device.
00206 *
00207 * @return  None.
00208 *
00209 * @note    None.
00210 *
00211 *****/
00212 #define XOpbArb_mSetParkMasterNot(BaseAddress)          \
00213     XIo_Out32((BaseAddress) + XOA_CR_OFFSET,           \
00214             (XIo_In32((BaseAddress) + XOA_CR_OFFSET) | \
00215             XOA_CR_PARK_ON_ID_MASK))
00216
00217 /*****
00218 /**
00219 * Clear park on master not last (park on a specific master ID) in the OPB
00220 * Arbiter.
00221 *
00222 * @param   BaseAddress contains the base address of the device.
00223 *
00224 * @return  None.
00225 *
00226 * @note    None.
00227 *
00228 *****/
00229 #define XOpbArb_mClearParkMasterNot(BaseAddress)        \
00230     XIo_Out32((BaseAddress) + XOA_CR_OFFSET,           \
00231             (XIo_In32((BaseAddress) + XOA_CR_OFFSET) & \
00232             ~XOA_CR_PARK_ON_ID_MASK))
00233
00234 /*****
00235 /**
00236 * Set the priority registers valid in the OPB Arbiter.
00237 *
00238 * @param   BaseAddress contains the base address of the device.
00239 *
00240 * @return  None.
00241 *
00242 * @note    None.
00243 *
00244 *****/
00245 #define XOpbArb_mSetPriorityRegsValid(BaseAddress)      \
00246     XIo_Out32((BaseAddress) + XOA_CR_OFFSET,           \
00247             (XIo_In32((BaseAddress) + XOA_CR_OFFSET) | \
00248             XOA_CR_PRIORITY_VALID_MASK))
00249
00250 /*****

```

```

00251 /**
00252 * Clear the priority registers valid in the Control Register of the OPB
00253 * Arbiter.
00254 *
00255 * @param   BaseAddress contains the base address of the device.
00256 *
00257 * @return  None.
00258 *
00259 * @note    None.
00260 *
00261 *****/
00262 #define XOpbArb_mClearPriorityRegsValid(BaseAddress) \
00263     XIo_Out32((BaseAddress) + XOA_CR_OFFSET, \
00264             (XIo_In32((BaseAddress) + XOA_CR_OFFSET) & \
00265             ~XOA_CR_PRIORITY_VALID_MASK))
00266
00267 *****/
00268 /**
00269 * Set the parked master ID in the Control Register in the OPB Arbiter.
00270 *
00271 * @param   BaseAddress contains the base address of the device.
00272 * @param   ParkedMasterId contains the ID of the master to park on (0 - 15).
00273 *
00274 * @return  None.
00275 *
00276 * @note    None.
00277 *
00278 *****/
00279 #define XOpbArb_mSetParkedMasterId(BaseAddress, ParkedMasterId) \
00280 { \
00281     Xuint32 ControlReg; \
00282 \
00283     ControlReg = XIo_In32((BaseAddress) + XOA_CR_OFFSET) & \
00284                 ~XOA_CR_PARK_ID_MASK; \
00285     XIo_Out32((BaseAddress) + XOA_CR_OFFSET, ControlReg | \
00286             (Xuint8)ParkedMasterId); \
00287 }
00288
00289 *****/
00290 /**
00291 * Set the priority register in the OPB Arbiter.
00292 *
00293 * @param   BaseAddress contains the base address of the device.
00294 * @param   Level contain the priority level of the register to set (0 - 15).
00295 * @param   MasterId contains the value to be written to the register (0 - 15).
00296 *
00297 * @return  None.
00298 *
00299 * @note    None.
00300 *
00301 *****/
00302 #define XOpbArb_mSetPriorityReg(BaseAddress, Level, MasterId) \

```

```

00303     XIo_Out32((BaseAddress) + XOA_LVLX_OFFSET + (Level * XOA_LVLX_SIZE), \
00304             MasterId)
00305
00306 /*****
00307 /**
00308 * Get the priority register in the OPB Arbiter.
00309 *
00310 * @param   BaseAddress contains the base address of the device.
00311 * @param   Level contain the priority level of the register to get (0 - 15).
00312 *
00313 * @return  The contents of the specified priority register, a master ID
00314 *          (0 - 15).
00315 *
00316 * @note    None.
00317 *
00318 *****/
00319 #define XOpbArb_mGetPriorityReg(BaseAddress, Level) \
00320     XIo_In32((BaseAddress) + XOA_LVLX_OFFSET + (Level * XOA_LVLX_SIZE))
00321
00322 /***** Function Prototypes *****/
00323
00324
00325 /***** Variable Definitions *****/
00326
00327 #endif          /* end of protection macro */
00328

```

Generated on 29 May 2003 for Xilinx Device Drivers

XOpbArb_Config Struct Reference

```
#include <xopbarb.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xuint8 NumMasters

Field Documentation

Xuint32 XOpbArb_Config::BaseAddress

Register base address

Xuint16 XOpbArb_Config::DeviceId

Unique ID of device

Xuint8 XOpbArb_Config::NumMasters

Number of masters on the bus

The documentation for this struct was generated from the following file:

- [opbarb/v1_02_a/src/xopbarb.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

opbarb/v1_02_a/src/xopbarb.c File Reference

Detailed Description

This component contains the implementation of the **XOpbArb** driver component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.02a	rpm	08/13/01	First release

```
#include "xparameters.h"  
#include "xopbarb.h"  
#include "xio.h"
```

Data Structures

```
struct OptionsMap
```

Functions

```
XStatus XOpbArb_Initialize (XOpbArb *InstancePtr, Xuint16 DeviceId)  
XStatus XOpbArb_SelfTest (XOpbArb *InstancePtr)  
XStatus XOpbArb_SetOptions (XOpbArb *InstancePtr, Xuint32 Options)
```


Xuint32 XOpbArb_GetOptions (XOpbArb *InstancePtr)

XStatus XOpbArb_SetPriorityLevel (XOpbArb *InstancePtr, **Xuint8** Level, **Xuint8** MasterId)

Xuint8 XOpbArb_GetPriorityLevel (XOpbArb *InstancePtr, **Xuint8** Level)

void XOpbArb_SuspendPriorityLevels (XOpbArb *InstancePtr)

XStatus XOpbArb_ResumePriorityLevels (XOpbArb *InstancePtr)

XStatus XOpbArb_SetParkId (XOpbArb *InstancePtr, **Xuint8** MasterId)

XStatus XOpbArb_GetParkId (XOpbArb *InstancePtr, **Xuint8** *MasterIdPtr)

XOpbArb_Config * XOpbArb_LookupConfig (**Xuint16** DeviceId)

Function Documentation

Xuint32 XOpbArb_GetOptions(XOpbArb * *InstancePtr*)

Gets the options for the arbiter. The options control how the device grants the bus to requesting masters.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

The options of the device. This is a bit mask where a 1 means the option is on, and a 0 means the option is off. See [xopbarb.h](#) for a description of the options.

Note:

None.

XStatus XOpbArb_GetParkId(XOpbArb * *InstancePtr*,
Xuint8 * *MasterIdPtr*
)

Gets the master ID currently used for bus parking.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

MasterIdPtr is a pointer to a byte that will hold the master ID currently used for bus parking. This is an output parameter. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

XST_SUCCESS if the park ID is successfully retrieved, or XST_NO_FEATURE if bus parking is not supported by the device.

Note:

None.

```
Xuint8 XOpbArb_GetPriorityLevel( XOpbArb * InstancePtr,  
                                Xuint8      Level  
                                )
```

Get the master ID at the given priority level.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Level is the priority level being retrieved. The level can range from 0 (highest) to N (lowest), where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

The master ID assigned to the given priority level. The ID can range from 0 to N, where N is the number of masters minus one.

Note:

If the arbiter is operating in dynamic priority mode, the value returned from this function may not be predictable because the arbiter changes the values on the fly.

```
XStatus XOpbArb_Initialize( XOpbArb * InstancePtr,  
                             Xuint16   DeviceId  
                             )
```

Initializes a specific **XOpbArb** instance. The driver is initialized to allow access to the device registers. In addition, the configuration information is retrieved for the device. Currently, configuration information is stored in **xopbarb_g.c**.

The state of the device after initialization is:

- Fixed or dynamic priority arbitration based on hardware parameter
- Bus parking is disabled

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XOpbArb** component. Passing in a device id associates the generic **XOpbArb** component to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful or XST_DEVICE_NOT_FOUND if no configuration data was found for this device.

Note:

None.

XOpbArb_Config* XOpbArb_LookupConfig(**Xuint16** *DeviceId*)

Looks up the device configuration based on the unique device ID. The table OpbArbConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to match on.

Returns:

A pointer to the configuration information for the matching device instance, or XNULL if no match is found.

Note:

None.

XStatus XOpbArb_ResumePriorityLevels(XOpbArb * InstancePtr)

Resumes use of the priority levels by the device. This function is typically called sometime after a call to SuspendPriorityLevels. The application must resume use of priority levels by the device when all modifications are done. If no call is made to this function after use of the priority levels has been suspended, the device will remain in its default priority arbitration mode of assigning priorities based on master IDs. A call to this function has no effect if no prior call was made to suspend the use of priority levels.

Every master must be represented by one and only one fixed priority level before the use of priority levels can be resumed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

- XST_SUCCESS if the slave is selected successfully.
- XST_OPBARB_INVALID_PRIORITY if there is either a master that is not assigned a priority level, or a master that is assigned two or more priority levels.

Note:

None.

XStatus XOpbArb_SelfTest(XOpbArb * InstancePtr)

Runs a self-test on the driver/device. The self-test simply verifies that the arbiter's registers can be read and written. This is an intrusive test in that the arbiter will not be using the priority registers while the test is being performed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

XST_SUCCESS if successful, or XST_REGISTER_ERROR if a register did not read or write correctly

Note:

The priority level registers are restored after testing them in order to prevent problems with the registers being the same value after the test.

If the arbiter is in dynamic priority mode, this test changes the mode to fixed to ensure that the

priority registers aren't changed by the arbiter during this test. The mode is restored to its entry value on exit.

```
XStatus XOpbArb_SetOptions( XOpbArb * InstancePtr,  
                             Xuint32   Options  
                             )
```

Sets the options for the OPB arbiter. The options control how the device grants the bus to requesting masters.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. See [xopbarb.h](#) for a description of the options.

Returns:

- XST_SUCCESS if options are successfully set.
- XST_NO_FEATURE if an attempt was made to enable dynamic priority arbitration when the device is configured only for fixed priority arbitration, or an attempt was made to enable parking when bus parking is not supported by the device.
- XST_OPBARB_PARK_NOT_ENABLED if bus parking by park ID was enabled but bus parking itself was not enabled.

Note:

None.

```
XStatus XOpbArb_SetParkId( XOpbArb * InstancePtr,  
                             Xuint8   MasterId  
                             )
```

Sets the master ID used for bus parking. Bus parking must be enabled and the option to use bus parking by park ID must be set for this park ID to take effect (see the SetOptions function). If the option to use bus parking by park ID is set but this function is not called, bus parking defaults to master 0.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

MasterId is the ID of the master that will be parked if bus parking is enabled. This master's grant signal remains asserted as long as no other master requests the bus. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

XST_SUCCESS if the park ID is successfully set, or XST_NO_FEATURE if bus parking is not supported by the device.

Note:

None.

```
XStatus XOpbArb_SetPriorityLevel( XOpbArb * InstancePtr,  
                                Xuint8      Level,  
                                Xuint8      MasterId  
                                )
```

Assigns a master ID to the given priority level. The use of priority levels by the device must be suspended before calling this function. Every master ID must be assigned to one and only one priority level. The driver enforces this before allowing use of priority levels by the device to be resumed.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Level is the priority level being set. The level can range from 0 (highest) to N (lowest), where N is the number of masters minus one. The device currently supports up to 16 masters.

MasterId is the ID of the master being assigned to the priority level. The ID can range from 0 to N, where N is the number of masters minus one. The device currently supports up to 16 masters.

Returns:

- XST_SUCCESS if the slave is selected successfully.
- XST_OPBARB_NOT_SUSPENDED if priority levels have not been suspended.

Before modifying the priority levels, use of priority levels by the device must be suspended.

- XST_OPBARB_NOT_FIXED_PRIORITY if the arbiter is in dynamic mode. It must be in fixed mode to modify the priority levels.

Note:

None.

```
void XOpbArb_SuspendPriorityLevels( XOpbArb * InstancePtr)
```

Suspends use of the priority levels by the device. Before modifying priority levels, the application must first suspend use of the levels by the device. This is to prevent possible OPB problems if no master is assigned a priority during the modification of priority levels. The application must resume use of priority levels by the device when all modifications are done. During the time priority levels are suspended, the device reverts to its default behavior of assigning priorities based on master IDs.

This function can be used when the device is configured for either fixed priority arbitration or dynamic priority arbitration. When used during dynamic priority arbitration, the application can configure the priority levels as a starting point for the LRU algorithm.

Parameters:

InstancePtr is a pointer to the **XOpbArb** instance to be worked on.

Returns:

None.

Note:

None.

opbarb/v1_02_a/src/xopbarb_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of OPB arbiter devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.02a	rpm	08/13/01	First release

```
#include "xopbarb.h"  
#include "xparameters.h"
```

Variables

[XOpbArb_Config](#) [XOpbArb_ConfigTable](#) [XPAR_XOPBARB_NUM_INSTANCES]

Variable Documentation

[XOpbArb_Config](#) [XOpbArb_ConfigTable](#)[XPAR_XOPBARB_NUM_INSTANCES]

The OPB arbiter configuration table, sized by the number of instances defined in [xparameters.h](#).

Generated on 29 May 2003 for Xilinx Device Drivers

plb2opb/v1_00_a/src/xplb2opb.h File Reference

Detailed Description

This component contains the implementation of the **XPlb2Opb** component. It is the driver for the PLB to OPB Bridge. The bridge converts PLB bus transactions to OPB bus transactions. The hardware acts as a slave on the PLB side and as a master on the OPB side. This interface is necessary for the processor to access OPB based peripherals.

This driver allows the user to access the Bridge registers to support the handling of bus errors and other access errors and determine an appropriate solution if possible.

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

The Xilinx PLB to OPB Bridge is a soft IP core designed for Xilinx FPGAs and contains the following features:

- PLB Slave interface
- 32-bit or 64-bit PLB Slave (configurable via a design parameter)
- Communicates with 32- or 64-bit PLB masters
- Non-burst transfers of 1-8 bytes
- Burst transfers, including word and double-word bursts of fixed or variable lengths, up to depth of burst buffer. Buffer depth configurable via a design parameter
- Limited support for byte, half-word, quad-word and octal-word bursts to maintain PLB compliance
- Cacheline transactions of 4, 8, and 16 words
- Support for transactions not utilized by the PPC405 Core can be eliminated via a design parameter
- PPC405 Core only utilizes single beat, 4, 8, or 16 word line transfers support for burst transactions can be eliminated via a design parameter
- Supports up to 8 PLB masters (number of PLB masters configurable via a design parameter)
- Programmable lower and upper address boundaries
- OPB Master interface with byte enable transfers *Note*: Does not support dynamic bus sizing without additional glue logic
- Data width configurable via a design parameter
- PLB and OPB clocks can have a 1:1, 1:2, 1:4 synchronous relationship
- Bus Error Address Registers (BEAR) and Bus Error Status Registers (BESR) to report errors
- DCR Slave interface provides access to BEAR/BESR

- BEAR, BESR, and DCR interface can be removed from the design via a design parameter
- Posted write buffer. Buffer depth configurable via a design parameter

Device Configuration

The device can be configured in various ways during the FPGA implementation process. The current configuration data contained in [xplb2opb_g.c](#). A table is defined where each entry contains configuration information for device. This information includes such things as the base address of the DCR mapped device, and the number of masters on the bus.

Note:

This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads.

The Bridge registers reside on the DCR address bus.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	12/7/01	First release
1.00a	rpm	05/14/02	Made configuration typedef/lookup public

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xplb2opb_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XPlb2Opb
struct XPlb2Opb_Config
```

PLB-OPB bridge error status masks

```
#define XP20_DRIVING_BEAR_MASK
#define XP20_ERROR_READ_MASK
#define XP20_ERROR_TYPE_MASK
#define XP20_LOCK_ERR_MASK
```

Functions

XStatus **XPlb2Opb_Initialize** (**XPlb2Opb** *InstancePtr, **Xuint16** DeviceId)

void **XPlb2Opb_Reset** (**XPlb2Opb** *InstancePtr)

XPlb2Opb_Config * **XPlb2Opb_LookupConfig** (**Xuint16** DeviceId)

Xboolean **XPlb2Opb_IsError** (**XPlb2Opb** *InstancePtr)

void **XPlb2Opb_ClearErrors** (**XPlb2Opb** *InstancePtr, **Xuint8** Master)

Xuint32 **XPlb2Opb_GetErrorStatus** (**XPlb2Opb** *InstancePtr, **Xuint8** Master)

Xuint32 **XPlb2Opb_GetErrorAddress** (**XPlb2Opb** *InstancePtr)

Xuint32 **XPlb2Opb_GetErrorByteEnables** (**XPlb2Opb** *InstancePtr)

Xuint8 **XPlb2Opb_GetMasterDrivingError** (**XPlb2Opb** *InstancePtr)

Xuint8 **XPlb2Opb_GetNumMasters** (**XPlb2Opb** *InstancePtr)

void **XPlb2Opb_EnableInterrupt** (**XPlb2Opb** *InstancePtr)

void **XPlb2Opb_DisableInterrupt** (**XPlb2Opb** *InstancePtr)

XStatus **XPlb2Opb_SelfTest** (**XPlb2Opb** *InstancePtr, **Xuint32** TestAddress)

Define Documentation

```
#define XP2O_DRIVING_BEAR_MASK
```

XP2O_DRIVING_BEAR_MASK

Indicates this master is driving the outstanding error

XP2O_ERROR_READ_MASK

Indicates the error is a read error. It is a write error otherwise.

XP2O_ERROR_TYPE_MASK

If set, the error was a timeout. Otherwise the error was an error acknowledge

XP2O_LOCK_ERR_MASK

Indicates the error is locked and cannot be overwritten.

```
#define XP2O_ERROR_READ_MASK
```

XP20_DRIVING_BEAR_MASK	Indicates this master is driving the outstanding error
XP20_ERROR_READ_MASK	Indicates the error is a read error. It is a write error otherwise.
XP20_ERROR_TYPE_MASK	If set, the error was a timeout. Otherwise the error was an error acknowledge
XP20_LOCK_ERR_MASK	Indicates the error is locked and cannot be overwritten.

#define XP20_ERROR_TYPE_MASK

XP20_DRIVING_BEAR_MASK	Indicates this master is driving the outstanding error
XP20_ERROR_READ_MASK	Indicates the error is a read error. It is a write error otherwise.
XP20_ERROR_TYPE_MASK	If set, the error was a timeout. Otherwise the error was an error acknowledge
XP20_LOCK_ERR_MASK	Indicates the error is locked and cannot be overwritten.

#define XP20_LOCK_ERR_MASK

XP20_DRIVING_BEAR_MASK	Indicates this master is driving the outstanding error
XP20_ERROR_READ_MASK	Indicates the error is a read error. It is a write error otherwise.
XP20_ERROR_TYPE_MASK	If set, the error was a timeout. Otherwise the error was an error acknowledge
XP20_LOCK_ERR_MASK	Indicates the error is locked and cannot be overwritten.

Function Documentation

```
void XPlb2Opb_ClearErrors( XPlb2Opb * InstancePtr,
                          Xuint8   Master
                          )
```

Clears any outstanding errors for the given master.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

None.

Note:

None.

```
void XPlb2Opb_DisableInterrupt( XPlb2Opb * InstancePtr)
```

Disables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void XPlb2Opb_EnableInterrupt( XPlb2Opb * InstancePtr)
```

Enables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the

necessary actions to correct the situation.

Xuint32 XPlb2Opb_GetErrorAddress(XPlb2Opb * InstancePtr)

Returns the OPB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XPlb2Opb_IsError()** is recommended to confirm that an error has occurred prior to calling **XPlb2Opb_GetErrorAddress()** to ensure that the data in the error address register is relevant.

Xuint32 XPlb2Opb_GetErrorByteEnables(XPlb2Opb * InstancePtr)

Returns the byte-enables asserted during the access causing the error. The enables are parameters in the hardware making the return value dynamic. An example of a 32-bit bus with all 4 byte enables available, **XPlb2Opb_GetErrorByteEnables** will have the value 0xF0000000 returned from a 32-bit access error.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The byte-enables asserted during the error causing access.

Note:

None.

Xuint32 XPlb2Opb_GetErrorStatus(XPlb2Opb * InstancePtr, Xuint8 Master)

Returns the error status for the specified master.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

The current error status for the requested master on the PLB. The status is a bit-mask and the values are described in **xplb2opb.h**.

Note:

None.

Xuint8 XPlb2Opb_GetMasterDrivingError(XPlb2Opb * InstancePtr)

Returns the ID of the master which is driving the error condition

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The ID of the master that is driving the error

Note:

None.

Xuint8 XPlb2Opb_GetNumMasters(XPlb2Opb * InstancePtr)

Returns the number of masters associated with the provided instance

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The number of masters. This is a number from 1 to the maximum of 32.

Note:

The value returned from this call needs to be adjusted if it is to be used as the argument for other calls since the masters are numbered from 0 and this function returns values starting at 1.


```
XStatus XPlb2Opb_Initialize( XPlb2Opb * InstancePtr,  
                             Xuint16      DeviceId  
                             )
```

Initializes a specific **XPlb2Opb** instance. Looks for configuration data for the specified device, then initializes instance data.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPlb2Opb** component. Passing in a device id associates the generic **XPlb2Opb** component to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
Xboolean XPlb2Opb_IsError( XPlb2Opb * InstancePtr)
```

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

```
XPlb2Opb_Config* XPlb2Opb_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID. The table `PlbOpbConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for

Returns:

A pointer to the configuration data for the given device, or `XNULL` if no match is found.

Note:

None.

```
void XPlb2Opb_Reset( XPlb2Opb * InstancePtr)
```

Forces a software-induced reset to occur in the bridge. Disables interrupts in the process.

Parameters:

InstancePtr is a pointer to the `XPlb2Opb` instance to be worked on.

Returns:

None.

Note:

Disables interrupts in the process.

```
XStatus XPlb2Opb_SelfTest( XPlb2Opb * InstancePtr,  
                          Xuint32   TestAddress  
                          )
```

Runs a self-test on the driver/device.

This tests reads the BCR to verify that the proper value is there.

`XST_SUCCESS` is returned if expected value is there, `XST_PLB2OPB_FAIL_SELFTEST` is returned otherwise.

Parameters:

InstancePtr is a pointer to the `XPlb2Opb` instance to be worked on.

TestAddress is a location that could cause an error on read, not used - user definable for hw specific implementations.

Returns:

`XST_SUCCESS` if successful, or `XST_PLB2OPB_FAIL_SELFTEST` if the driver fails self-test.

Note:

This test assumes that the bus error interrupts are not enabled.

Generated on 29 May 2003 for Xilinx Device Drivers

XPlb2Opb Struct Reference

```
#include <xplb2opb.h>
```

Detailed Description

The XPlb2Opb driver instance data. The user is required to allocate a variable of this type for every PLB-to_OPB bridge device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [plb2opb/v1_00_a/src/xplb2opb.h](#)

plb2opb/v1_00_a/src/xplb2opb.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xplb2opb.h,v 1.2 2002/07/26 20:19:19 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file plb2opb/v1_00_a/src/xplb2opb.h
00026 *
00027 * This component contains the implementation of the XPlb2Opb component. It is
00028 * the driver for the PLB to OPB Bridge. The bridge converts PLB bus
transactions
00029 * to OPB bus transactions. The hardware acts as a slave on the PLB side and as
00030 * a master on the OPB side. This interface is necessary for the processor to
00031 * access OPB based peripherals.
00032 *
00033 * This driver allows the user to access the Bridge registers to support
00034 * the handling of bus errors and other access errors and determine an
00035 * appropriate solution if possible.
00036 *
00037 * The bridge hardware generates interrupts in error conditions. These
interrupts
00038 * are not handled by the driver directly. It is the application's
responsibility
00039 * to attach to the appropriate interrupt with a handler which then calls
```

```
00040 * functions provided by this driver to determine the cause of the error and
take
00041 * the necessary actions to correct the situation.
00042 *
00043 * The Xilinx PLB to OPB Bridge is a soft IP core designed for Xilinx FPGAs and
00044 * contains the following features:
00045 *     - PLB Slave interface
00046 *     - 32-bit or 64-bit PLB Slave (configurable via a design parameter)
00047 *     - Communicates with 32- or 64-bit PLB masters
00048 *     - Non-burst transfers of 1-8 bytes
00049 *     - Burst transfers, including word and double-word bursts of fixed or
variable
00050 *         lengths, up to depth of burst buffer. Buffer depth configurable via a
00051 *         design parameter
00052 *     - Limited support for byte, half-word, quad-word and octal-word bursts to
00053 *         maintain PLB compliance
00054 *     - Cacheline transactions of 4, 8, and 16 words
00055 *     - Support for transactions not utilized by the PPC405 Core can be
eliminated
00056 *         via a design parameter
00057 *     - PPC405 Core only utilizes single beat, 4, 8, or 16 word line transfers
00058 *         support for burst transactions can be eliminated via a design parameter
00059 *     - Supports up to 8 PLB masters (number of PLB masters configurable via a
00060 *         design parameter)
00061 *     - Programmable lower and upper address boundaries
00062 *     - OPB Master interface with byte enable transfers
00063 *         <i>Note</i>: Does not support dynamic bus sizing without additional
glue logic
00064 *     - Data width configurable via a design parameter
00065 *     - PLB and OPB clocks can have a 1:1, 1:2, 1:4 synchronous relationship
00066 *     - Bus Error Address Registers (BEAR) and Bus Error Status Registers (BESR)
00067 *         to report errors
00068 *     - DCR Slave interface provides access to BEAR/BESR
00069 *     - BEAR, BESR, and DCR interface can be removed from the design via a design
00070 *         parameter
00071 *     - Posted write buffer. Buffer depth configurable via a design parameter
00072 *
00073 * <b>Device Configuration</b>
00074 *
00075 * The device can be configured in various ways during the FPGA implementation
00076 * process. The current configuration data contained in xplb2opb_g.c. A
00077 * table is defined where each entry contains configuration information for
00078 * device. This information includes such things as the base address of the DCR
00079 * mapped device, and the number of masters on the bus.
00080 *
00081 * @note
00082 *
00083 * This driver is not thread-safe. Thread safety must be guaranteed by the layer
00084 * above this driver if there is a need to access the device from multiple
00085 * threads.
00086 * <br><br>
00087 * The Bridge registers reside on the DCR address bus.
00088 *
```

```

00089 * <pre>
00090 * MODIFICATION HISTORY:
00091 *
00092 * Ver    Who    Date      Changes
00093 * -----
00094 * 1.00a ecm   12/7/01  First release
00095 * 1.00a rpm   05/14/02  Made configuration typedef/lookup public
00096 * </pre>
00097 *
00098 *****/
00099
00100 #ifndef XPLB2OPB_H /* prevent circular inclusions */
00101 #define XPLB2OPB_H /* by using protection macros */
00102
00103 /***** Include Files *****/
00104 #include "xbasic_types.h"
00105 #include "xstatus.h"
00106 #include "xplb2opb_1.h"
00107
00108 /***** Constant Definitions *****/
00109
00110
00111 /** @name PLB-OPB bridge error status masks
00112  * @{
00113  */
00114 /**
00115  * <pre>
00116  * XP20_DRIVING_BEAR_MASK          Indicates this master is driving the
00117  *                                outstanding error
00118  * XP20_ERROR_READ_MASK           Indicates the error is a read error. It is
00119  *                                a write error otherwise.
00120  * XP20_ERROR_TYPE_MASK           If set, the error was a timeout. Otherwise
00121  *                                the error was an error acknowledge
00122  * XP20_LOCK_ERR_MASK             Indicates the error is locked and cannot
00123  *                                be overwritten.
00124  * </pre>
00125  */
00126 #define XP20_DRIVING_BEAR_MASK          0x80000000UL
00127 #define XP20_ERROR_READ_MASK           0x40000000UL
00128 #define XP20_ERROR_TYPE_MASK           0x20000000UL
00129 #define XP20_LOCK_ERR_MASK             0x10000000UL
00130 /* @} */
00131
00132 /***** Type Definitions *****/
00133
00134 /**
00135  * This typedef contains configuration information for the device.
00136  */
00137 typedef struct
00138 {
00139     Xuint16 DeviceId;          /**< Unique ID of device */

```

```

00140     Xuint32 BaseAddress;    /**< Base address of device */
00141     Xuint8 NumMasters;      /**< Number of masters on the bus */
00142 } XPlb2Opb_Config;
00143
00144
00145 /**
00146  * The XPlb2Opb driver instance data. The user is required to allocate a
00147  * variable of this type for every PLB-to_OPB bridge device in the system.
00148  * A pointer to a variable of this type is then passed to the driver API
00149  * functions.
00150  */
00151 typedef struct
00152 {
00153     Xuint32 BaseAddress;      /* Base address of device */
00154     Xuint32 IsReady;         /* Device is initialized and ready */
00155     Xuint8 NumMasters;       /* number of masters for this bridge */
00156 } XPlb2Opb;
00157
00158
00159
00160
00161 /***** Macros (Inline Functions) Definitions *****/
00162
00163
00164 /***** Function Prototypes *****/
00165
00166
00167 /*
00168  * Required functions in xplb2opb.c
00169  */
00170
00171 /*
00172  * Initialization Functions
00173  */
00174 XStatus XPlb2Opb_Initialize(XPlb2Opb *InstancePtr, Xuint16 DeviceId);
00175 void XPlb2Opb_Reset(XPlb2Opb *InstancePtr);
00176 XPlb2Opb_Config *XPlb2Opb_LookupConfig(Xuint16 DeviceId);
00177
00178 /*
00179  * Access Functions
00180  */
00181
00182 Xboolean XPlb2Opb_IsError(XPlb2Opb *InstancePtr);
00183 void XPlb2Opb_ClearErrors(XPlb2Opb *InstancePtr, Xuint8 Master);
00184
00185 Xuint32 XPlb2Opb_GetErrorStatus(XPlb2Opb *InstancePtr, Xuint8 Master);
00186 Xuint32 XPlb2Opb_GetErrorAddress(XPlb2Opb *InstancePtr);
00187 Xuint32 XPlb2Opb_GetErrorByteEnables(XPlb2Opb *InstancePtr);
00188 Xuint8 XPlb2Opb_GetMasterDrivingError(XPlb2Opb *InstancePtr);
00189

```



```
00190 /*
00191  * Configuration
00192  */
00193 Xuint8 XPlb2Opb_GetNumMasters(XPlb2Opb *InstancePtr);
00194 void XPlb2Opb_EnableInterrupt(XPlb2Opb *InstancePtr);
00195 void XPlb2Opb_DisableInterrupt(XPlb2Opb *InstancePtr);
00196
00197
00198 /*
00199  * Self-test functions in xplb2opb_selftest.c
00200  */
00201 XStatus XPlb2Opb_SelfTest(XPlb2Opb *InstancePtr, Xuint32 TestAddress);
00202
00203 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

plb2opb/v1_00_a/src/xplb2opb_l.h File Reference

Detailed Description

This file contains identifiers and low-level macros that can be used to access the device directly. See [xplb2opb.h](#) for the high-level driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	rpm	05/10/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XPlb2Opb_mGetErrorDetectReg(BaseAddress)  
#define XPlb2Opb_mSetErrorDetectReg(BaseAddress, Mask)  
#define XPlb2Opb_mGetMasterDrivingReg(BaseAddress)  
#define XPlb2Opb_mGetReadWriteReg(BaseAddress)  
#define XPlb2Opb_mGetErrorTypeReg(BaseAddress)
```

```
#define XPlb2Opb_mGetLockBitReg(BaseAddress)
#define XPlb2Opb_mGetErrorAddressReg(BaseAddress)
#define XPlb2Opb_mGetByteEnableReg(BaseAddress)
#define XPlb2Opb_mSetControlReg(BaseAddress, Mask)
#define XPlb2Opb_mGetControlReg(BaseAddress)
```

Define Documentation

```
#define XPlb2Opb_mGetByteEnableReg( BaseAddress )
```

Get the error address byte enable register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit error address byte enable register contents.

Note:

None.

```
#define XPlb2Opb_mGetControlReg( BaseAddress )
```

Get the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the control register.

Note:

None.

```
#define XPlb2Opb_mGetErrorAddressReg( BaseAddress )
```

Get the error address (or BEAR), which is the address that just caused the error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit error address.

Note:

None.

```
#define XPlb2Opb_mGetErrorDetectReg( BaseAddress )
```

Get the error detect register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the error detect register.

Note:

None.

```
#define XPlb2Opb_mGetErrorTypeReg( BaseAddress )
```

Get the value of the error type register, which indicates whether the error is a timeout or a bus error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the BESR Lock error register.

Note:

None.

#define XPlb2Opb_mGetLockBitReg(BaseAddress)

Get the value of the lock bit register, which indicates whether the master has locked the error registers.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the BESR Lock error register.

Note:

None.

#define XPlb2Opb_mGetMasterDrivingReg(BaseAddress)

Get the master driving the error, if any.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the BESR Master driving error register.

Note:

None.

#define XPlb2Opb_mGetReadWriteReg(BaseAddress)

Get the value of the Read-Not-Write register, which indicates whether the error is a read error or write error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the BESR RNW error register.

Note:

None.

```
#define XPlb2Opb_mSetControlReg( BaseAddress,  
                                Mask          )
```

Set the control register to the given value.

Parameters:

BaseAddress is the base address of the device

Mask is the value to write to the control register.

Returns:

None.

Note:

None.

```
#define XPlb2Opb_mSetErrorDetectReg( BaseAddress,  
                                      Mask          )
```

Set the error detect register.

Parameters:

BaseAddress is the base address of the device

Mask is the 32-bit value to write to the error detect register.

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

plb2opb/v1_00_a/src/xplb2opb_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xplb2opb_l.h,v 1.2 2002/07/26 20:19:19 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file plb2opb/v1_00_a/src/xplb2opb_l.h
00026 *
00027 * This file contains identifiers and low-level macros that can be used to
00028 * access the device directly.  See xplb2opb.h for the high-level driver.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver   Who   Date       Changes
00034 * ----  ---  -
00035 * 1.00a rpm  05/10/02 First release
00036 * </pre>
00037 *
00038 *****/
00039
00040 #ifndef XPLB2OPB_L_H /* prevent circular inclusions */
00041 #define XPLB2OPB_L_H /* by using protection macros */
00042
```



```

00043 /***** Include Files *****/
00044 #include "xbasic_types.h"
00045 #include "xio.h"
00046 #include "xio_dcr.h"      /* DCR is only interface */
00047
00048 /***** Constant Definitions *****/
00049
00050 #define XP20_M0_ERROR_MASK      0x80000000UL
00051
00052 /* PLB-OPB Bridge Register offsets - DCR bus */
00053 #define XP20_BESR_MERR_OFFSET   0x00 /* error register */
00054
00055 #define XP20_BESR_MDRIVE_OFFSET 0x01 /* master driving the error */
00056 #define XP20_BESR_READ_OFFSET  0x02 /* error was a read operation */
00057 #define XP20_BESR_ERR_TYPE_OFFSET 0x03 /* error was a timeout */
00058 #define XP20_BESR_LCK_ERR_OFFSET 0x04 /* master has locked the registers */
00059 #define XP20_BEAR_ADDR_OFFSET   0x05 /* address where error occurred */
00060 #define XP20_BEAR_BYTE_EN_OFFSET 0x06 /* byte lane(s) where error occurred
*/
00061 #define XP20_BCR_OFFSET        0x07 /* control and status register */
00062
00063 /* BCR Register masks */
00064 #define XP20_BCR_ENABLE_INTR_MASK 0x80000000 /* set to enable interrupts */
00065 #define XP20_BCR_SOFTWARE_RESET_MASK 0x40000000 /* set to force reset,
*/
00066 /* clear otherwise */
00067
00068 /***** Type Definitions *****/
00069
00070 /***** Macros (Inline Functions) Definitions *****/
00071
00072 /* Define the appropriate I/O access method for the bridge currently only
00073  * DCR
00074  */
00075 #define XPlb2Opb_In32      XIo_DcrIn
00076 #define XPlb2Opb_Out32    XIo_DcrOut
00077
00078 /*****
00079  *
00080  * Low-level driver macros and functions. The list below provides signatures
00081  * to help the user use the macros.
00082  *
00083  * Xuint32 XPlb2Opb_mGetErrorDetectReg(Xuint32 BaseAddress)
00084  * void XPlb2Opb_mSetErrorDetectReg(Xuint32 BaseAddress, Xuint32 Mask)
00085  *
00086  * Xuint32 XPlb2Opb_mGetMasterDrivingReg(Xuint32 BaseAddress)
00087  * Xuint32 XPlb2Opb_mGetReadWriteReg(Xuint32 BaseAddress)
00088  * Xuint32 XPlb2Opb_mGetErrorTypeReg(Xuint32 BaseAddress)
00089  * Xuint32 XPlb2Opb_mGetLockBitReg(Xuint32 BaseAddress)
00090  * Xuint32 XPlb2Opb_mGetErrorAddressReg(Xuint32 BaseAddress)
00091  * Xuint32 XPlb2Opb_mGetByteEnableReg(Xuint32 BaseAddress)
00092  *
00093  * void XPlb2Opb_mSetControlReg(Xuint32 BaseAddress, Xuint32 Mask)

```

```

00094 * Xuint32 XPlb2Opb_mGetControlReg(Xuint32 BaseAddress)
00095 *
00096 *****/
00097
00098 /*****/
00099 /**
00100 *
00101 * Get the error detect register.
00102 *
00103 * @param BaseAddress is the base address of the device
00104 *
00105 * @return The 32-bit value of the error detect register.
00106 *
00107 * @note None.
00108 *
00109 *****/
00110 #define XPlb2Opb_mGetErrorDetectReg(BaseAddress) \
00111         XPlb2Opb_In32((BaseAddress) + XP20_BESR_MERR_OFFSET)
00112
00113
00114 /*****/
00115 /**
00116 *
00117 * Set the error detect register.
00118 *
00119 * @param BaseAddress is the base address of the device
00120 * @param Mask is the 32-bit value to write to the error detect register.
00121 *
00122 * @note None.
00123 *
00124 *****/
00125 #define XPlb2Opb_mSetErrorDetectReg(BaseAddress, Mask) \
00126         XPlb2Opb_Out32((BaseAddress) + XP20_BESR_MERR_OFFSET,
00127 (Mask))
00128
00129 /*****/
00130 /**
00131 *
00132 * Get the master driving the error, if any.
00133 *
00134 * @param BaseAddress is the base address of the device
00135 *
00136 * @return The 32-bit value of the BESR Master driving error register.
00137 *
00138 * @note None.
00139 *
00140 *****/
00141 #define XPlb2Opb_mGetMasterDrivingReg(BaseAddress) \
00142         XPlb2Opb_In32((BaseAddress) + XP20_BESR_MDRIVE_OFFSET)
00143
00144

```

```
00145 /*****  
00146 /**  
00147 *  
00148 * Get the value of the Read-Not-Write register, which indicates whether the  
00149 * error is a read error or write error.  
00150 *  
00151 * @param   BaseAddress is the base address of the device  
00152 *  
00153 * @return  The 32-bit value of the BESR RNW error register.  
00154 *  
00155 * @note    None.  
00156 *  
00157 *****/  
00158 #define XPlb2Opb_mGetReadWriteReg(BaseAddress) \  
00159         XPlb2Opb_In32((BaseAddress) + XP20_BESR_READ_OFFSET)  
00160  
00161  
00162 /*****  
00163 /**  
00164 *  
00165 * Get the value of the error type register, which indicates whether the error  
00166 * is a timeout or a bus error.  
00167 *  
00168 * @param   BaseAddress is the base address of the device  
00169 *  
00170 * @return  The 32-bit value of the BESR Lock error register.  
00171 *  
00172 * @note    None.  
00173 *  
00174 *****/  
00175 #define XPlb2Opb_mGetErrorTypeReg(BaseAddress) \  
00176         XPlb2Opb_In32((BaseAddress) + XP20_BESR_ERR_TYPE_OFFSET)  
00177  
00178  
00179 /*****  
00180 /**  
00181 *  
00182 * Get the value of the lock bit register, which indicates whether the master  
00183 * has locked the error registers.  
00184 *  
00185 * @param   BaseAddress is the base address of the device  
00186 *  
00187 * @return  The 32-bit value of the BESR Lock error register.  
00188 *  
00189 * @note    None.  
00190 *  
00191 *****/  
00192 #define XPlb2Opb_mGetLockBitReg(BaseAddress) \  
00193         XPlb2Opb_In32((BaseAddress) + XP20_BESR_LCK_ERR_OFFSET)  
00194  
00195  
00196 /*****
```

```

00197 /**
00198 *
00199 * Get the error address (or BEAR), which is the address that just caused the
00200 * error.
00201 *
00202 * @param   BaseAddress is the base address of the device
00203 *
00204 * @return  The 32-bit error address.
00205 *
00206 * @note    None.
00207 *
00208 *****/
00209 #define XPlb2Opb_mGetErrorAddressReg(BaseAddress) \
00210         XPlb2Opb_In32((BaseAddress) + XP20_BEAR_ADDR_OFFSET)
00211
00212
00213 *****/
00214 /**
00215 *
00216 * Get the error address byte enable register.
00217 *
00218 * @param   BaseAddress is the base address of the device
00219 *
00220 * @return  The 32-bit error address byte enable register contents.
00221 *
00222 * @note    None.
00223 *
00224 *****/
00225 #define XPlb2Opb_mGetByteEnableReg(BaseAddress) \
00226         XPlb2Opb_In32((BaseAddress) + XP20_BEAR_BYTE_EN_OFFSET)
00227
00228
00229 *****/
00230 /**
00231 *
00232 * Set the control register to the given value.
00233 *
00234 * @param   BaseAddress is the base address of the device
00235 * @param   Mask is the value to write to the control register.
00236 *
00237 * @return  None.
00238 *
00239 * @note    None.
00240 *
00241 *****/
00242 #define XPlb2Opb_mSetControlReg(BaseAddress, Mask) \
00243         XPlb2Opb_Out32((BaseAddress) + XP20_BCR_OFFSET, (Mask))
00244
00245
00246 *****/
00247 /**
00248 *

```

```
00249 * Get the contents of the control register.
00250 *
00251 * @param    BaseAddress is the base address of the device
00252 *
00253 * @return   The 32-bit value of the control register.
00254 *
00255 * @note     None.
00256 *
00257 *****/
00258 #define XPlb2Opb_mGetControlReg(BaseAddress) \
00259     XPlb2Opb_In32((BaseAddress) + XP20_BCR_OFFSET)
00260
00261
00262 /***** Function Prototypes *****/
00263
00264 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

XPlb2Opb_Config Struct Reference

```
#include <xplb2opb.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xuint8 NumMasters

Field Documentation

Xuint32 XPlb2Opb_Config::BaseAddress

Base address of device

Xuint16 XPlb2Opb_Config::DeviceId

Unique ID of device

Xuint8 XPlb2Opb_Config::NumMasters

Number of masters on the bus

The documentation for this struct was generated from the following file:

- [plb2opb/v1_00_a/src/xplb2opb.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

plb2opb/v1_00_a/src/xplb2opb.c File Reference

Detailed Description

Contains required functions for the [XPlb2Opb](#) component. See [xplb2opb.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	12/7/01	First release
1.00a	rpm	05/14/02	Made configuration typedef/lookup public

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xplb2opb.h"  
#include "xplb2opb_i.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Functions

[XStatus](#) [XPlb2Opb_Initialize](#) ([XPlb2Opb](#) *InstancePtr, [Xuint16](#) DeviceId)
[Xboolean](#) [XPlb2Opb_IsError](#) ([XPlb2Opb](#) *InstancePtr)
void [XPlb2Opb_ClearErrors](#) ([XPlb2Opb](#) *InstancePtr, [Xuint8](#) Master)

Xuint32 XPlb2Opb_GetErrorStatus (XPlb2Opb *InstancePtr, **Xuint8** Master)

Xuint32 XPlb2Opb_GetErrorAddress (XPlb2Opb *InstancePtr)

Xuint32 XPlb2Opb_GetErrorByteEnables (XPlb2Opb *InstancePtr)

Xuint8 XPlb2Opb_GetMasterDrivingError (XPlb2Opb *InstancePtr)

Xuint8 XPlb2Opb_GetNumMasters (XPlb2Opb *InstancePtr)

void XPlb2Opb_EnableInterrupt (XPlb2Opb *InstancePtr)

void XPlb2Opb_DisableInterrupt (XPlb2Opb *InstancePtr)

void XPlb2Opb_Reset (XPlb2Opb *InstancePtr)

XPlb2Opb_Config * XPlb2Opb_LookupConfig (**Xuint16** DeviceId)

Function Documentation

```
void XPlb2Opb_ClearErrors( XPlb2Opb * InstancePtr,  
                          Xuint8     Master  
                          )
```

Clears any outstanding errors for the given master.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

None.

Note:

None.

```
void XPlb2Opb_DisableInterrupt( XPlb2Opb * InstancePtr)
```

Disables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void XPlb2Opb_EnableInterrupt( XPlb2Opb * InstancePtr)
```

Enables the interrupt output from the bridge

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

The bridge hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
Xuint32 XPlb2Opb_GetErrorAddress( XPlb2Opb * InstancePtr)
```

Returns the OPB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XPlb2Opb_IsError()** is recommended to confirm that an error has occurred prior to calling **XPlb2Opb_GetErrorAddress()** to ensure that the data in the error address register is relevant.

Xuint32 XPlb2Opb_GetErrorByteEnables(XPlb2Opb * InstancePtr)

Returns the byte-enables asserted during the access causing the error. The enables are parameters in the hardware making the return value dynamic. An example of a 32-bit bus with all 4 byte enables available, XPlb2Opb_GetErrorByteEnables will have the value 0xF0000000 returned from a 32-bit access error.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The byte-enables asserted during the error causing access.

Note:

None.

**Xuint32 XPlb2Opb_GetErrorStatus(XPlb2Opb * InstancePtr,
Xuint8 Master
)**

Returns the error status for the specified master.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

The current error status for the requested master on the PLB. The status is a bit-mask and the values are described in **xplb2opb.h**.

Note:

None.

Xuint8 XPlb2Opb_GetMasterDrivingError(**XPlb2Opb** * *InstancePtr*)

Returns the ID of the master which is driving the error condition

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The ID of the master that is driving the error

Note:

None.

Xuint8 XPlb2Opb_GetNumMasters(**XPlb2Opb** * *InstancePtr*)

Returns the number of masters associated with the provided instance

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

The number of masters. This is a number from 1 to the maximum of 32.

Note:

The value returned from this call needs to be adjusted if it is to be used as the argument for other calls since the masters are numbered from 0 and this function returns values starting at 1.

```
XStatus XPlb2Opb_Initialize( XPlb2Opb * InstancePtr,  
                             Xuint16      DeviceId  
                             )
```

Initializes a specific **XPlb2Opb** instance. Looks for configuration data for the specified device, then initializes instance data.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPlb2Opb** component. Passing in a device id associates the generic **XPlb2Opb** component to a specific device, as chosen by the caller or application developer.

Returns:

- o XST_SUCCESS if everything starts up as expected.
- o XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
Xboolean XPlb2Opb_IsError( XPlb2Opb * InstancePtr)
```

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

XPlb2Opb_Config* XPlb2Opb_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table PlbOpbConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for

Returns:

A pointer to the configuration data for the given device, or XNULL if no match is found.

Note:

None.

void XPlb2Opb_Reset(XPlb2Opb * InstancePtr)

Forces a software-induced reset to occur in the bridge. Disables interrupts in the process.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

Returns:

None.

Note:

Disables interrupts in the process.

plb2opb/v1_00_a/src/xplb2opb_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xplb2opb_i.h,v 1.1 2002/06/26 15:36:52 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file plb2opb/v1_00_a/src/xplb2opb_i.h
00026 *
00027 * This file contains data which is shared between files and internal to the
00028 * XPlb2Opb component. It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date      Changes
00034 * -----
00035 * 1.00a  ecm    02/28/02  First release
00036 * 1.00a  rpm    05/14/02  Moved identifiers to xplb2opb_1.h
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XPLB2OPB_I_H /* prevent circular inclusions */
00042 #define XPLB2OPB_I_H /* by using protection macros */
```



```
00043
00044 /***** Include Files *****/
00045 #include "xplb2opb_1.h"
00046
00047 /***** Constant Definitions *****/
00048
00049 /***** Type Definitions *****/
00050
00051 /***** Macros (Inline Functions) Definitions *****/
00052
00053 /***** Function Prototypes *****/
00054
00055 /***** Variables *****/
00056
00057 extern XPlb2Opb_Config XPlb2Opb_ConfigTable[];
00058
00059 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

plb2opb/v1_00_a/src/xplb2opb_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XPlb2Opb** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	ecm	02/28/02	First release
1.00a	rpm	05/14/02	Moved identifiers to xplb2opb_1.h

```
#include "xplb2opb\_1.h"
```

[Go to the source code of this file.](#)

Variables

[XPlb2Opb_Config](#) [XPlb2Opb_ConfigTable](#) []

Variable Documentation

XPlb2Opb_Config XPlb2Opb_ConfigTable[]()

The PLB-to-OPB bridge configuration table, sized by the number of instances defined in **xparameters.h**.

Generated on 29 May 2003 for Xilinx Device Drivers

plb2opb/v1_00_a/src/xplb2opb_selftest.c

File Reference

Detailed Description

Contains diagnostic self-test functions for the [XPlb2Opb](#) component. See [xplb2opb.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	ecm	12/7/01	First release

```
#include "xstatus.h"  
#include "xplb2opb.h"  
#include "xplb2opb_i.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Functions

[XStatus XPlb2Opb_SelfTest](#) ([XPlb2Opb](#) *InstancePtr, [Xuint32](#) TestAddress)

Function Documentation

```
XStatus XPlb2Opb_SelfTest( XPlb2Opb * InstancePtr,  
                           Xuint32   TestAddress  
                           )
```

Runs a self-test on the driver/device.

This tests reads the BCR to verify that the proper value is there.

XST_SUCCESS is returned if expected value is there, XST_PLB2OPB_FAIL_SELFTEST is returned otherwise.

Parameters:

InstancePtr is a pointer to the **XPlb2Opb** instance to be worked on.

TestAddress is a location that could cause an error on read, not used - user definable for hw specific implementations.

Returns:

XST_SUCCESS if successful, or XST_PLB2OPB_FAIL_SELFTEST if the driver fails self-test.

Note:

This test assumes that the bus error interrupts are not enabled.

plb2opb/v1_00_a/src/xplb2opb_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of PLB-to-OPB bridge devices in the system. Each bridge device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	11/16/01	First release
1.00a	rpm	05/14/02	Made configuration typedef/lookup public

```
#include "xplb2opb.h"  
#include "xparameters.h"
```

Variables

[XPlb2Opb_Config](#) [XPlb2Opb_ConfigTable](#) [XPAR_XPLB2OPB_NUM_INSTANCES]

Variable Documentation

[XPlb2Opb_Config](#) [XPlb2Opb_ConfigTable](#)[XPAR_XPLB2OPB_NUM_INSTANCES]

The PLB-to-OPB bridge configuration table, sized by the number of instances defined in [xparameters.h](#).

Generated on 29 May 2003 for Xilinx Device Drivers

pci/v1_00_a/src/xpci_intr.c File Reference

Detailed Description

Implements PCI interrupt processing functions for the [XPci](#) component. See [xpci.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	rmm	03/25/02	Original code

```
#include "xpci.h"
```

Functions

```
void XPci\_InterruptGlobalEnable (XPci *InstancePtr)  
void XPci\_InterruptGlobalDisable (XPci *InstancePtr)  
void XPci\_InterruptEnable (XPci *InstancePtr, Xuint32 Mask)  
void XPci\_InterruptDisable (XPci *InstancePtr, Xuint32 Mask)  
void XPci\_InterruptClear (XPci *InstancePtr, Xuint32 Mask)
```

```
Xuint32 XPci\_InterruptGetEnabled (XPci *InstancePtr)
```

```
Xuint32 XPci\_InterruptGetStatus (XPci *InstancePtr)
```

```
Xuint32 XPci\_InterruptGetPending (XPci *InstancePtr)
```

```
Xuint32 XPci\_InterruptGetHighestPending (XPci *InstancePtr)
```



```
void XPCI_InterruptPciEnable (XPCI *InstancePtr, XUINT32 Mask)
void XPCI_InterruptPciDisable (XPCI *InstancePtr, XUINT32 Mask)
void XPCI_InterruptPciClear (XPCI *InstancePtr, XUINT32 Mask)
XUINT32 XPCI_InterruptPciGetEnabled (XPCI *InstancePtr)
XUINT32 XPCI_InterruptPciGetStatus (XPCI *InstancePtr)
void XPCI_AckSend (XPCI *InstancePtr, XUINT32 Vector)
XUINT32 XPCI_AckRead (XPCI *InstancePtr)
void XPCI_SpecialCycle (XPCI *InstancePtr, XUINT32 Data)
```

Function Documentation

```
XUINT32 XPCI_AckRead( XPCI * InstancePtr)
```

Read the contents of the PCI interrupt acknowledge vector register.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

System dependent interrupt vector.

Note:

None

```
void XPCI_AckSend( XPCI * InstancePtr,
                  XUINT32 Vector
                  )
```

Generate a PCI interrupt acknowledge bus cycle with the given vector.

Parameters:

InstancePtr is the PCI component to operate on.

Vector is a system dependent interrupt vector to place on the bus.

Note:

None

```
void XPci_InterruptClear( XPci * InstancePtr,  
                          Xuint32 Mask  
                          )
```

Clear device level pending interrupts with the provided mask.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK

Note:

None

```
void XPci_InterruptDisable( XPci * InstancePtr,  
                            Xuint32 Mask  
                            )
```

Disable device interrupts. Any component interrupts enabled through [XPci_InterruptPciEnable\(\)](#) and/or the DMA driver will no longer have any effect. The component interrupt settings will be retained however.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK

Note:

None

```
void XPci_InterruptEnable( XPci * InstancePtr,  
                           Xuint32 Mask  
                           )
```

Enable device interrupts. Device interrupts must be enabled by this function before component interrupts enabled by **XPCI_InterruptPciEnable()** and/or the DMA driver have any effect.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK.

Note:

None

Xuint32 XPCI_InterruptGetEnabled(XPCI * InstancePtr)

Returns the device level interrupt enable mask as set by **XPCI_InterruptEnable()**.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK.

Note:

None

Xuint32 XPCI_InterruptGetHighestPending(XPCI * InstancePtr)

Returns the highest priority pending device interrupt that has been enabled by **XPCI_InterruptEnable()**.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask is one set bit made from XPCI_IPIF_INT_MASK or zero if no interrupts are pending.

Note:

None

Xuint32 XPCI_InterruptGetPending(XPCI * InstancePtr)

Returns the pending status of device level interrupt signals that have been enabled by [XPCI_InterruptEnable\(\)](#). Any bit in the mask set to 1 indicates that an interrupt is pending from the given component

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK or zero if no interrupts are pending.

Note:

None

Xuint32 XPCI_InterruptGetStatus(XPCI * InstancePtr)

Returns the status of device level interrupt signals. Any bit in the mask set to 1 indicates that the given component has asserted an interrupt condition.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK.

Note:

The interrupt status indicates the status of the device regardless if the interrupts from the devices have been enabled or not through [XPCI_InterruptEnable\(\)](#).

void XPCI_InterruptGlobalDisable(XPCI * InstancePtr)

Disable the core's interrupt output signal. Interrupts enabled through **XPci_InterruptEnable()** and **XPci_InterruptPciEnable()** will no longer be passed through until the IPIF global enable bit is set by **XPci_InterruptGlobalEnable()**.

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

```
void XPci_InterruptGlobalEnable( XPci * InstancePtr)
```

Enable the core's interrupt output signal. Interrupts enabled through **XPci_InterruptEnable()** and **XPci_InterruptPciEnable()** will not be passed through until the IPIF global enable bit is set by this function.

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

```
void XPci_InterruptPciClear( XPci * InstancePtr,  
                             Xuint32 Mask  
                             )
```

Clear PCI bridge specific interrupt status bits with the provided mask.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XPCI_IR_MASK

Note:

None

```
void XPCI_InterruptPciDisable( XPCI * InstancePtr,  
                               Xuint32 Mask  
                               )
```

Disable PCI bridge specific interrupts.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XPCI_IR_MASK

Note:

None

```
void XPCI_InterruptPciEnable( XPCI * InstancePtr,  
                              Xuint32 Mask  
                              )
```

Enable PCI bridge specific interrupts. Before this function has any effect in generating interrupts, the function **XPCI_InterruptEnable()** must be invoked with the XPCI_IPIF_INT_PCI bit set.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from XPCI_IR_MASK.

Note:

None

```
Xuint32 XPCI_InterruptPciGetEnabled( XPCI * InstancePtr)
```

Get the PCI bridge specific interrupts enabled through [XPci_InterruptPciEnable\(\)](#). Bits set to 1 mean that interrupt source is enabled.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of enabled bits made from XPCI_IR_MASK.

Note:

None

Xuint32 XPci_InterruptPciGetStatus(XPci * InstancePtr)

Get the status of PCI bridge specific interrupts that have been asserted Bits set to 1 are in an asserted state. Bits may be set to 1 irregardless of whether they have been enabled or not though [XPci_InterruptPciEnable\(\)](#). To get the pending interrupts, AND the results of this function with [XPci_InterruptPciGetEnabled\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of enabled bits made from XPCI_IR_MASK.

Note:

None

**void XPci_SpecialCycle(XPci * InstancePtr,
Xuint32 Data
)**

Broadcasts a message to all listening PCI targets.

Parameters:

InstancePtr is the PCI component to operate on.

Data is the data to broadcast.

Note:

None

Generated on 29 May 2003 for Xilinx Device Drivers

XPCI Struct Reference

```
#include <xpci.h>
```

Detailed Description

The XPCI driver instance data. The user is required to allocate a variable of this type for every PCI device in the system that will be using this API. A pointer to a variable of this type is passed to the driver API functions defined here.

Data Fields

Xuint32 RegBaseAddr

Xuint32 DmaRegBaseAddr

Xuint32 IsReady

Xuint8 DmaType

Field Documentation

Xuint32 XPCI::DmaRegBaseAddr

Base address of DMA (if included)

Xuint8 XPCI::DmaType

Type of DMA (if enabled), see XPCI_DMA_TYPE constants in [xpci_l.h](#)

Xuint32 XPci::IsReady

Device is initialized and ready

Xuint32 XPci::RegBaseAddr

Base address of registers

The documentation for this struct was generated from the following file:

- [pci/v1_00_a/src/xpci.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

pci/v1_00_a/src/xpci.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xpci.h,v 1.3 2003/05/07 22:15:57 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file pci/v1_00_a/src/xpci.h
00026 *
00027 * This file contains the software API definition of the Xilinx PCI bridge
00028 * (XPCI) component. This component bridges between local bus IPIF and the
00029 * Xilinx LogiCORE PCI64 Interface v3.0 core. It provides full bridge
00030 * functionality between the local bus a 32 bit V2.2 compliant PCI
00031 * bus.
00032 *
00033 * <b>Features</b>
00034 *
00035 * This driver allows the user to access the device's registers to perform PCI
00036 * configuration read and write access, error detection and processing, and
00037 * interrupt management.
00038 *
00039 * The Xilinx PCI bridge controller is a soft IP core designed for
00040 * Xilinx FPGAs and contains the following features:
00041 * - Supports 32 bit OPB local bus
00042 * - PCI V2.2 Compliant
```

00043 * - Robust error reporting and diagnostics
00044 * - DMA capable
00045 *
00046 * **Interrupt Management**
00047 *
00048 * The XPCI component driver provides interrupt management functions.
00049 * Implementation of callback handlers is left to the user. Refer to the
provided
00050 * PCI code fragments in the examples directory.
00051 *
00052 * The PCI bridge IP core uses the IPIF to manage interrupts from devices within
00053 * it. Devices in this core include the PCI bridge itself and an optional DMA
00054 * engine. To manage interrupts from these devices, a three layer approach is
00055 * utilized and is modeled on the IPIF.
00056 *
00057 * Device specific interrupt control is at the lowest layer. This is where
00058 * individual sources are managed. For example, PCI Master Abort or DMA complete
00059 * interrupts are enabled/disabled/cleared here. The XPCI function API that
00060 * manages this layer is identified as XPCI_InterruptPci<operation>(). DMA
00061 * interrupts at this layer are managed by the XDma_Channel software component.
00062 *
00063 * The middle layer is utilized to manage interrupts at a device level. For
00064 * example, enabling PCI interrupts at this layer allows any PCI device specific
00065 * interrupt enabled at the lowest layer to be passed up to the highest layer.
00066 * The XPCI function API that manages this layer is identified as
00067 * XPCI_Interrupt<operation>().
00068 *
00069 * The middle layer serves little purpose when there is no DMA engine and can
00070 * largely be ignored. During initialization, use XPCI_InterruptEnable(...,
00071 * XPCI_IPIF_INT_PCI) to allow all PCI interrupts enabled at the lowest layer
00072 * to pass through. After this operation, the middle layer can be forgotten.
00073 *
00074 * The highest layer is simply a global interrupt enable/disable switch that
00075 * allows all or none of the enabled interrupts to be passed on to an interrupt
00076 * controller. The XPCI function API that manages this level is identified as
00077 * XPCI_InterruptGlobal<operation>().
00078 *
00079 * **DMA**
00080 *
00081 * The PCI bridge can include a DMA engine in HW. The XPCI software driver can
00082 * be used to query which type of DMA engine has been implemented and manage
00083 * interrupts. The application is required to initialize an XDma_Channel
00084 * component driver and provide an interrupt service routine to service DMA
00085 * exceptions. Example DMA management code is provided in the examples
directory.
00086 *
00087 * @note
00088 * This driver is intended to be used to bridge across multiple types of
00089 * buses (PLB or OPB). While the register set will remain the same for all
buses,
00090 * their bit definitions may change slightly from bus to bus. The differences
00091 * that arise out of this are clearly documented in this file.
00092 *

```

00093 * <pre>
00094 * MODIFICATION HISTORY:
00095 *
00096 * Ver    Who    Date      Changes
00097 * -----
00098 * 1.00a rmm  04/15/03 First release
00099 * </pre>
00100 *
00101 *****/
00102 #ifndef XPCI_H /* prevent circular inclusions */
00103 #define XPCI_H /* by using protection macros */
00104
00105 /***** Include Files *****/
00106 #include "xbasic_types.h"
00107 #include "xstatus.h"
00108 #include "xpci_l.h"
00109
00110 /***** Constant Definitions *****/
00111
00112
00113 /*
00114  * XPCI_CLEAR_ALL_ERRORS is for use with XPCI_ErrorClear()
00115  */
00116 #define XPCI_CLEAR_ALL_ERRORS ((XPCI_Error*)3)
00117
00118
00119 /***** Type Definitions *****/
00120
00121 /*
00122  * This typedef contains configuration information for the device.
00123  */
00124 typedef struct
00125 {
00126     Xuint16 DeviceId; /*< Unique ID of device */
00127     Xuint32 RegBaseAddr; /*< Register base address */
00128     Xuint32 DmaBaseAddr; /*< DMA register base address */
00129     Xuint32 CfgBar0; /*< PCI BAR 0 */
00130     Xuint32 CfgBar1; /*< PCI BAR 1 */
00131     Xuint32 CfgBar2; /*< PCI BAR 2 */
00132     Xuint8 CfgPrefetch0; /*< prefetchable setting for PCI BAR 0 */
00133     Xuint8 CfgPrefetch1; /*< prefetchable setting for PCI BAR 1 */
00134     Xuint8 CfgPrefetch2; /*< prefetchable setting for PCI BAR 2 */
00135     Xuint8 CfgSpace0; /*< IO or memory space for PCI BAR 0 */
00136     Xuint8 CfgSpace1; /*< IO or memory space for PCI BAR 1 */
00137     Xuint8 CfgSpace2; /*< IO or memory space for PCI BAR 2 */
00138     Xuint8 DmaType; /*< DMA type */
00139 } XPCI_Config;
00140
00141 /**
00142  * The XPCI driver instance data. The user is required to allocate a

```

```

00143 * variable of this type for every PCI device in the system that will be
00144 * using this API. A pointer to a variable of this type is passed to the driver
00145 * API functions defined here.
00146 */
00147 typedef struct
00148 {
00149     Xuint32 RegBaseAddr;    /**< Base address of registers */
00150     Xuint32 DmaRegBaseAddr;/**< Base address of DMA (if included) */
00151     Xuint32 IsReady;       /**< Device is initialized and ready */
00152     Xuint8  DmaType;       /**< Type of DMA (if enabled), see XPCI_DMA_TYPE
00153                             constants in xpci_l.h */
00154 } XPCI;
00155
00156 /**
00157 * XPCIError is used to retrieve a snapshot of the bridge's error state.
00158 * Most of the attributes of this structure are copies of various bridge
00159 * registers. See XPCI_ErrorGet() and XPCI_ErrorClear().
00160 */
00161 typedef struct
00162 {
00163     Xboolean IsError;      /**< Global error indicator */
00164     Xuint32 LocalBusReason;/**< Local bus master address definition */
00165     Xuint32 PciReason;    /**< PCI address definition */
00166     Xuint32 PciSerrReason;/**< PCI System error definiton */
00167     Xuint32 LocalBusReadAddr;/**< Local bus master read error address */
00168     Xuint32 LocalBusWriteAddr;/**< Local bus master write error address */
00169     Xuint32 PciReadAddr;  /**< PCI read error address */
00170     Xuint32 PciWriteAddr; /**< PCI write error address */
00171     Xuint32 PciSerrReadAddr;/**< PCI initiator read SERR address */
00172     Xuint32 PciSerrWriteAddr;/**< PCI initiator write SERR address */
00173 } XPCIError;
00174
00175 /***** Macros (Inline Functions) Definitions *****/
00176
00177
00178 /***** Function Prototypes *****/
00179
00180 /*
00181 * Initialization & raw PCI configuration functions.
00182 * This API is implemented in xpci.c
00183 */
00184 XStatus XPCI_Initialize(XPCI *InstancePtr, Xuint16 DeviceId, int BusNo,
00185                        int SubBusNo);
00186 void XPCI_Reset(XPCI *InstancePtr);
00187 Xuint32 XPCI_ConfigPack(unsigned Bus, unsigned Device, unsigned Function);
00188 Xuint32 XPCI_ConfigIn(XPCI *InstancePtr, Xuint32 ConfigAddress, Xuint8 Offset);
00189 void XPCI_ConfigOut(XPCI *InstancePtr, Xuint32 ConfigAddress, Xuint8 Offset,
00190                    Xuint32 ConfigData);
00191 void XPCI_ErrorGet(XPCI *InstancePtr, XPCIError *ErrorDataPtr);

```

```

00192 void      XPci_ErrorClear(XPci *InstancePtr, XPciError *ErrorDataPtr);
00193 void      XPci_InhibitAfterError(XPci *InstancePtr, Xuint32 Mask);
00194 void      XPci_SetBusNumber(XPci *InstancePtr, int BusNo, int SubBusNo);
00195 void      XPci_GetBusNumber(XPci *InstancePtr, int *BusNoPtr, int *SubBusNoPtr);
00196 void      XPci_GetDmaImplementation(XPci *InstancePtr, Xuint32 *BaseAddr,
00197                                     Xuint8 *DmaType);
00198 XPci_Config *XPci_LookupConfig(Xuint16 DeviceId);
00199
00200 /*
00201  * PCI bus configuration functions.
00202  * This API is implemented in xpci_config.c
00203  */
00204 XStatus XPci_ConfigIn8(XPci *InstancePtr, unsigned Bus, unsigned Device,
00205                       unsigned Func, unsigned Offset, Xuint8 *Data);
00206 XStatus XPci_ConfigIn16(XPci *InstancePtr, unsigned Bus, unsigned Device,
00207                        unsigned Func, unsigned Offset, Xuint16 *Data);
00208 XStatus XPci_ConfigIn32(XPci *InstancePtr, unsigned Bus, unsigned Device,
00209                       unsigned Func, unsigned Offset, Xuint32 *Data);
00210
00211 XStatus XPci_ConfigOut8(XPci *InstancePtr, unsigned Bus, unsigned Device,
00212                      unsigned Func, unsigned Offset, Xuint8 Data);
00213 XStatus XPci_ConfigOut16(XPci *InstancePtr, unsigned Bus, unsigned Device,
00214                       unsigned Func, unsigned Offset, Xuint16 Data);
00215 XStatus XPci_ConfigOut32(XPci *InstancePtr, unsigned Bus, unsigned Device,
00216                       unsigned Func, unsigned Offset, Xuint32 Data);
00217
00218 /*
00219  * Interrupt processing and special cycle functions
00220  * This API is implemented in xpci_intr.c
00221  */
00222 void      XPci_InterruptGlobalEnable(XPci *InstancePtr);
00223 void      XPci_InterruptGlobalDisable(XPci *InstancePtr);
00224
00225 void      XPci_InterruptEnable(XPci *InstancePtr, Xuint32 Mask);
00226 void      XPci_InterruptDisable(XPci *InstancePtr, Xuint32 Mask);
00227 void      XPci_InterruptClear(XPci *InstancePtr, Xuint32 Mask);
00228 Xuint32 XPci_InterruptGetEnabled(XPci *InstancePtr);
00229 Xuint32 XPci_InterruptGetStatus(XPci *InstancePtr);
00230 Xuint32 XPci_InterruptGetPending(XPci *InstancePtr);
00231 Xuint32 XPci_InterruptGetHighestPending(XPci *InstancePtr);
00232
00233 void      XPci_InterruptPciEnable(XPci *InstancePtr, Xuint32 Mask);
00234 void      XPci_InterruptPciDisable(XPci *InstancePtr, Xuint32 Mask);
00235 void      XPci_InterruptPciClear(XPci *InstancePtr, Xuint32 Mask);
00236 Xuint32 XPci_InterruptPciGetEnabled(XPci *InstancePtr);
00237 Xuint32 XPci_InterruptPciGetStatus(XPci *InstancePtr);
00238
00239 void      XPci_AckSend(XPci *InstancePtr, Xuint32 Vector);

```

```
00240 Xuint32 XPCI_AckRead(XPCI *InstancePtr);
00241 void     XPCI_SpecialCycle(XPCI *InstancePtr, Xuint32 Data);
00242
00243 /*
00244  * V3 core access functions
00245  * This API implemented in xpci_v3.c
00246  */
00247 Xuint32 XPCI_V3StatusCommandGet(XPCI *InstancePtr);
00248 Xuint32 XPCI_V3TransactionStatusGet(XPCI *InstancePtr);
00249 void     XPCI_V3TransactionStatusClear(XPCI *InstancePtr, Xuint32 Data);
00250
00251 /*
00252  * Selftest
00253  * This API is implemented in xpci_selftest.c
00254  */
00255 XStatus XPCI_SelfTest(XPCI *InstancePtr);
00256
00257 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

pci/v1_00_a/src/xpci.h File Reference

Detailed Description

This file contains the software API definition of the Xilinx PCI bridge (**XPci**) component. This component bridges between local bus IPIF and the Xilinx LogiCORE PCI64 Interface v3.0 core. It provides full bridge functionality between the local bus a 32 bit V2.2 compliant PCI bus.

Features

This driver allows the user to access the device's registers to perform PCI configuration read and write access, error detection and processing, and interrupt management.

The Xilinx PCI bridge controller is a soft IP core designed for Xilinx FPGAs and contains the following features:

- Supports 32 bit OPB local bus
- PCI V2.2 Compliant
- Robust error reporting and diagnostics
- DMA capable

Interrupt Management

The **XPci** component driver provides interrupt management functions. Implementation of callback handlers is left to the user. Refer to the provided PCI code fragments in the examples directory.

The PCI bridge IP core uses the IPIF to manage interrupts from devices within it. Devices in this core include the PCI bridge itself and an optional DMA engine. To manage interrupts from these devices, a three layer approach is utilized and is modeled on the IPIF.

Device specific interrupt control is at the lowest layer. This is where individual sources are managed. For example, PCI Master Abort or DMA complete interrupts are enabled/disabled/cleared here. The **XPci** function API that manages this layer is identified as `XPci_InterruptPci<operation>()`. DMA interrupts at this layer are managed by the `XDma_Channel` software component.

The middle layer is utilized to manage interrupts at a device level. For example, enabling PCI interrupts at this layer allows any PCI device specific interrupt enabled at the lowest layer to be passed up to the highest layer. The XPCI function API that manages this layer is identified as XPCI_Interrupt<operation>().

The middle layer serves little purpose when there is no DMA engine and can largely be ignored. During initialization, use XPCI_InterruptEnable(..., XPCI_IPIF_INT_PCI) to allow all PCI interrupts enabled at the lowest layer to pass through. After this operation, the middle layer can be forgotten.

The highest layer is simply a global interrupt enable/disable switch that allows all or none of the enabled interrupts to be passed on to an interrupt controller. The **XPCI** function API that manages this level is identified as XPCI_InterruptGlobal<operation>().

DMA

The PCI bridge can include a DMA engine in HW. The **XPCI** software driver can be used to query which type of DMA engine has been implemented and manage interrupts. The application is required to initialize an XDma_Channel component driver and provide an interrupt service routine to service DMA exceptions. Example DMA management code is provided in the examples directory.

Note:

This driver is intended to be used to bridge across multiple types of buses (PLB or OPB). While the register set will remain the same for all buses, their bit definitions may change slightly from bus to bus. The differences that arise out of this are clearly documented in this file.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rmm	04/15/03	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xpci_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XPCI  
struct XPCI_Config
```

struct **XPCIError**

Functions

XStatus **XPci_Initialize** (**XPci** *InstancePtr, **Xuint16** DeviceId, int BusNo, int SubBusNo)

void **XPci_Reset** (**XPci** *InstancePtr)

Xuint32 **XPci_ConfigPack** (unsigned Bus, unsigned Device, unsigned Function)

Xuint32 **XPci_ConfigIn** (**XPci** *InstancePtr, **Xuint32** ConfigAddress, **Xuint8** Offset)

void **XPci_ConfigOut** (**XPci** *InstancePtr, **Xuint32** ConfigAddress, **Xuint8** Offset, **Xuint32** ConfigData)

void **XPci_ErrorGet** (**XPci** *InstancePtr, **XPCIError** *ErrorDataPtr)

void **XPci_ErrorClear** (**XPci** *InstancePtr, **XPCIError** *ErrorDataPtr)

void **XPci_InhibitAfterError** (**XPci** *InstancePtr, **Xuint32** Mask)

void **XPci_SetBusNumber** (**XPci** *InstancePtr, int BusNo, int SubBusNo)

void **XPci_GetBusNumber** (**XPci** *InstancePtr, int *BusNoPtr, int *SubBusNoPtr)

void **XPci_GetDmaImplementation** (**XPci** *InstancePtr, **Xuint32** *BaseAddr, **Xuint8** *DmaType)

XPci_Config * **XPci_LookupConfig** (**Xuint16** DeviceId)

XStatus **XPci_ConfigIn8** (**XPci** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint8** *Data)

XStatus **XPci_ConfigIn16** (**XPci** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint16** *Data)

XStatus **XPci_ConfigIn32** (**XPci** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint32** *Data)

XStatus **XPci_ConfigOut8** (**XPci** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint8** Data)

XStatus **XPci_ConfigOut16** (**XPci** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint16** Data)

XStatus **XPci_ConfigOut32** (**XPci** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint32** Data)

void **XPci_InterruptGlobalEnable** (**XPci** *InstancePtr)

void **XPci_InterruptGlobalDisable** (**XPci** *InstancePtr)

void **XPci_InterruptEnable** (**XPci** *InstancePtr, **Xuint32** Mask)

void **XPci_InterruptDisable** (**XPci** *InstancePtr, **Xuint32** Mask)

void **XPci_InterruptClear** (**XPci** *InstancePtr, **Xuint32** Mask)

Xuint32 **XPci_InterruptGetEnabled** (**XPci** *InstancePtr)

Xuint32 **XPci_InterruptGetStatus** (**XPci** *InstancePtr)

Xuint32 **XPci_InterruptGetPending** (**XPci** *InstancePtr)

Xuint32 **XPci_InterruptGetHighestPending** (**XPci** *InstancePtr)

void **XPci_InterruptPciEnable** (**XPci** *InstancePtr, **Xuint32** Mask)

void **XPci_InterruptPciDisable** (**XPci** *InstancePtr, **Xuint32** Mask)

```
void XPCI_InterruptPciClear (XPCI *InstancePtr, XUINT32 Mask)
XUINT32 XPCI_InterruptPciGetEnabled (XPCI *InstancePtr)
XUINT32 XPCI_InterruptPciGetStatus (XPCI *InstancePtr)
void XPCI_AckSend (XPCI *InstancePtr, XUINT32 Vector)
XUINT32 XPCI_AckRead (XPCI *InstancePtr)
void XPCI_SpecialCycle (XPCI *InstancePtr, XUINT32 Data)
XUINT32 XPCI_V3StatusCommandGet (XPCI *InstancePtr)
XUINT32 XPCI_V3TransactionStatusGet (XPCI *InstancePtr)
void XPCI_V3TransactionStatusClear (XPCI *InstancePtr, XUINT32 Data)
XSTATUS XPCI_SelfTest (XPCI *InstancePtr)
```

Function Documentation

```
XUINT32 XPCI_AckRead( XPCI * InstancePtr)
```

Read the contents of the PCI interrupt acknowledge vector register.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

System dependent interrupt vector.

Note:

None

```
void XPCI_AckSend( XPCI * InstancePtr,
                  XUINT32 Vector
                  )
```

Generate a PCI interrupt acknowledge bus cycle with the given vector.

Parameters:

InstancePtr is the PCI component to operate on.

Vector is a system dependent interrupt vector to place on the bus.

Note:

None

```

Xuint32 XPCI_ConfigIn( XPCI * InstancePtr,
                        Xuint32 ConfigAddress,
                        Xuint8 Offset
                      )

```

Perform a 32 bit configuration read transaction.

Parameters:

InstancePtr is the PCI component to operate on.

ConfigAddress contains the address of the PCI device to access. It should be properly formatted for writing to the PCI configuration access port. (see [XPCI_ConfigPack\(\)](#))

Offset is the register offset within the PCI device being accessed.

Returns:

32 bit data word from addressed device

Note:

This function performs the same type of operation that XPCI_ConfigIn32, does except the user must format the ConfigAddress

```

XStatus XPCI_ConfigIn16( XPCI * InstancePtr,
                          unsigned Bus,
                          unsigned Device,
                          unsigned Func,
                          unsigned Offset,
                          Xuint16 * Data
                        )

```

Perform a 16 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Data is the data read from the target.

Returns:

- o XST_SUCCESS Operation was successful.
- o XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus XPCI_ConfigIn32( XPCI * InstancePtr,  
                        unsigned Bus,  
                        unsigned Device,  
                        unsigned Func,  
                        unsigned Offset,  
                        Xuint32 * Data  
                        )
```

Perform a 32 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Data is the data read from the target.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus XPCI_ConfigIn8( XPCI * InstancePtr,  
                       unsigned Bus,  
                       unsigned Device,  
                       unsigned Func,  
                       unsigned Offset,  
                       Xuint8 * Data  
                       )
```

Perform a 8 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.
- Data* is the data read from the target.

Returns:

- XST_SUCCESS Operation was successful.
- XST_PCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
void XPCI_ConfigOut( XPCI * InstancePtr,  
                   XUINT32 ConfigAddress,  
                   XUINT8 Offset,  
                   XUINT32 ConfigData  
                   )
```

Perform a 32 bit configuration write transaction.

Parameters:

- InstancePtr* is the PCI component to operate on.
- ConfigAddress* contains the address of the PCI device to access. It should be properly formatted for writing to the PCI configuration access port. (see [XPCI_ConfigPack\(\)](#))
- Offset* is the register offset within the PCI device being accessed.
- ConfigData* is the data to write to the addressed device.

Note:

This function performs the same type of operation that XPCI_ConfigOutWord, does except the user must format the Car.

```
XStatus XPci_ConfigOut16( XPci * InstancePtr,  
                          unsigned Bus,  
                          unsigned Device,  
                          unsigned Func,  
                          unsigned Offset,  
                          Xuint16 Data  
                          )
```

Perform a 16 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus XPci_ConfigOut32( XPci * InstancePtr,  
                          unsigned Bus,  
                          unsigned Device,  
                          unsigned Func,  
                          unsigned Offset,  
                          Xuint32 Data  
                          )
```


Perform a 32 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus XPci_ConfigOut8( XPci * InstancePtr,  
                        unsigned Bus,  
                        unsigned Device,  
                        unsigned Func,  
                        unsigned Offset,  
                        Xuint8 Data  
                        )
```

Perform a 8 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
Xuint32 XPCI_ConfigPack( unsigned Bus,  
                        unsigned Device,  
                        unsigned Function  
                        )
```

Pack configuration address data.

Parameters:

Bus is the PCI bus number. Valid range 0..255.
Device is the PCI device number. Valid range 0..31.
Function is the PCI function number. Valid range 0..7.

Returns:

Encoded Bus, Device & Function formatted to be written to PCI configuration address register.

Note:

None

```
void XPCI_ErrorClear( XPCI * InstancePtr,  
                    XPCIErr * ErrorDataPtr  
                    )
```

Clear errors associated with the PCI bridge. Which errors are cleared depend on the Reason attributes of the ErrorData parameter. For every bit set, that corresponding error is cleared.

XPCI_ErrorGet() and **XPCI_ErrorClear()** are designed to be used in tandem. Use ErrorGet to retrieve the errors, then ErrorClear to clear the error state.

```
XPCI_ErrorGet(ThisInstance, &Errors) if (Errors->IsError) { // Handle error XPCI_ErrorClear(ThisInstance,  
&Errors); }
```

If it is desired to clear some but not all errors, or a specific set of errors, then prepare ErrorData Bitmap attributes appropriately. If it is desired to clear all errors indiscriminately, then use XPCI_CLEAR_ALL_ERRORS. This has the advantage of not requiring the caller to explicitly setup an **XPCIErr** structure.

Parameters:

InstancePtr is the PCI component to operate on.
ErrorDataPtr is used to determine which error conditions to clear. Only the Bitmap attributes are used. Addr attributes of this structure are ignored. If this parameter is set to XPCI_CLEAR_ALL_ERRORS then all errors are cleared.

Note:

If PciSerrReason attribute is set or XPCI_CLEAR_ALL_ERRORS is passed, then the IPIF interrupt status register bits associated with SERR are cleared. This has the same effect as acknowledging an interrupt. If you don't intend on doing this, then clear PciSerrReason before calling XPCI_ErrorClear.

```
void XPCI_ErrorGet( XPCI *      InstancePtr,
                   XPCIError * ErrorDataPtr
                 )
```

Get a snapshot of the PCI bridge's error state, summarize and place results in an **XPCIError** structure. Several bridge registers are read and their contents placed into the structure as follows. Register definitions and their bitmaps are located in **xpci_l.h**:

Attribute	Source Register
-----	-----
LocalBusReason	XPCI_LMADDR_OFFSET
PciReason	XPCI_PIADDR_OFFSET
PciSerrReason	IPIF_IISR
LocalBusReadAddr	XPCI_LMA_R_OFFSET
LocalBusWriteAddr	XPCI_LMA_W_OFFSET
PciSerrReadAddr	XPCI_SERR_R_OFFSET
PciSerrWriteAddr	XPCI_SERR_W_OFFSET
PciReadAddr	XPCI_PIA_R_OFFSET
PciWriteAddr	XPCI_PIA_W_OFFSET

LocalBusReadAddr, LocalBusWriteAddr, PciSerrReadAddr, PciSerrWriteAddr, PCIReadAddr, and PciWriteAddr are all error addresses whose contents are latched at the time of the error.

LocalBusReason and PciReason are present to allow the caller to precisely determine the source of the error. The summary below indicates which bits cause the associated error address to become valid and which interrupt bits from interrupt status register are the cause if the error was reported via an interrupt.

LocalBusReason:

Bit	Error addr is valid	Associated Interrupt bit
-----	-----	-----
XPCI_LMADDR_SERR_R	LocalBusReadAddr	XPCI_IR_LM_SERR_R
XPCI_LMADDR_PERR_R	LocalBusReadAddr	XPCI_IR_LM_PERR_R
XPCI_LMADDR_TA_R	LocalBusReadAddr	XPCI_IR_LM_TA_R
XPCI_LMADDR_SERR_W	LocalBusWriteAddr	XPCI_IR_LM_SERR_W
XPCI_LMADDR_PERR_W	LocalBusWriteAddr	XPCI_IR_LM_PERR_W
XPCI_LMADDR_TA_W	LocalBusWriteAddr	XPCI_IR_LM_TA_W
XPCI_LMADDR_MA_W	LocalBusWriteAddr	XPCI_IR_LM_MA_W
XPCI_LMADDR_BR_W	LocalBusWriteAddr	XPCI_IR_LM_BR_W

XPCI_LMADDR_BRD_W	LocalBusWriteAddr	XPCI_IR_LM_BRD_W
XPCI_LMADDR_BRT_W	LocalBusWriteAddr	XPCI_IR_LM_BRT_W
XPCI_LMADDR_BRANGE_W	LocalBusWriteAddr	XPCI_IR_LM_BRANGE_W

PciReason:

Bit	Error addr is valid	Associated Interrupt bit
-----	-----	-----
XPCI_PIADDR_ERRACK_R	PciReadAddr	N/A
XPCI_PIADDR_ERRACK_W	PciWriteAddr	N/A
XPCI_PIADDR_RETRY_W	PciWriteAddr	N/A
XPCI_PIADDR_TIMEOUT_W	PciWriteAddr	N/A
XPCI_PIADDR_RANGE_W	PciWriteAddr	N/A

PciReasonSerr:

Bit	Error addr is valid	Associated Interrupt bit
-----	-----	-----
XPCI_IR_PI_SERR_R	PciSerrReadAddr	XPCI_IR_PI_SERR_R
XPCI_IR_PI_SERR_W	PciSerrWriteAddr	XPCI_IR_PI_SERR_W

If any of the above mentioned error reason bits are set, then attribute IsError is set to XTRUE. If no errors are detected, then it is set to XFALSE.

Parameters:

InstancePtr is the PCI component to operate on.

ErrorData is the error snapshot data returned from the PCI bridge.

Note:

None

```
void XPCI_GetBusNumber( XPCI * InstancePtr,
                       int * BusNoPtr,
                       int * SubBusNoPtr
                       )
```

Get the bus number and subordinate bus number of the pci bridge.

Parameters:

InstancePtr is the PCI component to operate on
BusNoPtr is storage to place the bus number
SubBusNoPtr is storage to place the subordinate bus number

Note:

None

```
void XPCI_GetDmaImplementation( XPCI * InstancePtr,  
                               Xuint32 * BaseAddr,  
                               Xuint8 * DmaType  
                               )
```

Get the DMA engine implementation information for this instance.

Parameters:

InstancePtr is the PCI component to operate on.
BaseAddr is a return value indicating the base address of the DMA registers.
DmaType is a return value indicating the type of DMA implemented. The possible types are XPCI_DMA_TYPE_NONE for no DMA, XPCI_DMA_TYPE_SIMPLE for simple DMA, and XPCI_DMA_TYPE_SG for scatter-gather DMA.

Note:

None

```
void XPCI_InhibitAfterError( XPCI * InstancePtr,  
                             Xuint32 Mask  
                             )
```

Change how the bridge handles subsequent PCI transactions after errors occur. Transactions can be prohibited once an error occurs then allowed again once the error is cleared. Or transactions are be allowed to continue despite an error condition.

Parameters:

InstancePtr is the PCI component to operate on.

Mask defines the type of transactions affected. OR together bits from XPCI_INHIBIT_* to form the mask. Bits set to 1 will cause transactions to be inhibited when an error exists. Bits set to 0 will allow transactions to proceed.

Note:

None

```
XStatus XPci_Initialize( XPci * InstancePtr,  
                        Xuint16 DeviceId,  
                        int BusNo,  
                        int SubBusNo  
                        )
```

Initialize the **XPci** instance provided by the caller based on the given DeviceID.

Initialization includes setting up the bar registers in the bridge's PCI header to match the IPIF settings. Not performing this step will cause the the IPIF to not respond to PCI bus hits.

Parameters:

InstancePtr is a pointer to an **XPci** instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the component through the **XPci** API must be made with this pointer.

DeviceId is the unique id of the device controlled by this **XPci** component. Passing in a device id associates the generic **XPci** instance to a specific device, as chosen by the caller or application developer.

BusNo is the initial PCI bus number to assign to the host bridge. This value can be changed later with a call to **XPci_SetBusNumber()**

SubBusNo is the initial PCI sub-bus number to assign to the host bridge This value can be changed later with a call to **XPci_SetBusNumber()**

Returns:

- XST_SUCCESS Initialization was successfull.
- XST_DEVICE_NOT_FOUND Device configuration data was not found for a device with the supplied device ID.

Note:

None

```
void XPCI_InterruptClear( XPCI * InstancePtr,  
                          Xuint32 Mask  
                          )
```

Clear device level pending interrupts with the provided mask.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK

Note:

None

```
void XPCI_InterruptDisable( XPCI * InstancePtr,  
                            Xuint32 Mask  
                            )
```

Disable device interrupts. Any component interrupts enabled through [XPCI_InterruptPciEnable\(\)](#) and/or the DMA driver will no longer have any effect. The component interrupt settings will be retained however.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK

Note:

None

```
void XPCI_InterruptEnable( XPCI * InstancePtr,  
                           Xuint32 Mask  
                           )
```

Enable device interrupts. Device interrupts must be enabled by this function before component interrupts enabled by [XPci_InterruptPciEnable\(\)](#) and/or the DMA driver have any effect.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from XPCI_IPIF_INT_MASK.

Note:

None

Xuint32 XPci_InterruptGetEnabled(XPci * InstancePtr)

Returns the device level interrupt enable mask as set by [XPci_InterruptEnable\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK.

Note:

None

Xuint32 XPci_InterruptGetHighestPending(XPci * InstancePtr)

Returns the highest priority pending device interrupt that has been enabled by [XPci_InterruptEnable\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask is one set bit made from XPCI_IPIF_INT_MASK or zero if no interrupts are pending.

Note:

None

Xuint32 XPci_InterruptGetPending(XPci * InstancePtr)

Returns the pending status of device level interrupt signals that have been enabled by [XPci_InterruptEnable\(\)](#). Any bit in the mask set to 1 indicates that an interrupt is pending from the given component

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK or zero if no interrupts are pending.

Note:

None

Xuint32 XPci_InterruptGetStatus(XPci * InstancePtr)

Returns the status of device level interrupt signals. Any bit in the mask set to 1 indicates that the given component has asserted an interrupt condition.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of bits made from XPCI_IPIF_INT_MASK.

Note:

The interrupt status indicates the status of the device regardless if the interrupts from the devices have been enabled or not through [XPci_InterruptEnable\(\)](#).

void XPci_InterruptGlobalDisable(XPci * InstancePtr)

Disable the core's interrupt output signal. Interrupts enabled through [XPci_InterruptEnable\(\)](#) and [XPci_InterruptPciEnable\(\)](#) will no longer be passed through until the IPIF global enable bit is set by [XPci_InterruptGlobalEnable\(\)](#).

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

```
void XPCI_InterruptGlobalEnable( XPCI * InstancePtr)
```

Enable the core's interrupt output signal. Interrupts enabled through [XPCI_InterruptEnable\(\)](#) and [XPCI_InterruptPciEnable\(\)](#) will not be passed through until the IPIF global enable bit is set by this function.

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

```
void XPCI_InterruptPciClear( XPCI * InstancePtr,  
                             Xuint32 Mask  
                             )
```

Clear PCI bridge specific interrupt status bits with the provided mask.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to clear pending interrupts for. Bit positions of 1 are cleared. This mask is formed by OR'ing bits from XPCI_IR_MASK

Note:

None

```
void XPCI_InterruptPciDisable( XPCI * InstancePtr,  
                               Xuint32 Mask  
                               )
```

Disable PCI bridge specific interrupts.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to disable. Bits set to 1 are disabled. The mask is formed by OR'ing bits from XPCI_IR_MASK

Note:

None

```
void XPCI_InterruptPciEnable( XPCI * InstancePtr,  
                             Xuint32 Mask  
                             )
```

Enable PCI bridge specific interrupts. Before this function has any effect in generating interrupts, the function [XPCI_InterruptEnable\(\)](#) must be invoked with the XPCI_IPIF_INT_PCI bit set.

Parameters:

InstancePtr is the PCI component to operate on.

Mask is the mask to enable. Bit positions of 1 are enabled. The mask is formed by OR'ing bits from XPCI_IR_MASK.

Note:

None

```
Xuint32 XPCI_InterruptPciGetEnabled( XPCI * InstancePtr)
```

Get the PCI bridge specific interrupts enabled through [XPCI_InterruptPciEnable\(\)](#). Bits set to 1 mean that interrupt source is enabled.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of enabled bits made from XPCI_IR_MASK.

Note:

None

```
Xuint32 XPCI_InterruptPciGetStatus( XPCI * InstancePtr)
```

Get the status of PCI bridge specific interrupts that have been asserted Bits set to 1 are in an asserted state. Bits may be set to 1 irregardless of whether they have been enabled or not though **XPci_InterruptPciEnable()**. To get the pending interrupts, AND the results of this function with **XPci_InterruptPciGetEnabled()**.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Mask of enabled bits made from XPCI_IR_MASK.

Note:

None

XPci_Config* XPci_LookupConfig(Xuint16 DeviceId)

Lookup the device configuration based on the unique device ID. The table ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceID is the device identifier to lookup.

Returns:

- o **XEmc** configuration structure pointer if DeviceID is found.
- o XNULL if DeviceID is not found.

void XPci_Reset(XPci * InstancePtr)

Reset the PCI IP core. This is a destructive operation that could cause loss of data, local bus errors, or PCI bus errors if reset occurs while a transaction is pending.

Parameters:

InstancePtr is the PCI component to operate on.

Note:

None

XStatus XPci_SelfTest(XPci * InstancePtr)

Run a self-test on the driver/device. This includes the following tests:

- Configuration read of the bridge device and vendor ID.

Parameters:

InstancePtr is a pointer to the **XPci** instance to be worked on. This parameter must have been previously initialized with **XPci_Initialize()**.

Returns:

- XST_SUCCESS If test passed
- XST_FAILURE If test failed

Note:

None

```
void XPci_SetBusNumber( XPci * InstancePtr,  
                        int    BusNo,  
                        int    SubBusNo  
                        )
```

Set the bus number and subordinate bus number of the pci bridge. This function has effect only if the PCI bridge is configured as a PCI host bridge.

Parameters:

InstancePtr is the PCI component to operate on.

BusNo is the bus number to set

SubBusNo is the subordinate bus number to set

Note:

None

```
void XPci_SpecialCycle( XPci * InstancePtr,  
                        Xuint32 Data  
                        )
```

Broadcasts a message to all listening PCI targets.

Parameters:

InstancePtr is the PCI component to operate on.

Data is the data to broadcast.

Note:

None

Xuint32 XPCI_V3StatusCommandGet(XPCI * InstancePtr)

Read the contents of the V3 bridge's status & command register. This same register can be retrieved by a PCI configuration access. The register can be written only with a PCI configuration access.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Contents of the V3 bridge's status and command register

Note:

None

**void XPCI_V3TransactionStatusClear(XPCI * InstancePtr,
Xuint32 Data
)**

Clear status bits in the V3 bridge's transaction status register. The contents of this register can be decoded using XPCI_STATV3_* constants defined in [xpci_1.h](#).

Parameters:

InstancePtr is the PCI component to operate on.

Data is the contents to write to the register. Or XPCI_STATV3_* constants for those bits to be cleared. Bits in the register that are read-only are not affected.

Note:

None

Xuint32 XPCI_V3TransactionStatusGet(XPCI * InstancePtr)

Read the contents of the V3 bridge's transaction status register. The contents of this register can be decoded using XPCI_STATV3_* constants defined in [xpci_l.h](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Contents of the V3 bridge's transaction status register.

Note:

None

pci/v1_00_a/src/xpci_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xpci_l.h,v 1.6 2003/05/13 13:36:22 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file pci/v1_00_a/src/xpci_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  High-level driver functions
00029 * are defined and more PCI documentation is in xpci.h.
00030 *
00031 * PCI configuration read/write macro functions can be changed so that data is
00032 * swapped before being written to the configuration address/data registers.
00033 * As delivered in this file, these macros do not swap.  Change the definitions
00034 * of XIo_InPci() and XIo_OutPci() in this file to suit system needs.
00035 *
00036 * <pre>
00037 * MODIFICATION HISTORY:
00038 *
00039 * Ver   Who   Date       Changes
00040 * -----
00041 * 1.00a rmm   05/19/02 First release
00042 * </pre>
```



```

00043 *
00044 *****/
00045
00046 #ifndef XPCI_L_H /* prevent circular inclusions */
00047 #define XPCI_L_H /* by using protection macros */
00048
00049 /***** Include Files *****/
00050
00051 #include "xbasic_types.h"
00052 #include "xio.h"
00053 #include "xipif_v1_23_b.h"
00054
00055 /***** Constant Definitions *****/
00056
00057 /** @name Registers
00058 *
00059 * Register offsets for this device. Note that the following IPIF registers
00060 * are implemented. Macros are defined to specifically access these registers
00061 * without knowing which version of IPIF being used.
00062 * @{
00063 */
00064 #define XPCI_PREOVRD_OFFSET 0x00000100 /**< Prefetch override */
00065 #define XPCI_IAR_OFFSET 0x00000104 /**< PCI interrupt ack */
00066 #define XPCI_SC_DATA_OFFSET 0x00000108 /**< Special cycle data */
00067 #define XPCI_CAR_OFFSET 0x0000010C /**< Config addr reg (port) */
00068 #define XPCI_CDR_OFFSET 0x00000110 /**< Config command data */
00069 #define XPCI_BUSNO_OFFSET 0x00000114 /**< bus/subordinate bus numbers */
00070 #define XPCI_STATCMD_OFFSET 0x00000118 /**< PCI config status/command */
00071 #define XPCI_STATV3_OFFSET 0x0000011C /**< V3 core transaction status */
00072 #define XPCI_INHIBIT_OFFSET 0x00000120 /**< Inhibit transfers on errors */
00073 #define XPCI_LMADDR_OFFSET 0x00000124 /**< Local bus master address definition
*/
00074 #define XPCI_LMA_R_OFFSET 0x00000128 /**< Local bus master read error address
*/
00075 #define XPCI_LMA_W_OFFSET 0x0000012C /**< Local bus master write error
address */
00076 #define XPCI_SERR_R_OFFSET 0x00000130 /**< PCI initiator read SERR address */
00077 #define XPCI_SERR_W_OFFSET 0x00000134 /**< PCI initiator write SERR address */
00078 #define XPCI_PIADDR_OFFSET 0x00000138 /**< PCI address definition */
00079 #define XPCI_PIA_R_OFFSET 0x0000013C /**< PCI read error address */
00080 #define XPCI_PIA_W_OFFSET 0x00000140 /**< PCI write error address */
00081 /*@}*/
00082
00083 /**
00084 * <pre>
00085 * The following IPIF registers are implemented. Macros are defined to
00086 * specifically access these registers without knowing which version of IPIF
00087 * is being used
00088 *
00089 * XIIF_Version_DISR_OFFSET Device interrupt status

```

```

00090 *   XIIF_Version_DIPR_OFFSET      Device interrupt pending
00091 *   XIIF_Version_DIER_OFFSET      Device interrupt enable
00092 *   XIIF_Version_DIIR_OFFSET      Device interrupt ID
00093 *   XIIF_Version_DGIER_OFFSET      Device global interrupt enable
00094 *   XIIF_Version_IISR_OFFSET      Bridge Interrupt status
00095 *   XIIF_Version_IIER_OFFSET      Bridge Interrupt enable
00096 *   XIIF_Version_RESETR_OFFSET    Reset (1)
00097 *
00098 * (1) This version of the PCI core places the RESETR at a nonstandard location
00099 * of 0x80 instead of 0x40. The XPCI_mReset() macro has been adjusted.
00100 * </pre>
00101 */
00102
00103 /** @name Interrupt Status and Enable Register bitmaps and masks
00104 *
00105 * Bit definitions for the interrupt status register and interrupt enable
00106 * registers.
00107 * @{
00108 */
00109 #define XPCI_IR_MASK                0x00001FFF /**< Mask of all bits */
00110 #define XPCI_IR_LM_SERR_R           0x00000001 /**< Local bus master read SERR */
00111 #define XPCI_IR_LM_PERR_R           0x00000002 /**< Local bus master read PERR */
00112 #define XPCI_IR_LM_TA_R             0x00000004 /**< Local bus master read target abort
*/
00113 #define XPCI_IR_LM_SERR_W           0x00000008 /**< Local bus master write SERR */
00114 #define XPCI_IR_LM_PERR_W           0x00000010 /**< Local bus master write PERR */
00115 #define XPCI_IR_LM_TA_W             0x00000020 /**< Local bus master write target abort
*/
00116 #define XPCI_IR_LM_MA_W             0x00000040 /**< Local bus master abort write */
00117 #define XPCI_IR_LM_BR_W             0x00000080 /**< Local bus master burst write retry
*/
00118 #define XPCI_IR_LM_BRD_W            0x00000100 /**< Local bus master burst write retry
00119                                     disconnect */
00120 #define XPCI_IR_LM_BRT_W            0x00000200 /**< Local bus master burst write retry
00121                                     timeout */
00122 #define XPCI_IR_LM_BRANGE_W         0x00000400 /**< Local bus master burst write range
*/
00123 #define XPCI_IR_PI_SERR_R           0x00000800 /**< PCI initiator read SERR */
00124 #define XPCI_IR_PI_SERR_W           0x00001000 /**< PCI initiator write SERR */
00125 /*@}*/
00126
00127 /** @name Inhibit transfers on errors register bitmaps and masks.
00128 *
00129 * These bits control whether subsequent PCI transactions are allowed after
00130 * an error occurs. Bits set to 1 inhibit further transactions, while bits
00131 * set to 0 allow further transactions after an error.
00132 * @{
00133 */
00134 #define XPCI_INHIBIT_MASK           0x0000000F /**< Mask for all bits defined
below */
00135 #define XPCI_INHIBIT_LOCAL_BUS_R    0x00000001 /**< Local bus master reads */

```

```

00136 #define XPCI_INHIBIT_LOCAL_BUS_W 0x00000002 /**< Local bus mater writes */
00137 #define XPCI_INHIBIT_PCI_R      0x00000004 /**< PCI initiator reads */
00138 #define XPCI_INHIBIT_PCI_W      0x00000008 /**< PCI initiator writes */
00139 /*@}*/
00140
00141 /** @name PCI configuration status & command register bitmaps and masks.
00142  *
00143  * Bit definitions for the PCI configuration status & command register. The
00144  * definition of this register is standard for PCI devices.
00145  * @{
00146  */
00147 #define XPCI_STATCMD_IO_EN        0x00000001 /**< I/O access enable */
00148 #define XPCI_STATCMD_MEM_EN      0x00000002 /**< Memory access enable */
00149 #define XPCI_STATCMD_BUSM_EN     0x00000004 /**< Bus master enable */
00150 #define XPCI_STATCMD_SPECIALCYC  0x00000008 /**< Special cycles */
00151 #define XPCI_STATCMD_MEMWR_INV_EN 0x00000010 /**< Memory write & invalidate */
00152 #define XPCI_STATCMD_VGA_SNOOP_EN 0x00000020 /**< VGA palette snoop enable */
00153 #define XPCI_STATCMD_PARITY      0x00000040 /**< Report parity errors */
00154 #define XPCI_STATCMD_STEPPING    0x00000080 /**< Stepping control */
00155 #define XPCI_STATCMD_SERR_EN     0x00000100 /**< SERR report enable */
00156 #define XPCI_STATCMD_BACK_EN     0x00000200 /**< Fast back-to-back enable */
00157 #define XPCI_STATCMD_INT_DISABLE 0x00000400 /**< Interrupt disable (PCI v2.3)
*/
00158 #define XPCI_STATCMD_INT_STATUS  0x00100000 /**< Interrupt status (PCI v2.3)
*/
00159 #define XPCI_STATCMD_66MHZ_CAP   0x00200000 /**< 66MHz capable */
00160 #define XPCI_STATCMD_MPERR       0x01000000 /**< Master data PERR detected */
00161 #define XPCI_STATCMD_DEVSEL_MSK  0x06000000 /**< Device select timing mask */
00162 #define XPCI_STATCMD_DEVSEL_FAST 0x04000000 /**< Device select timing fast */
00163 #define XPCI_STATCMD_DEVSEL_MED  0x02000000 /**< Device select timing medium
*/
00164 #define XPCI_STATCMD_TGTABRT_SIG 0x08000000 /**< Signaled target abort */
00165 #define XPCI_STATCMD_TGTABRT_RCV 0x10000000 /**< Received target abort */
00166 #define XPCI_STATCMD_MSTABRT_RCV 0x20000000 /**< Received master abort */
00167 #define XPCI_STATCMD_SERR_SIG    0x40000000 /**< Signaled SERR */
00168 #define XPCI_STATCMD_PERR_DET    0x80000000 /**< Detected PERR */
00169 #define XPCI_STATCMD_ERR_MASK    (XPCI_STATCMD_PERR_DET |      \
00170                                  XPCI_STATCMD_SERR_SIG |      \
00171                                  XPCI_STATCMD_MSTABRT_RCV |    \
00172                                  XPCI_STATCMD_TGTABRT_RCV |    \
00173                                  XPCI_STATCMD_TGTABRT_SIG |    \
00174                                  XPCI_STATCMD_MPERR) /**< Error bits or'd
together */
00175 /*@}*/
00176
00177 /** @name V3 core transaction status register bitmaps and masks.
00178  *
00179  * Bit definitions for the V3 core transaction status register. This register
00180  * consists of status information on V3 core transactions.

```

```

00181 * @{
00182 */
00183 #define XPCI_STATV3_MASK          0x000007FF /**< Mask of all bits */
00184 #define XPCI_STATV3_DATA_XFER     0x00000001 /**< Data transfer. Read only */
00185 #define XPCI_STATV3_TRANS_END     0x00000002 /**< Transaction end. Read only */
00186 #define XPCI_STATV3_NORM_TERM     0x00000004 /**< Normal termination. Read only
*/
00187 #define XPCI_STATV3_TGT_TERM      0x00000008 /**< Target termination. Read only
*/
00188 #define XPCI_STATV3_DISC_WODATA   0x00000010 /**< Disconnect without data. Read
only */
00189 #define XPCI_STATV3_DISC_WDATA   0x00000020 /**< Disconnect with data. Read
only */
00190 #define XPCI_STATV3_TGT_ABRT     0x00000040 /**< Target abort. Read only */
00191 #define XPCI_STATV3_MASTER_ABRT  0x00000080 /**< Master abort. Read only */
00192 #define XPCI_STATV3_PCI_RETRY_R   0x00000100 /**< PCI retry on read. Read/write
*/
00193 #define XPCI_STATV3_PCI_RETRY_W   0x00000200 /**< PCI retry on write. Read/write
*/
00194 #define XPCI_STATV3_WRITE_BUSY    0x00000400 /**< Write busy. Read only */
00195 /*@}*/
00196
00197
00198 /** @name Bus number and subordinate bus number register bitmaps and masks
00199 *
00200 * @{
00201 */
00202 #define XPCI_BUSNO_BUS_MASK       0x00FF0000 /**< Mask for bus number */
00203 #define XPCI_BUSNO_SUBBUS_MASK    0x000000FF /**< Mask for subordinate bus no */
00204 /*@}*/
00205
00206
00207 /** @name Local bus master address register bitmaps and masks
00208 *
00209 * Bit definitions for the local bus master address definition register.
00210 * This register defines the meaning of the address stored in the local bus
00211 * master read (XPCI_LMA_R_OFFSET) and master write error (XPCI_LMA_W_OFFSET)
00212 * registers.
00213 * @{
00214 */
00215 #define XPCI_LMADDR_MASK          0x000007FF /**< Mask of all bits */
00216 #define XPCI_LMADDR_SERR_R        0x00000001 /**< Master read SERR */
00217 #define XPCI_LMADDR_PERR_R        0x00000002 /**< Master read PERR */
00218 #define XPCI_LMADDR_TA_R          0x00000004 /**< Master read target abort */
00219 #define XPCI_LMADDR_SERR_W        0x00000008 /**< Master write SERR */
00220 #define XPCI_LMADDR_PERR_W        0x00000010 /**< Master write PERR */
00221 #define XPCI_LMADDR_TA_W          0x00000020 /**< Master write target abort */
00222 #define XPCI_LMADDR_MA_W          0x00000040 /**< Master abort write */
00223 #define XPCI_LMADDR_BR_W          0x00000080 /**< Master burst write retry */
00224 #define XPCI_LMADDR_BRD_W         0x00000100 /**< Master burst write retry

```

```

disconnect */
00225 #define XPCI_LMADDR_BRT_W          0x00000200 /**< Master burst write retry
timeout */
00226 #define XPCI_LMADDR_BRANGE_W     0x00000400 /**< Master burst write range */
00227 /*}*/
00228
00229 /** @name PCI error address definition bitmaps and masks
00230 *
00231 * Bit definitions for the PCI address definition register. This register
00232 * defines the meaning of the address stored in the PCI read error address
00233 * (XPCI_PIA_R_OFFSET) and PCI write error address (XPCI_PIA_W_OFFSET)
registers.
00234 * @{
00235 */
00236 #define XPCI_PIADDR_MASK             0x0000001F /**< Mask of all bits */
00237 #define XPCI_PIADDR_ERRACK_R         0x00000001 /**< PCI initiator read ErrAck */
00238 #define XPCI_PIADDR_ERRACK_W         0x00000002 /**< PCI initiator write ErrAck */
00239 #define XPCI_PIADDR_RETRY_W          0x00000004 /**< PCI initiator write retries */
00240 #define XPCI_PIADDR_TIMEOUT_W        0x00000008 /**< PCI initiator write timeout */
00241 #define XPCI_PIADDR_RANGE_W          0x00000010 /**< PCI initiator write range */
00242 /*}*/
00243
00244
00245 /** @name PCI configuration header offsets
00246 *
00247 * Defines the offsets in the standard PCI configuration header
00248 * @{
00249 */
00250 #define XPCI_HDR_VENDOR               0x00 /**< Vendor ID */
00251 #define XPCI_HDR_DEVICE               0x02 /**< Device ID */
00252 #define XPCI_HDR_COMMAND              0x04 /**< Command register */
00253 #define XPCI_HDR_STATUS               0x06 /**< Status register */
00254 #define XPCI_HDR_REVID                0x08 /**< Revision ID */
00255 #define XPCI_HDR_CLASSCODE            0x09 /**< Class code */
00256 #define XPCI_HDR_CACHE_LINE_SZ        0x0C /**< Cache line size */
00257 #define XPCI_HDR_LATENCY              0x0D /**< Latency timer */
00258 #define XPCI_HDR_TYPE                 0x0E /**< Header type */
00259 #define XPCI_HDR_BIST                 0x0F /**< Built in self test */
00260 #define XPCI_HDR_BAR0                 0x10 /**< Base address 0 */
00261 #define XPCI_HDR_BAR1                 0x14 /**< Base address 1 */
00262 #define XPCI_HDR_BAR2                 0x18 /**< Base address 2 */
00263 #define XPCI_HDR_BAR3                 0x1C /**< Base address 3 */
00264 #define XPCI_HDR_BAR4                 0x20 /**< Base address 4 */
00265 #define XPCI_HDR_BAR5                 0x24 /**< Base address 5 */
00266 #define XPCI_HDR_CARDBUS_PTR          0x28 /**< Cardbus CIS pointer */
00267 #define XPCI_HDR_SUB_VENDOR           0x2C /**< Subsystem Vendor ID */
00268 #define XPCI_HDR_SUB_DEVICE           0x2E /**< Subsystem ID */
00269 #define XPCI_HDR_ROM_BASE             0x30 /**< Expansion ROM base address */
00270 #define XPCI_HDR_CAP_PTR              0x34 /**< Capabilities pointer */

```

```

00271 #define XPCI_HDR_INT_LINE          0x3C /**< Interrupt line */
00272 #define XPCI_HDR_INT_PIN           0x3D /**< Interrupt pin */
00273 #define XPCI_HDR_MIN_GNT           0x3E /**< Timeslice request */
00274 #define XPCI_HDR_MAX_LAT           0x3F /**< Priority level request */
00275 /*@}*/
00276
00277
00278 /** @name PCI BAR register definitions
00279  *
00280  * Defines the masks and bits for the PCI XPAR_HDR_BARn registers
00281  * The bridge supports the first three BARs in the PCI configuration header
00282  * @{
00283  */
00284 #define XPCI_HDR_NUM_BAR            6          /**< Number of BARs in the PCI
header */
00285 #define XPCI_HDR_BAR_ADDR_MASK      0xFFFFFFF0 /**< Base address mask */
00286 #define XPCI_HDR_BAR_PREFETCH_YES  0x00000008 /**< Range is prefetchable */
00287 #define XPCI_HDR_BAR_PREFETCH_NO   0x00000000 /**< Range is not prefetchable
*/
00288 #define XPCI_HDR_BAR_TYPE_MASK      0x00000006 /**< Memory type mask */
00289 #define XPCI_HDR_BAR_TYPE_BELOW_4GB 0x00000000 /**< Locate anywhere below 4GB
*/
00290 #define XPCI_HDR_BAR_TYPE_BELOW_1MB 0x00000002 /**< Reserved in PCI 2.2 */
00291 #define XPCI_HDR_BAR_TYPE_ABOVE_4GB 0x00000004 /**< Locate anywhere above 4GB
*/
00292 #define XPCI_HDR_BAR_TYPE_RESERVED  0x00000006 /**< Reserved */
00293 #define XPCI_HDR_BAR_SPACE_IO        0x00000001 /**< IO space indicator */
00294 #define XPCI_HDR_BAR_SPACE_MEMORY    0x00000000 /**< Memory space indicator */
00295 /*@}*/
00296
00297
00298 /** @name DMA type constants
00299  *
00300  * Defines the types of DMA engines
00301  * @{
00302  */
00303 #define XPCI_DMA_TYPE_NONE           9 /**< No DMA */
00304 #define XPCI_DMA_TYPE_SIMPLE         0 /**< Simple DMA */
00305 #define XPCI_DMA_TYPE_SG             1 /**< Scatter-gather DMA */
00306 /*@}*/
00307
00308 /*
00309  * Default IPIF Device interrupt enable (XIIF_Version_IIER_OFFSET),
00310  * status registers (XIIF_Version_IISR_OFFSET), and pending register
00311  * (XIIF_Version_IIPR_OFFSET)
00312  */
00313 #define XPCI_IPIF_INT_PCI            0x00000004 /* PCI core interrupt */
00314 #define XPCI_IPIF_INT_DMA            0x00000008 /* DMA interrupt */
00315
00316 /* Define IPIF interrupt mask */

```

```

00317 #define XPCI_IPIF_INT_MASK (XPCI_IPIF_INT_PCI | XPCI_IPIF_INT_DMA)
00318
00319 /* Define value of IPIF DIIR register when no interrupt is pending */
00320 #define XPCI_IPIF_INT_NONE_PENDING  XIIF_V123B_NO_INTERRUPT_ID
00321
00322 /*
00323  * These macros define how IO access to the CAR/CDR are made
00324  * Use XIo_In/Out32 normally or XIo_InSwap/OutSwap32 if these registers
00325  * present data in an endianness opposite processor's.
00326  */
00327 #define XIo_InPci(InAddress)          XIo_In32(InAddress)
00328 #define XIo_OutPci(OutAddress, Data) XIo_Out32(OutAddress, Data)
00329
00330
00331 /*****
00332  *
00333  * Low-level driver macros and functions. The list below provides signatures
00334  * to help the user use the macros.
00335  *
00336  * void      XPci_mReset(Xuint32 BaseAddress)
00337  * void      XPci_mIntrGlobalEnable(Xuint32 BaseAddress)
00338  * void      XPci_mIntrGlobalDisable(Xuint32 BaseAddress)
00339  * void      XPci_mIntrEnable(Xuint32 BaseAddress, Mask)
00340  * void      XPci_mIntrDisable(Xuint32 BaseAddress, Mask)
00341  * void      XPci_mIntrClear(Xuint32 BaseAddress, Xuint32 Mask)
00342  * Xuint32   XPci_mIntrReadIER(Xuint32 BaseAddress)
00343  * Xuint32   XPci_mIntrReadISR(Xuint32 BaseAddress)
00344  * void      XPci_mIntrWriteISR(Xuint32 BaseAddress, Xuint32 Mask)
00345  * Xuint32   XPci_mIntrReadIPR(Xuint32 BaseAddress)
00346  * Xuint32   XPci_mIntrReadID(Xuint32 BaseAddress)
00347  * void      XPci_mIntrPciEnable(Xuint32 BaseAddress, Xuint32 Mask)
00348  * void      XPci_mIntrPciDisable(Xuint32 BaseAddress, Xuint32 Mask)
00349  * void      XPci_mIntrPciClear(Xuint32 BaseAddress, Xuint32 Mask)
00350  * Xuint32   XPci_mIntrPciReadIER(Xuint32 BaseAddress)
00351  * Xuint32   XPci_mIntrPciReadISR(Xuint32 BaseAddress)
00352  * void      XPci_mIntrPciWriteISR(Xuint32 BaseAddress, Xuint32 Mask)
00353  * Xuint32   XPci_mReadReg(Xuint32 BaseAddress, Xuint32 RegOffset)
00354  * void      XPci_mWriteReg(Xuint32 BaseAddress, Xuint32 RegOffset, Xuint32 Data)
00355  * void      XPci_mConfigIn(Xuint32 BaseAddress, Xuint32 ConfigAddress,
00356  *                          Xuint32 ConfigData)
00357  * void      XPci_mConfigOut(Xuint32 BaseAddress, Xuint32 ConfigAddress,
00358  *                          Xuint32 ConfigData)
00359  * void      XPci_mAckSend(Xuint32 BaseAddress, Xuint32 Vector)
00360  * Xuint32   XPci_mAckRead(Xuint32 BaseAddress)
00361  * Xuint32   XPci_mSpecialCycle(Xuint32 BaseAddress, Xuint32 Data)
00362  * Xuint32   XPci_mLocal2Pci(Xuint32 LocalAddress, Xuint32 TranslationOffset)
00363  * Xuint32   XPci_mPci2Local(Xuint32 PciAddress, Xuint32 TranslationOffset)
00364  *
00365  *****/
00366
00367
00368 /**

```

```

00369 * IPIF Low level PCI reset function. Reset the V3 core.
00370 *
00371 * C-style signature:
00372 *     void XPCI_mReset(Xuint32 BaseAddress)
00373 *
00374 * @param BaseAddress is the base address of the PCI component.
00375 *
00376 * @return None
00377 *
00378 * @note The IPIF RESETR register is located at (base + 0x80) instead of
00379 * (base + 0x40) where the IPIF component driver expects it. This macro adjusts
00380 * for this difference.
00381
*****
00382 #define XPCI_mReset(BaseAddress) \
00383     XIIF_V123B_RESET((BaseAddress + 0x40))
00384
00385
00386
/*****
00387 /**
00388 * Global interrupt enable. Must be called before any interrupts enabled by
00389 * XPCI_mIntrEnable() or XPCI_mIntrPciEnable() have any effect.
00390 *
00391 * C-style signature:
00392 *     void XPCI_mIntrGlobalEnable(Xuint32 BaseAddress)
00393 *
00394 * @param BaseAddress is the base address of the PCI component.
00395 *
00396 * @return None
00397 *
00398 * @note None
00399
*****
00400 #define XPCI_mIntrGlobalEnable(BaseAddress) \
00401     XIIF_V123B_GINTR_ENABLE((BaseAddress))
00402
00403
00404
/*****
00405 /**
00406 * Global interrupt disable. Disable all interrupts from this core. Any
00407 * interrupts enabled by XPCI_mIntrEnable() or XPCI_mIntrPciEnable()
00408 * are disabled, however their settings remain unchanged.
00409 *
00410 * C-style signature:
00411 *     void XPCI_mIntrGlobalDisable(Xuint32 BaseAddress)
00412 *
00413 * @param BaseAddress is the base address of the PCI component.
00414 *
00415 * @return None
00416 *

```



```
00417 * @note None
00418
*****
00419 #define XPCI_mIntrGlobalDisable(BaseAddress) \
00420     XIIF_V123B_GINTR_DISABLE((BaseAddress))
00421
00422
00423
/*****
00424 /**
00425 * Enable interrupts in the device interrupt enable register (DIER)
00426 *
00427 * C-style signature:
00428 *     void XPCI_mIntrEnable(Xuint32 BaseAddress, Mask)
00429 *
00430 * @param BaseAddress is the base address of the PCI component.
00431 * @param Mask is the group of interrupts to enable. Use a logical OR of
00432 *     constants in XPCI_IPIF_INT_MASK. Bits set to 1 are enabled,
00433 *     bits set to 0 are not affected.
00434 *
00435 * @return None
00436 *
00437 * @note None
00438
*****
00439 #define XPCI_mIntrEnable(BaseAddress, Mask) \
00440 { \
00441     Xuint32 Temp; \
00442     Temp = XIIF_V123B_READ_DIER((BaseAddress)); \
00443     Temp |= (Mask); \
00444     XIIF_V123B_WRITE_DIER((BaseAddress), (Temp)); \
00445 }
00446
00447
00448
/*****
00449 /**
00450 * Disable interrupts in the device interrupt enable register (DIER)
00451 *
00452 * C-style signature:
00453 *     void XPCI_mIntrDisable(Xuint32 BaseAddress, Mask)
00454 *
00455 * @param BaseAddress is the base address of the PCI component.
00456 * @param Mask is the group of interrupts to disable. Use a logical OR of
00457 *     constants in XPCI_IPIF_INT_MASK. Bits set to 1 are disabled,
00458 *     bits set to 0 are not affected.
00459 *
00460 * @return None
00461 *
00462 * @note None
00463
*****
```

```

00464 #define XPCI_mIntrDisable(BaseAddress, Mask) \
00465 { \
00466     Xuint32 Temp; \
00467     Temp = XIIF_V123B_READ_DIER((BaseAddress)); \
00468     Temp &= ~(Mask); \
00469     XIIF_V123B_WRITE_DIER((BaseAddress), (Temp)); \
00470 }
00471
00472
00473
/*****/
00474 /**
00475 * Clear pending interrupts in the device interrupt status register (DISR)
00476 * This is a toggle on write register.
00477 *
00478 * C-style signature:
00479 *     void XPCI_mIntrClear(Xuint32 BaseAddress, Xuint32 Mask)
00480 *
00481 * @param BaseAddress is the base address of the PCI component.
00482 * @param Mask is the group of interrupts to clear. Use a logical OR of
00483 *     constants in XPCI_IPIF_INT_MASK. Bits set to 1 are cleared,
00484 *     bits set to 0 are not affected.
00485 *
00486 * @return None
00487 *
00488 * @note None
00489
*****/
00490 #define XPCI_mIntrClear(BaseAddress, Mask) \
00491     XIIF_V123B_WRITE_DISR((BaseAddress), \
00492         XIIF_V123B_READ_DISR((BaseAddress)) & (Mask))
00493
00494
00495
/*****/
00496 /**
00497 * Read the contents of the device interrupt enable register (DIER)
00498 *
00499 * C-style signature:
00500 *     Xuint32 XPCI_mIntrReadIER(Xuint32 BaseAddress)
00501 *
00502 * @param BaseAddress is the base address of the PCI component.
00503 *
00504 * @return Value of the register.
00505 *
00506 * @note None
00507
*****/
00508 #define XPCI_mIntrReadIER(BaseAddress) \
00509     XIIF_V123B_READ_DIER((BaseAddress))
00510
00511

```

```

00512
/*****
00513 /**
00514 * Read the contents of the device interrupt status register (DISR)
00515 *
00516 * C-style signature:
00517 *     Xuint32 XPCI_mIntrReadISR(Xuint32 BaseAddress)
00518 *
00519 * @param BaseAddress is the base address of the PCI component.
00520 *
00521 * @return Value of the register.
00522 *
00523 * @note None
00524
*****
00525 #define XPCI_mIntrReadISR(BaseAddress) \
00526     XIIF_V123B_READ_DISR((BaseAddress))
00527
00528
00529
/*****
00530 /**
00531 * Write to the device interrupt status register (DISR)
00532 *
00533 * C-style signature:
00534 *     void XPCI_mIntrWriteISR(Xuint32 BaseAddress, Xuint32 Mask)
00535 *
00536 * @param BaseAddress is the base address of the PCI component.
00537 * @param Mask is the value to write to the register and is assumed to be
00538 *     bits or'd together from the XPCI_IPIF_INT_MASK.
00539 *
00540 * @return None
00541 *
00542 * @note None
00543
*****
00544 #define XPCI_mIntrWriteISR(BaseAddress, Mask) \
00545     XIIF_V123B_WRITE_DISR((BaseAddress), (Mask))
00546
00547
00548
/*****
00549 /**
00550 * Read the contents of the device interrupt pending register (DIPR).
00551 *
00552 * C-style signature:
00553 *     Xuint32 XPCI_mIntrReadIPR(Xuint32 BaseAddress)
00554 *
00555 * @param BaseAddress is the base address of the PCI component.
00556 *
00557 * @return Value of the register.
00558 *

```

```

00559 * @note None
00560
*****
00561 #define XPCI_mIntrReadIPR(BaseAddress) \
00562     XIIF_V123B_READ_DIPR((BaseAddress))
00563
00564
00565
/*****
00566 /**
00567 * Read the contents of the device interrupt ID register (DIIR).
00568 *
00569 * C-style signature:
00570 *     Xuint32 XPCI_mIntrReadID(Xuint32 BaseAddress)
00571 *
00572 * @param BaseAddress is the base address of the PCI component.
00573 *
00574 * @return Value of the register.
00575 *
00576 * @note None
00577
*****
00578 #define XPCI_mIntrReadID(BaseAddress) \
00579     XIIF_V123B_READ_DIIR((BaseAddress))
00580
00581
00582
/*****
00583 /**
00584 * Enable PCI specific interrupt sources in the PCI interrupt enable register
00585 * (IIER)
00586 *
00587 * C-style signature:
00588 *     void XPCI_mIntrPciEnable(Xuint32 BaseAddress, Xuint32 Mask)
00589 *
00590 * @param BaseAddress is the base address of the PCI component.
00591 * @param Mask is the group of interrupts to enable. Bits set to 1 are enabled,
00592 *     bits set to 0 are not affected. The mask is made up by OR'ing bits
00593 *     from XPCI_IR_MASK.
00594 *
00595 * @return None
00596 *
00597 * @note None
00598
*****
00599 #define XPCI_mIntrPciEnable(BaseAddress, Mask) \
00600 { \
00601     Xuint32 Temp; \
00602     Temp = XIIF_V123B_READ_IIER((BaseAddress)); \
00603     Temp |= (Mask); \
00604     XIIF_V123B_WRITE_IIER((BaseAddress), (Temp)); \
00605 }

```

```

00606
00607
00608
/*****
00609 /**
00610 * Disable PCI specific interrupt sources in the PCI interrupt enable register
00611 * (IIER)
00612 *
00613 * C-style signature:
00614 *     void XPCI_mIntrPciDisable(Xuint32 BaseAddress, Xuint32 Mask)
00615 *
00616 * @param BaseAddress is the base address of the PCI component.
00617 * @param Mask is the group of interrupts to disable. Bits set to 1 are
disabled,
00618 *         bits set to 0 are not affected. The mask is made up by OR'ing bits
00619 *         from XPCI_IR_MASK.
00620 *
00621 * @return None
00622 *
00623 * @note None
00624
*****/
00625 #define XPCI_mIntrPciDisable(BaseAddress, Mask) \
00626 { \
00627     Xuint32 Temp; \
00628     Temp = XIIF_V123B_READ_IIER((BaseAddress)); \
00629     Temp &= ~(Mask); \
00630     XIIF_V123B_WRITE_IIER((BaseAddress), (Temp)); \
00631 }
00632
00633
00634
/*****
00635 /**
00636 * Clear PCI specific interrupts in the interrupt status register (IISR).
00637 * This is a toggle on write register.
00638 *
00639 * C-style signature:
00640 *     void XPCI_mIntrPciClear(Xuint32 BaseAddress, Xuint32 Mask)
00641 *
00642 * @param BaseAddress is the base address of the PCI component.
00643 * @param Mask is the group of interrupts to clear. Use a logical OR of
00644 *         constants in XPCI_IR_MASK. Bits set to 1 are cleared,
00645 *         bits set to 0 are not affected.
00646 *
00647 * @return None
00648 *
00649 * @note None
00650
*****/
00651 #define XPCI_mIntrPciClear(BaseAddress, Mask) \
00652     XIIF_V123B_WRITE_IISR((BaseAddress), \

```

```

00653             XIIF_V123B_READ_IISR((BaseAddress)) & (Mask))
00654
00655
00656
/*****/
00657 /**
00658 * Read the contents of the PCI specific interrupt enable register (IIER)
00659 *
00660 * C-style signature:
00661 *     Xuint32 XPCI_mIntrPciReadIER(Xuint32 BaseAddress)
00662 *
00663 * @param BaseAddress is the base address of the PCI component.
00664 *
00665 * @return Contents of the pending interrupt register. The mask is made up of
00666 *         bits defined in XPCI_IR_MASK.
00667 *
00668 * @note None
00669
*****/
00670 #define XPCI_mIntrPciReadIER(BaseAddress) \
00671     XIIF_V123B_READ_IIER((BaseAddress))
00672
00673
00674
/*****/
00675 /**
00676 * Read the contents of the PCI specific interrupt status register (IISR)
00677 *
00678 * C-style signature:
00679 *     Xuint32 XPCI_mIntrPciReadISR(Xuint32 BaseAddress)
00680 *
00681 * @param BaseAddress is the base address of the PCI component.
00682 *
00683 * @return
00684 * Contents of the pending interrupt register. The mask is made up of
00685 * bits defined in XPCI_IR_MASK.
00686 *
00687 * @note None
00688
*****/
00689 #define XPCI_mIntrPciReadISR(BaseAddress) \
00690     XIIF_V123B_READ_IISR((BaseAddress))
00691
00692
00693
/*****/
00694 /**
00695 * Write to the PCI interrupt status register (IISR)
00696 *
00697 * C-style signature:
00698 *     void XPCI_mIntrPciWriteISR(Xuint32 BaseAddress, Xuint32 Mask)
00699 *

```

```

00700 * @param BaseAddress is the base address of the PCI component.
00701 * @param Mask is the value to write to the register and is assumed to be
00702 *         bits or'd together from the XPCI_IR_MASK.
00703 *
00704 * @return None
00705 *
00706 * @note None
00707
*****
00708 #define XPCI_mIntrPciWriteISR(BaseAddress, Mask) \
00709     XIIF_V123B_WRITE_IISR((BaseAddress), (Mask))
00710
00711
00712
/*****
00713 /**
00714 * Low level register read function.
00715 *
00716 * C-style signature:
00717 *     Xuint32 XPCI_mReadReg(Xuint32 BaseAddress, Xuint32 RegOffset)
00718 *
00719 * @param BaseAddress is the base address of the PCI component.
00720 * @param RegOffset is the register offset.
00721 *
00722 * @return Value of the register.
00723 *
00724 * @note None
00725
*****
00726 #define XPCI_mReadReg(BaseAddress, RegOffset) \
00727     XIo_In32((BaseAddress) + (RegOffset))
00728
00729
00730
/*****
00731 /**
00732 * Low level register write function.
00733 *
00734 * C-style signature:
00735 *     void XPCI_mWriteReg(Xuint32 BaseAddress, Xuint32 RegOffset, Xuint32 Data)
00736 *
00737 * @param BaseAddress is the base address of the PCI component.
00738 * @param RegOffset is the register offset.
00739 * @param Data is the data to write.
00740 *
00741 * @return None
00742 *
00743 * @note None
00744
*****
00745 #define XPCI_mWriteReg(BaseAddress, RegOffset, Data) \
00746     XIo_Out32((BaseAddress) + (RegOffset), (Data))

```

```
00747
00748
00749
/*****
00750 /**
00751 * Low level PCI configuration read
00752 *
00753 * C-style signature:
00754 *     void XPCI_mConfigIn(Xuint32 BaseAddress, Xuint32 ConfigAddress,
00755 *                         Xuint32 ConfigData)
00756 *
00757 * @param BaseAddress is the base address of the PCI component.
00758 * @param ConfigAddress is the PCI configuration space address in a packed
00759 *     format.
00760 * @param ConfigData is the data read from the ConfigAddress
00761 *
00762 * @return Data from configuration address
00763 *
00764 * @note None
00765
*****/
00766 #define XPCI_mConfigIn(BaseAddress, ConfigAddress, ConfigData) \
00767     XIo_OutPci((BaseAddress) + XPCI_CAR_OFFSET, (ConfigAddress)); \
00768     (ConfigData) = XIo_InPci((BaseAddress) + XPCI_CDR_OFFSET)
00769
00770
00771
/*****
00772 /**
00773 * Low level PCI configuration write
00774 *
00775 * C-style signature:
00776 *     void XPCI_mConfigOut(Xuint32 BaseAddress, Xuint32 ConfigAddress,
00777 *                          Xuint32 ConfigData)
00778 *
00779 * @param BaseAddress is the base address of the PCI component.
00780 * @param ConfigAddress is the PCI configuration space address in a packed
00781 *     format.
00782 * @param ConfigData is the data to write at the ConfigAddress
00783 *
00784 * @return None
00785 *
00786 * @note None
00787
*****/
00788 #define XPCI_mConfigOut(BaseAddress, ConfigAddress, ConfigData) \
00789     XIo_OutPci((BaseAddress) + XPCI_CAR_OFFSET, (ConfigAddress)); \
00790     XIo_OutPci((BaseAddress) + XPCI_CDR_OFFSET, (ConfigData))
00791
00792
00793
/*****
```



```

00794 /**
00795 * Generate a PCI interrupt acknowledge bus cycle with the given vector.
00796 *
00797 * C-style signature:
00798 *     void XPCI_mAckSend(Xuint32 BaseAddress, Xuint32 Vector)
00799 *
00800 * @param BaseAddress is the base address of the PCI component.
00801 * @param Mask is the interrupt vector to place on the PCI bus.
00802 *
00803 * @return None
00804 *
00805 * @note None
00806
00807 *****/
00807 #define XPCI_mAckSend(BaseAddress, Vector) \
00808     XPCI_mWriteReg((BaseAddress), XPCI_IAR_OFFSET, (Vector))
00809
00810
00811
00812 *****/
00812 /**
00813 * Read the contents of the PCI interrupt acknowledge vector register.
00814 *
00815 * C-style signature:
00816 *     Xuint32 XPCI_mAckRead(Xuint32 BaseAddress)
00817 *
00818 * @param BaseAddress is the base address of the PCI component.
00819 *
00820 * @return System dependent interrupt vector.
00821 *
00822 * @note None
00823
00824 *****/
00824 #define XPCI_mAckRead(BaseAddress) \
00825     XPCI_mReadReg((BaseAddress), XPCI_IAR_OFFSET)
00826
00827
00828
00829 *****/
00829 /**
00830 * Broadcasts a message to all listening PCI targets.
00831 *
00832 * C-style signature:
00833 *     Xuint32 XPCI_mSpecialCycle(Xuint32 BaseAddress, Xuint32 Data)
00834 *
00835 * @param BaseAddress is the base address of the PCI component.
00836 * @param Data is the data to broadcast.
00837 *
00838 * @return None
00839 *
00840 * @note None
00841

```

```

00842 #define XPCI_mSpecialCycle(BaseAddress, Data) \
00843     XPCI_mWriteReg((BaseAddress), XPCI_SC_DATA_OFFSET, (Data))
00844
00845
00846
/*****
00847 /**
00848 * Convert local bus address to a PCI address
00849 *
00850 * C-style signature:
00851 *     Xuint32 XPCI_mLocal2Pci(Xuint32 LocalAddress, Xuint32 TranslationOffset)
00852 *
00853 * @param LocalAddress is the local address to find the equivalent PCI address
00854 *     for.
00855 * @param TO is the translation offset to apply
00856 *
00857 * @return Address in PCI space
00858 *
00859 * @note IPIFBAR_n, IPIFHIGHADDR_n, and IPIFBAR2PCI_n, defined in xparameters.h,
00860 * are defined for each BAR. To make a proper conversion, LocalAddress must
00861 * fall within range of a IPIFBAR_n and IPIFHIGHADDR_n pair and TO specified
00862 * must be the matching IPIFBAR2PCI_n. Example: pciAddr = XPCI_mLocal2Pci(
00863 * XPAR_PCI_IPIFBAR_0, XPAR_PCI_IPIFBAR2PCI_0) finds the PCI equivalent address
00864 * for the local address named by XPAR_PCI_IPIFBAR_0.
00865
*****/
00866 #define XPCI_mLocal2Pci(LocalAddr, TO) ((Xuint32)(LocalAddr) + (Xuint32)(TO))
00867
00868
00869
/*****
00870 /**
00871 * Convert PCI address to a local bus address
00872 *
00873 * C-style signature:
00874 *     Xuint32 XPCI_mPci2Local(Xuint32 PciAddress, Xuint32 TranslationOffset)
00875 *
00876 * @param PciAddress is the PCI address to find the equivalent local address
00877 *     for.
00878 * @param TO is the translation offset to apply
00879 *
00880 * @return Address in local space
00881 *
00882 * @note PCIBAR_n, PCIBAR_LEN_n, and PCIBAR2IPIF_n, defined in xparameters.h,
00883 * are defined for each BAR. To make a proper conversion, PciAddress must
00884 * fall within range of a PCIBAR_n and PCIBAR_LEN_n pair and TO specified
00885 * must be the matching PCIBAR2IPIF_n. Example: localAddr = XPCI_mPci2Local(
00886 * XPAR_PCI_PCIBAR_0, XPAR_PCI_PCIBAR2IPIF_0) finds the local address that
00887 * corresponds to XPAR_PCI_PCIBAR_0 on the PCI bus. Note that PCIBAR_LEN is
00888 * expressed as a power of 2.
00889

```

```
*****/  
00890 #define XPci_mPci2Local(PciAddr, TO) ((Xuint32)(PciAddr) + (Xuint32)(TO))  
00891  
00892 #endif /* XPCI_L_H */
```

Generated on 29 May 2003 for Xilinx Device Drivers

pci/v1_00_a/src/xpci_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined and more PCI documentation is in [xpci.h](#).

PCI configuration read/write macro functions can be changed so that data is swapped before being written to the configuration address/data registers. As delivered in this file, these macros do not swap. Change the definitions of [XIo_InPci\(\)](#) and [XIo_OutPci\(\)](#) in this file to suit system needs.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	rmm	05/19/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

[Go to the source code of this file.](#)

Registers

Register offsets for this device. Note that the following IPIF registers are implemented. Macros are defined to specifically access these registers without knowing which version of IPIF being used.

```
#define XPCI_PREOVRD_OFFSET
```

```
#define XPCI_IAR_OFFSET
#define XPCI_SC_DATA_OFFSET
#define XPCI_CAR_OFFSET
#define XPCI_CDR_OFFSET
#define XPCI_BUSNO_OFFSET
#define XPCI_STATCMD_OFFSET
#define XPCI_STATV3_OFFSET
#define XPCI_INHIBIT_OFFSET
#define XPCI_LMADDR_OFFSET
#define XPCI_LMA_R_OFFSET
#define XPCI_LMA_W_OFFSET
#define XPCI_SERR_R_OFFSET
#define XPCI_SERR_W_OFFSET
#define XPCI_PIADDR_OFFSET
#define XPCI_PIA_R_OFFSET
#define XPCI_PIA_W_OFFSET
```

Interrupt Status and Enable Register bitmaps and masks

Bit definitions for the interrupt status register and interrupt enable registers.

```
#define XPCI_IR_MASK
#define XPCI_IR_LM_SERR_R
#define XPCI_IR_LM_PERR_R
#define XPCI_IR_LM_TA_R
#define XPCI_IR_LM_SERR_W
#define XPCI_IR_LM_PERR_W
#define XPCI_IR_LM_TA_W
#define XPCI_IR_LM_MA_W
#define XPCI_IR_LM_BR_W
#define XPCI_IR_LM_BRD_W
#define XPCI_IR_LM_BRT_W
#define XPCI_IR_LM_BRANGE_W
#define XPCI_IR_PI_SERR_R
#define XPCI_IR_PI_SERR_W
```

Inhibit transfers on errors register bitmaps and masks.

These bits control whether subsequent PCI transactions are allowed after an error occurs. Bits set to 1 inhibit further transactions, while bits set to 0 allow further transactions after an error.

```
#define XPCI_INHIBIT_MASK
#define XPCI_INHIBIT_LOCAL_BUS_R
#define XPCI_INHIBIT_LOCAL_BUS_W
#define XPCI_INHIBIT_PCI_R
#define XPCI_INHIBIT_PCI_W
```

PCI configuration status & command register bitmaps and masks.

Bit definitions for the PCI configuration status & command register. The definition of this register is standard for PCI devices.

```
#define XPCI_STATCMD_IO_EN
#define XPCI_STATCMD_MEM_EN
#define XPCI_STATCMD_BUSM_EN
#define XPCI_STATCMD_SPECIALCYC
#define XPCI_STATCMD_MEMWR_INV_EN
#define XPCI_STATCMD_VGA_SNOOP_EN
#define XPCI_STATCMD_PARITY
#define XPCI_STATCMD_STEPPING
#define XPCI_STATCMD_SERR_EN
#define XPCI_STATCMD_BACK_EN
#define XPCI_STATCMD_INT_DISABLE
#define XPCI_STATCMD_INT_STATUS
#define XPCI_STATCMD_66MHZ_CAP
#define XPCI_STATCMD_MPERR
#define XPCI_STATCMD_DEVSEL_MSK
#define XPCI_STATCMD_DEVSEL_FAST
#define XPCI_STATCMD_DEVSEL_MED
```

```
#define XPCI_STATCMD_TGTABRT_SIG
#define XPCI_STATCMD_TGTABRT_RCV
#define XPCI_STATCMD_MSTABRT_RCV
#define XPCI_STATCMD_SERR_SIG
#define XPCI_STATCMD_PERR_DET
#define XPCI_STATCMD_ERR_MASK
```

V3 core transaction status register bitmaps and masks.

Bit definitions for the V3 core transaction status register. This register consists of status information on V3 core transactions.

```
#define XPCI_STATV3_MASK
#define XPCI_STATV3_DATA_XFER
#define XPCI_STATV3_TRANS_END
#define XPCI_STATV3_NORM_TERM
#define XPCI_STATV3_TGT_TERM
#define XPCI_STATV3_DISC_WODATA
#define XPCI_STATV3_DISC_WDATA
#define XPCI_STATV3_TGT_ABRT
#define XPCI_STATV3_MASTER_ABRT
#define XPCI_STATV3_PCI_RETRY_R
#define XPCI_STATV3_PCI_RETRY_W
#define XPCI_STATV3_WRITE_BUSY
```

Bus number and subordinate bus number register bitmaps and masks

```
#define XPCI_BUSNO_BUS_MASK
#define XPCI_BUSNO_SUBBUS_MASK
```

Local bus master address register bitmaps and masks

Bit definitions for the local bus master address definition register. This register defines the meaning of the address stored in the local bus master read (XPCI_LMA_R_OFFSET) and master write error (XPCI_LMA_W_OFFSET) registers.

```
#define XPCI_LMADDR_MASK
#define XPCI_LMADDR_SERR_R
#define XPCI_LMADDR_PERR_R
#define XPCI_LMADDR_TA_R
#define XPCI_LMADDR_SERR_W
#define XPCI_LMADDR_PERR_W
#define XPCI_LMADDR_TA_W
#define XPCI_LMADDR_MA_W
#define XPCI_LMADDR_BR_W
#define XPCI_LMADDR_BRD_W
#define XPCI_LMADDR_BRT_W
#define XPCI_LMADDR_BRANGE_W
```

PCI error address definition bitmaps and masks

Bit definitions for the PCI address definition register. This register defines the meaning of the address stored in the PCI read error address (XPCI_PIA_R_OFFSET) and PCI write error address (XPCI_PIA_W_OFFSET) registers.

```
#define XPCI_PIADDR_MASK
#define XPCI_PIADDR_ERRACK_R
#define XPCI_PIADDR_ERRACK_W
#define XPCI_PIADDR_RETRY_W
#define XPCI_PIADDR_TIMEOUT_W
#define XPCI_PIADDR_RANGE_W
```


PCI configuration header offsets

Defines the offsets in the standard PCI configuration header

```
#define XPCI_HDR_VENDOR
#define XPCI_HDR_DEVICE
#define XPCI_HDR_COMMAND
#define XPCI_HDR_STATUS
#define XPCI_HDR_REVID
#define XPCI_HDR_CLASSCODE
#define XPCI_HDR_CACHE_LINE_SZ
#define XPCI_HDR_LATENCY
#define XPCI_HDR_TYPE
#define XPCI_HDR_BIST
#define XPCI_HDR_BAR0
#define XPCI_HDR_BAR1
#define XPCI_HDR_BAR2
#define XPCI_HDR_BAR3
#define XPCI_HDR_BAR4
#define XPCI_HDR_BAR5
#define XPCI_HDR_CARDBUS_PTR
#define XPCI_HDR_SUB_VENDOR
#define XPCI_HDR_SUB_DEVICE
#define XPCI_HDR_ROM_BASE
#define XPCI_HDR_CAP_PTR
#define XPCI_HDR_INT_LINE
#define XPCI_HDR_INT_PIN
#define XPCI_HDR_MIN_GNT
#define XPCI_HDR_MAX_LAT
```

PCI BAR register definitions

Defines the masks and bits for the PCI XPAR_HDR_BARn registers The bridge supports the first three BARs in the PCI configuration header

```
#define XPCI_HDR_NUM_BAR
```

```
#define XPCI_HDR_BAR_ADDR_MASK
#define XPCI_HDR_BAR_PREFETCH_YES
#define XPCI_HDR_BAR_PREFETCH_NO
#define XPCI_HDR_BAR_TYPE_MASK
#define XPCI_HDR_BAR_TYPE_BELOW_4GB
#define XPCI_HDR_BAR_TYPE_BELOW_1MB
#define XPCI_HDR_BAR_TYPE_ABOVE_4GB
#define XPCI_HDR_BAR_TYPE_RESERVED
#define XPCI_HDR_BAR_SPACE_IO
#define XPCI_HDR_BAR_SPACE_MEMORY
```

DMA type constants

Defines the types of DMA engines

```
#define XPCI_DMA_TYPE_NONE
#define XPCI_DMA_TYPE_SIMPLE
#define XPCI_DMA_TYPE_SG
```

Defines

```
#define XPci_mReset(BaseAddress)
#define XPci_mIntrGlobalEnable(BaseAddress)
#define XPci_mIntrGlobalDisable(BaseAddress)
#define XPci_mIntrEnable(BaseAddress, Mask)
#define XPci_mIntrDisable(BaseAddress, Mask)
#define XPci_mIntrClear(BaseAddress, Mask)
#define XPci_mIntrReadIER(BaseAddress)
#define XPci_mIntrReadISR(BaseAddress)
#define XPci_mIntrWriteISR(BaseAddress, Mask)
#define XPci_mIntrReadIPR(BaseAddress)
#define XPci_mIntrReadID(BaseAddress)
#define XPci_mIntrPciEnable(BaseAddress, Mask)
#define XPci_mIntrPciDisable(BaseAddress, Mask)
#define XPci_mIntrPciClear(BaseAddress, Mask)
#define XPci_mIntrPciReadIER(BaseAddress)
```

```
#define XPCI_mIntrPciReadISR(BaseAddress)
#define XPCI_mIntrPciWriteISR(BaseAddress, Mask)
#define XPCI_mReadReg(BaseAddress, RegOffset)
#define XPCI_mWriteReg(BaseAddress, RegOffset, Data)
#define XPCI_mConfigIn(BaseAddress, ConfigAddress, ConfigData)
#define XPCI_mConfigOut(BaseAddress, ConfigAddress, ConfigData)
#define XPCI_mAckSend(BaseAddress, Vector)
#define XPCI_mAckRead(BaseAddress)
#define XPCI_mSpecialCycle(BaseAddress, Data)
#define XPCI_mLocal2Pci(LocalAddr, TO)
#define XPCI_mPci2Local(PciAddr, TO)
```

Define Documentation

```
#define XPCI_BUSNO_BUS_MASK
```

Mask for bus number

```
#define XPCI_BUSNO_OFFSET
```

bus/subordinate bus numbers

```
#define XPCI_BUSNO_SUBBUS_MASK
```

Mask for subordinate bus no

```
#define XPCI_CAR_OFFSET
```

Config addr reg (port)

```
#define XPCI_CDR_OFFSET
```

Config command data

```
#define XPCI_DMA_TYPE_NONE
```

No DMA

```
#define XPCI_DMA_TYPE_SG
```

Scatter-gather DMA

#define XPCI_DMA_TYPE_SIMPLE

Simple DMA

#define XPCI_HDR_BAR0

Base address 0

#define XPCI_HDR_BAR1

Base address 1

#define XPCI_HDR_BAR2

Base address 2

#define XPCI_HDR_BAR3

Base address 3

#define XPCI_HDR_BAR4

Base address 4

#define XPCI_HDR_BAR5

Base address 5

#define XPCI_HDR_BAR_ADDR_MASK

Base address mask

#define XPCI_HDR_BAR_PREFETCH_NO

Range is not prefetchable

#define XPCI_HDR_BAR_PREFETCH_YES

Range is prefetchable

#define XPCI_HDR_BAR_SPACE_IO

IO space indicator

#define XPCI_HDR_BAR_SPACE_MEMORY

Memory space indicator

#define XPCI_HDR_BAR_TYPE_ABOVE_4GB

Locate anywhere above 4GB

#define XPCI_HDR_BAR_TYPE_BELOW_1MB

Reserved in PCI 2.2

#define XPCI_HDR_BAR_TYPE_BELOW_4GB

Locate anywhere below 4GB

#define XPCI_HDR_BAR_TYPE_MASK

Memory type mask

#define XPCI_HDR_BAR_TYPE_RESERVED

Reserved

#define XPCI_HDR_BIST

Built in self test

#define XPCI_HDR_CACHE_LINE_SZ

Cache line size

#define XPCI_HDR_CAP_PTR

Capabilities pointer

#define XPCI_HDR_CARDBUS_PTR

Cardbus CIS pointer

#define XPCI_HDR_CLASSCODE

Class code

#define XPCI_HDR_COMMAND

Command register

#define XPCI_HDR_DEVICE

Device ID

#define XPCI_HDR_INT_LINE

Interrupt line

#define XPCI_HDR_INT_PIN

Interrupt pin

#define XPCI_HDR_LATENCY

Latency timer

#define XPCI_HDR_MAX_LAT

Priority level request

#define XPCI_HDR_MIN_GNT

Timeslice request

#define XPCI_HDR_NUM_BAR

Number of BARs in the PCI header

#define XPCI_HDR_REVID

Revision ID

#define XPCI_HDR_ROM_BASE

Expansion ROM base address

#define XPCI_HDR_STATUS

Status register

#define XPCI_HDR_SUB_DEVICE

Subsystem ID

#define XPCI_HDR_SUB_VENDOR

Subsystem Vendor ID

#define XPCI_HDR_TYPE

Header type

#define XPCI_HDR_VENDOR

Vendor ID

#define XPCI_IAR_OFFSET

PCI interrupt ack

#define XPCI_INHIBIT_LOCAL_BUS_R

Local bus master reads

#define XPCI_INHIBIT_LOCAL_BUS_W

Local bus mater writes

#define XPCI_INHIBIT_MASK

Mask for all bits defined below

#define XPCI_INHIBIT_OFFSET

Inhibit transfers on errors

#define XPCI_INHIBIT_PCI_R

PCI initiator reads

#define XPCI_INHIBIT_PCI_W

PCI initiator writes

#define XPCI_IR_LM_BR_W

Local bus master burst write retry

#define XPCI_IR_LM_BRANGE_W

Local bus master burst write range

#define XPCI_IR_LM_BRD_W

Local bus master burst write retry disconnect

#define XPCI_IR_LM_BRT_W

Local bus master burst write retry timeout

#define XPCI_IR_LM_MA_W

Local bus master abort write

#define XPCI_IR_LM_PERR_R

Local bus master read PERR

#define XPCI_IR_LM_PERR_W

Local bus master write PERR

#define XPCI_IR_LM_SERR_R

Local bus master read SERR

#define XPCI_IR_LM_SERR_W

Local bus master write SERR

#define XPCI_IR_LM_TA_R

Local bus master read target abort

#define XPCI_IR_LM_TA_W

Local bus master write target abort

#define XPCI_IR_MASK

Mask of all bits

#define XPCI_IR_PI_SERR_R

PCI initiator read SERR

#define XPCI_IR_PI_SERR_W

PCI initiator write SERR

#define XPCI_LMA_R_OFFSET

Local bus master read error address

#define XPCI_LMA_W_OFFSET

Local bus master write error address

#define XPCI_LMADDR_BR_W

Master burst write retry

#define XPCI_LMADDR_BRANGE_W

Master burst write range

#define XPCI_LMADDR_BRD_W

Master burst write retry disconnect

#define XPCI_LMADDR_BRT_W

Master burst write retry timeout

#define XPCI_LMADDR_MA_W

Master abort write

```
#define XPCI_LMADDR_MASK
```

Mask of all bits

```
#define XPCI_LMADDR_OFFSET
```

Local bus master address definition

```
#define XPCI_LMADDR_PERR_R
```

Master read PERR

```
#define XPCI_LMADDR_PERR_W
```

Master write PERR

```
#define XPCI_LMADDR_SERR_R
```

Master read SERR

```
#define XPCI_LMADDR_SERR_W
```

Master write SERR

```
#define XPCI_LMADDR_TA_R
```

Master read target abort

```
#define XPCI_LMADDR_TA_W
```

Master write target abort

```
#define XPCI_mAckRead( BaseAddress )
```

Read the contents of the PCI interrupt acknowledge vector register.

C-style signature: Xuint32 **XPci_mAckRead**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

System dependent interrupt vector.

Note:

None

```
#define XPci_mAckSend( BaseAddress,  
                    Vector      )
```

Generate a PCI interrupt acknowledge bus cycle with the given vector.

C-style signature: void **XPci_mAckSend**(Xuint32 BaseAddress, Xuint32 Vector)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the interrupt vector to place on the PCI bus.

Returns:

None

Note:

None

```
#define XPci_mConfigIn( BaseAddress,  
                      ConfigAddress,  
                      ConfigData  )
```

Low level PCI configuration read

C-style signature: void XPCI_mConfigIn(Xuint32 BaseAddress, Xuint32 ConfigAddress, Xuint32 ConfigData)

Parameters:

BaseAddress is the base address of the PCI component.

ConfigAddress is the PCI configuration space address in a packed format.

ConfigData is the data read from the ConfigAddress

Returns:

Data from configuration address

Note:

None

```
#define XPCI_mConfigOut( BaseAddress,  
                        ConfigAddress,  
                        ConfigData    )
```

Low level PCI configuration write

C-style signature: void XPCI_mConfigOut(Xuint32 BaseAddress, Xuint32 ConfigAddress, Xuint32 ConfigData)

Parameters:

BaseAddress is the base address of the PCI component.

ConfigAddress is the PCI configuration space address in a packed format.

ConfigData is the data to write at the ConfigAddress

Returns:

None

Note:

None

```
#define XPCI_mIntrClear( BaseAddress,  
                        Mask          )
```

Clear pending interrupts in the device interrupt status register (DISR) This is a toggle on write register.

C-style signature: void **XPCI_mIntrClear**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to clear. Use a logical OR of constants in XPCI_IPIF_INT_MASK. Bits set to 1 are cleared, bits set to 0 are not affected.

Returns:

None

Note:

None

```
#define XPCI_mIntrDisable( BaseAddress,  
                          Mask          )
```

Disable interrupts in the device interrupt enable register (DIER)

C-style signature: void **XPCI_mIntrDisable**(Xuint32 BaseAddress, Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to disable. Use a logical OR of constants in XPCI_IPIF_INT_MASK. Bits set to 1 are disabled, bits set to 0 are not affected.

Returns:

None

Note:

None

```
#define XPCI_mIntrEnable( BaseAddress,  
                          Mask          )
```

Enable interrupts in the device interrupt enable register (DIER)

C-style signature: void **XPci_mIntrEnable**(Xuint32 BaseAddress, Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to enable. Use a logical OR of constants in XPCI_IPIF_INT_MASK. Bits set to 1 are enabled, bits set to 0 are not affected.

Returns:

None

Note:

None

```
#define XPci_mIntrGlobalDisable( BaseAddress )
```

Global interrupt disable. Disable all interrupts from this core. Any interrupts enabled by **XPci_mIntrEnable**() or **XPci_mIntrPciEnable**() are disabled, however their settings remain unchanged.

C-style signature: void **XPci_mIntrGlobalDisable**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

None

Note:

None

```
#define XPci_mIntrGlobalEnable( BaseAddress )
```

Global interrupt enable. Must be called before any interrupts enabled by **XPCI_mIntrEnable()** or **XPCI_mIntrPciEnable()** have any effect.

C-style signature: void **XPCI_mIntrGlobalEnable**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

None

Note:

None

```
#define XPCI_mIntrPciClear( BaseAddress,  
                           Mask          )
```

Clear PCI specific interrupts in the interrupt status register (IISR). This is a toggle on write register.

C-style signature: void **XPCI_mIntrPciClear**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to clear. Use a logical OR of constants in XPCI_IR_MASK. Bits set to 1 are cleared, bits set to 0 are not affected.

Returns:

None

Note:

None

```
#define XPCI_mIntrPciDisable( BaseAddress,  
                              Mask          )
```

Disable PCI specific interrupt sources in the PCI interrupt enable register (IIER)

C-style signature: void **XPci_mIntrPciDisable**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to disable. Bits set to 1 are disabled, bits set to 0 are not affected. The mask is made up by OR'ing bits from XPCI_IR_MASK.

Returns:

None

Note:

None

```
#define XPci_mIntrPciEnable( BaseAddress,  
                           Mask          )
```

Enable PCI specific interrupt sources in the PCI interrupt enable register (IIER)

C-style signature: void **XPci_mIntrPciEnable**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the group of interrupts to enable. Bits set to 1 are enabled, bits set to 0 are not affected. The mask is made up by OR'ing bits from XPCI_IR_MASK.

Returns:

None

Note:

None

```
#define XPci_mIntrPciReadIER( BaseAddress )
```


Read the contents of the PCI specific interrupt enable register (IIER)

C-style signature: Xuint32 **XPci_mIntrPciReadIER**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Contents of the pending interrupt register. The mask is made up of bits defined in XPCI_IR_MASK.

Note:

None

```
#define XPci_mIntrPciReadISR( BaseAddress )
```

Read the contents of the PCI specific interrupt status register (IISR)

C-style signature: Xuint32 **XPci_mIntrPciReadISR**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Contents of the pending interrupt register. The mask is made up of bits defined in XPCI_IR_MASK.

Note:

None

```
#define XPci_mIntrPciWriteISR( BaseAddress,  
                               Mask      )
```

Write to the PCI interrupt status register (IISR)

C-style signature: void **XPCI_mIntrPciWriteISR**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the value to write to the register and is assumed to be bits or'd together from the XPCI_IR_MASK.

Returns:

None

Note:

None

```
#define XPCI_mIntrReadID( BaseAddress )
```

Read the contents of the device interrupt ID register (DIIR).

C-style signature: Xuint32 **XPCI_mIntrReadID**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Value of the register.

Note:

None

```
#define XPCI_mIntrReadIER( BaseAddress )
```

Read the contents of the device interrupt enable register (DIER)

C-style signature: Xuint32 **XPci_mIntrReadIER**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Value of the register.

Note:

None

```
#define XPci_mIntrReadIPR( BaseAddress )
```

Read the contents of the device interrupt pending register (DIPR).

C-style signature: Xuint32 **XPci_mIntrReadIPR**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Value of the register.

Note:

None

```
#define XPci_mIntrReadISR( BaseAddress )
```

Read the contents of the device interrupt status register (DISR)

C-style signature: Xuint32 **XPci_mIntrReadISR**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

Value of the register.

Note:

None

```
#define XPci_mIntrWriteISR( BaseAddress,  
                           Mask          )
```

Write to the device interrupt status register (DISR)

C-style signature: void **XPci_mIntrWriteISR**(Xuint32 BaseAddress, Xuint32 Mask)

Parameters:

BaseAddress is the base address of the PCI component.

Mask is the value to write to the register and is assumed to be bits or'd together from the XPCI_IPIF_INT_MASK.

Returns:

None

Note:

None

```
#define XPci_mLocal2Pci( LocalAddr,  
                        TO          )
```

Convert local bus address to a PCI address

C-style signature: Xuint32 **XPci_mLocal2Pci**(Xuint32 LocalAddress, Xuint32 TranslationOffset)

Parameters:

LocalAddress is the local address to find the equivalent PCI address for.

TO is the translation offset to apply

Returns:

Address in PCI space

Note:

IPIFBAR_n, IPIFHIGHADDR_n, and IPIFBAR2PCI_n, defined in [xparameters.h](#), are defined for each BAR. To make a proper conversion, LocalAddress must fall within range of a IPIFBAR_n and IPIFHIGHADDR_n pair and TO specified must be the matching IPIFBAR2PCI_n. Example: pciAddr = XPci_mLocal2Pci(XPAR_PCI_IPIFBAR_0, XPAR_PCI_IPIFBAR2PCI_0) finds the PCI equivalent address for the local address named by XPAR_PCI_IPIFBAR_0.

```
#define XPci_mPci2Local( PciAddr,  
                        TO      )
```

Convert PCI address to a local bus address

C-style signature: Xuint32 **XPci_mPci2Local**(Xuint32 PciAddress, Xuint32 TranslationOffset)

Parameters:

PciAddress is the PCI address to find the equivalent local address for.

TO is the translation offset to apply

Returns:

Address in local space

Note:

PCIBAR_n, PCIBAR_LEN_n, and PCIBAR2IPIF_n, defined in [xparameters.h](#), are defined for each BAR. To make a proper conversion, PciAddress must fall within range of a PCIBAR_n and PCIBAR_LEN_n pair and TO specified must be the matching PCIBAR2IPIF_n. Example: localAddr = XPci_mPci2Local(XPAR_PCI_PCIBAR_0, XPAR_PCI_PCIBAR2IPIF_0) finds the local address that corresponds to

XPAR_PCI_PCIBAR_0 on the PCI bus. Note that PCIBAR_LEN is expressed as a power of 2.

```
#define XPCI_mReadReg( BaseAddress,  
                    RegOffset  )
```

Low level register read function.

C-style signature: Xuint32 **XPCI_mReadReg**(Xuint32 BaseAddress, Xuint32 RegOffset)

Parameters:

BaseAddress is the base address of the PCI component.
RegOffset is the register offset.

Returns:

Value of the register.

Note:

None

```
#define XPCI_mReset( BaseAddress )
```

IPIF Low level PCI reset function. Reset the V3 core.

C-style signature: void **XPCI_mReset**(Xuint32 BaseAddress)

Parameters:

BaseAddress is the base address of the PCI component.

Returns:

None

Note:

The IPIF RESETR register is located at (base + 0x80) instead of (base + 0x40) where the IPIF component driver expects it. This macro adjusts for this difference.

```
#define XPCI_mSpecialCycle( BaseAddress,  
                           Data          )
```

Broadcasts a message to all listening PCI targets.

C-style signature: Xuint32 **XPCI_mSpecialCycle**(Xuint32 BaseAddress, Xuint32 Data)

Parameters:

BaseAddress is the base address of the PCI component.

Data is the data to broadcast.

Returns:

None

Note:

None

```
#define XPCI_mWriteReg( BaseAddress,  
                      RegOffset,  
                      Data          )
```

Low level register write function.

C-style signature: void **XPCI_mWriteReg**(Xuint32 BaseAddress, Xuint32 RegOffset, Xuint32 Data)

Parameters:

BaseAddress is the base address of the PCI component.

RegOffset is the register offset.

Data is the data to write.

Returns:

None

Note:

None

```
#define XPCI_PIA_R_OFFSET
```

PCI read error address

#define XPCI_PIA_W_OFFSET

PCI write error address

#define XPCI_PIADDR_ERRACK_R

PCI initiator read ErrAck

#define XPCI_PIADDR_ERRACK_W

PCI initiator write ErrAck

#define XPCI_PIADDR_MASK

Mask of all bits

#define XPCI_PIADDR_OFFSET

PCI address definition

#define XPCI_PIADDR_RANGE_W

PCI initiator write range

#define XPCI_PIADDR_RETRY_W

PCI initiator write retries

#define XPCI_PIADDR_TIMEOUT_W

PCI initiator write timeout

#define XPCI_PREOVRD_OFFSET

Prefetch override

#define XPCI_SC_DATA_OFFSET

Special cycle data

#define XPCI_SERR_R_OFFSET

PCI initiator read SERR address

#define XPCI_SERR_W_OFFSET

PCI initiator write SERR address

#define XPCI_STATCMD_66MHZ_CAP

66MHz capable

#define XPCI_STATCMD_BACK_EN

Fast back-to-back enable

#define XPCI_STATCMD_BUSM_EN

Bus master enable

#define XPCI_STATCMD_DEVSEL_FAST

Device select timing fast

#define XPCI_STATCMD_DEVSEL_MED

Device select timing medium

#define XPCI_STATCMD_DEVSEL_MSK

Device select timing mask

#define XPCI_STATCMD_ERR_MASK

Error bits or'd together

#define XPCI_STATCMD_INT_DISABLE

Interrupt disable (PCI v2.3)

#define XPCI_STATCMD_INT_STATUS

Interrupt status (PCI v2.3)

#define XPCI_STATCMD_IO_EN

I/O access enable

#define XPCI_STATCMD_MEM_EN

Memory access enable

#define XPCI_STATCMD_MEMWR_INV_EN

Memory write & invalidate

#define XPCI_STATCMD_MPERR

Master data PERR detected

#define XPCI_STATCMD_MSTABRT_RCV

Received master abort

#define XPCI_STATCMD_OFFSET

PCI config status/command

#define XPCI_STATCMD_PARITY

Report parity errors

#define XPCI_STATCMD_PERR_DET

Detected PERR

#define XPCI_STATCMD_SERR_EN

SERR report enable

#define XPCI_STATCMD_SERR_SIG

Signaled SERR

#define XPCI_STATCMD_SPECIALCYC

Special cycles

#define XPCI_STATCMD_STEPPING

Stepping control

#define XPCI_STATCMD_TGTABRT_RCV

Received target abort

#define XPCI_STATCMD_TGTABRT_SIG

Signaled target abort

#define XPCI_STATCMD_VGA_SNOOP_EN

VGA palette snoop enable

#define XPCI_STATV3_DATA_XFER

Data transfer. Read only

#define XPCI_STATV3_DISC_WDATA

Disconnect with data. Read only

#define XPCI_STATV3_DISC_WODATA

Disconnect without data. Read only

#define XPCI_STATV3_MASK

Mask of all bits

#define XPCI_STATV3_MASTER_ABRT

Master abort. Read only

#define XPCI_STATV3_NORM_TERM

Normal termination. Read only

#define XPCI_STATV3_OFFSET

V3 core transaction status

#define XPCI_STATV3_PCI_RETRY_R

PCI retry on read. Read/write

#define XPCI_STATV3_PCI_RETRY_W

PCI retry on write. Read/write

#define XPCI_STATV3_TGT_ABRT

Target abort. Read only

#define XPCI_STATV3_TGT_TERM

Target termination. Read only

#define XPCI_STATV3_TRANS_END

Transaction end. Read only

#define XPCI_STATV3_WRITE_BUSY

Write busy. Read only

Generated on 29 May 2003 for Xilinx Device Drivers

XPciError Struct Reference

```
#include <xpci.h>
```

Detailed Description

XPciError is used to retrieve a snapshot of the bridge's error state. Most of the attributes of this structure are copies of various bridge registers. See [XPci_ErrorGet\(\)](#) and [XPci_ErrorClear\(\)](#).

Data Fields

Xboolean IsError

Xuint32 LocalBusReason

Xuint32 PciReason

Xuint32 PciSerrReason

Xuint32 LocalBusReadAddr

Xuint32 LocalBusWriteAddr

Xuint32 PciReadAddr

Xuint32 PciWriteAddr

Xuint32 PciSerrReadAddr

Xuint32 PciSerrWriteAddr

Field Documentation

Xboolean XPciError::IsError

Global error indicator

Xuint32 XPCIError::LocalBusReadAddr

Local bus master read error address

Xuint32 XPCIError::LocalBusReason

Local bus master address definition

Xuint32 XPCIError::LocalBusWriteAddr

Local bus master write error address

Xuint32 XPCIError::PciReadAddr

PCI read error address

Xuint32 XPCIError::PciReason

PCI address definition

Xuint32 XPCIError::PciSerrReadAddr

PCI initiator read SERR address

Xuint32 XPCIError::PciSerrReason

PCI System error definition

Xuint32 XPCIError::PciSerrWriteAddr

PCI initiator write SERR address

Xuint32 XPCIError::PciWriteAddr

PCI write error address

The documentation for this struct was generated from the following file:

- [pci/v1_00_a/src/xpci.h](#)

pci/v1_00_a/src/xpci_config.c File Reference

Detailed Description

Implements advanced PCI configuration functions for the **XPCI** component. See [xpci.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	rmm	03/25/02	Original code

```
#include "xpci.h"
```

Functions

XStatus XPCI_ConfigIn8 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint8** *Data)

XStatus XPCI_ConfigIn16 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint16** *Data)

XStatus XPCI_ConfigIn32 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint32** *Data)

XStatus XPCI_ConfigOut8 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint8** Data)

XStatus XPCI_ConfigOut16 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint16** Data)

XStatus XPCI_ConfigOut32 (**XPCI** *InstancePtr, unsigned Bus, unsigned Device, unsigned Func, unsigned Offset, **Xuint32** Data)

Function Documentation

```
XStatus XPCI_ConfigIn16( XPCI * InstancePtr,  
                        unsigned Bus,  
                        unsigned Device,  
                        unsigned Func,  
                        unsigned Offset,  
                        Xuint16 * Data  
                        )
```

Perform a 16 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Data is the data read from the target.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus XPCI_ConfigIn32( XPCI * InstancePtr,  
                        unsigned Bus,  
                        unsigned Device,  
                        unsigned Func,  
                        unsigned Offset,  
                        Xuint32 * Data  
                        )
```

Perform a 32 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Data is the data read from the target.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus XPCI_ConfigIn8( XPCI * InstancePtr,  
                        unsigned Bus,  
                        unsigned Device,  
                        unsigned Func,  
                        unsigned Offset,  
                        Xuint8 * Data  
                        )
```

Perform a 8 bit read transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.
- Data* is the data read from the target.

Returns:

- XST_SUCCESS Operation was successful.
- XST_PCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus XPci_ConfigOut16( XPci * InstancePtr,  
                          unsigned Bus,  
                          unsigned Device,  
                          unsigned Func,  
                          unsigned Offset,  
                          Xuint16 Data  
                          )
```

Perform a 16 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

- InstancePtr* is the PCI component to operate on.
- Bus* is the target PCI Bus #.
- Device* is the target device number.
- Func* is the target device's function number.
- Offset* is the target device's configuration space I/O offset to address.

Returns:

- XST_SUCCESS Operation was successful.

- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus Xpci_ConfigOut32( Xpci * InstancePtr,  
                          unsigned Bus,  
                          unsigned Device,  
                          unsigned Func,  
                          unsigned Offset,  
                          Xuint32 Data  
                          )
```

Perform a 32 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

```
XStatus XPCI_ConfigOut8( XPCI * InstancePtr,  
                        unsigned Bus,  
                        unsigned Device,  
                        unsigned Func,  
                        unsigned Offset,  
                        Xuint8 Data  
                        )
```

Perform a 8 bit write transaction in PCI configuration space. Together, the Bus, Device, Func, & Offset form the address of the PCI target to access.

Parameters:

InstancePtr is the PCI component to operate on.

Bus is the target PCI Bus #.

Device is the target device number.

Func is the target device's function number.

Offset is the target device's configuration space I/O offset to address.

Returns:

- XST_SUCCESS Operation was successful.
- XPCI_INVALID_ADDRESS One of Bus, Device, Func, or Offset form an invalid address.

Note:

None

pci/v1_00_a/src/xpci_v3.c File Reference

Detailed Description

Implements V3 core processing functions for the **XPci** component. See [xpci.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	rmm	03/25/02	Original code

```
#include "xpci.h"
```

Functions

Xuint32 [XPci_V3StatusCommandGet](#) (**XPci** *InstancePtr)

Xuint32 [XPci_V3TransactionStatusGet](#) (**XPci** *InstancePtr)

void [XPci_V3TransactionStatusClear](#) (**XPci** *InstancePtr, **Xuint32** Data)

Function Documentation

Xuint32 [XPci_V3StatusCommandGet](#)(**XPci** * *InstancePtr*)

Read the contents of the V3 bridge's status & command register. This same register can be retrieved by a PCI configuration access. The register can be written only with a PCI configuration access.

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Contents of the V3 bridge's status and command register

Note:

None

```
void XPCI_V3TransactionStatusClear( XPCI * InstancePtr,  
                                     Xuint32 Data  
                                     )
```

Clear status bits in the V3 bridge's transaction status register. The contents of this register can be decoded using XPCI_STATV3_* constants defined in [xpci_l.h](#).

Parameters:

InstancePtr is the PCI component to operate on.

Data is the contents to write to the register. Or XPCI_STATV3_* constants for those bits to be cleared. Bits in the register that are read-only are not affected.

Note:

None

```
Xuint32 XPCI_V3TransactionStatusGet( XPCI * InstancePtr)
```

Read the contents of the V3 bridge's transaction status register. The contents of this register can be decoded using XPCI_STATV3_* constants defined in [xpci_l.h](#).

Parameters:

InstancePtr is the PCI component to operate on.

Returns:

Contents of the V3 bridge's transaction status register.

Note:

None

pci/v1_00_a/src/xpci_selftest.c File Reference

Detailed Description

Implements self test for the [XPci](#) component. See [xpci.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	03/25/02	Original code

```
#include "xpci.h"
```

Functions

[XStatus](#) [XPci_SelfTest](#) ([XPci](#) *InstancePtr)

Function Documentation

[XStatus](#) [XPci_SelfTest](#)([XPci](#) * *InstancePtr*)

Run a self-test on the driver/device. This includes the following tests:

- Configuration read of the bridge device and vendor ID.

Parameters:

InstancePtr is a pointer to the **XPCI** instance to be worked on. This parameter must have been previously initialized with **XPCI_Initialize()**.

Returns:

- XST_SUCCESS If test passed
- XST_FAILURE If test failed

Note:

None

plbarb/v1_01_a/src/xplbarb.h File Reference

Detailed Description

This component contains the implementation of the **XPlbArb** component. It is the driver for the PLB (Processor Local Bus) Arbiter. The arbiter performs bus arbitration on the PLB transactions.

This driver allows the user to access the PLB Arbiter registers to support the handling of bus errors and other access errors and determine an appropriate solution if possible.

The Arbiter Hardware generates an interrupt in error conditions which is not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

Hardware Features

The Xilinx PLB Arbiter is a soft IP core designed for Xilinx FPGAs and contains the following features:

- PLB address and data steering support for up to eight masters. Number of PLB masters is configurable via a design parameter
- 64-bit and/or 32-bit support for masters and slaves
- PLB address pipelining
- PLB arbitration support for up to eight masters. Number of PLB masters is configurable via a design parameter
- Three cycle arbitration
- Four levels of dynamic master request priority
- PLB watchdog timer
- PLB architecture compliant

Device Configuration

The device can be configured in various ways during the FPGA implementation process. The configuration data for each device is contained in **xplbarb_g.c**. A table is defined where each entry contains configuration information for a device. This information includes such things as the base address of the DCR mapped device, and the number of masters on the bus.

Note:

This driver is not thread-safe. Thread safety must be guaranteed by the layer above this driver if there is a need to access the device from multiple threads.

The Arbiter registers reside on the DCR address bus.

Any and all outstanding errors are cleared in the initialization function.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	12/7/01	First release
1.01a	rpm	05/13/02	Updated to match hw version, publicized LookupConfig

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XPlbArb  
struct XPlbArb_Config
```

Functions

```
XStatus XPlbArb_Initialize (XPlbArb *InstancePtr, Xuint16 DeviceId)  
void XPlbArb_Reset (XPlbArb *InstancePtr)  
XPlbArb_Config * XPlbArb_LookupConfig (Xuint16 DeviceId)  
Xboolean XPlbArb_IsError (XPlbArb *InstancePtr)  
void XPlbArb_ClearErrors (XPlbArb *InstancePtr, Xuint8 Master)  
Xuint32 XPlbArb_GetErrorStatus (XPlbArb *InstancePtr, Xuint8 Master)  
Xuint32 XPlbArb_GetErrorAddress (XPlbArb *InstancePtr)  
Xuint8 XPlbArb_GetNumMasters (XPlbArb *InstancePtr)  
void XPlbArb_EnableInterrupt (XPlbArb *InstancePtr)  
void XPlbArb_DisableInterrupt (XPlbArb *InstancePtr)  
XStatus XPlbArb_SelfTest (XPlbArb *InstancePtr, Xuint32 TestAddress)
```

Function Documentation

```
void XPlbArb_ClearErrors( XPlbArb * InstancePtr,  
                        Xuint8 Master  
                        )
```

Clears the Errors for the specified master

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters - 1 on the bus

Returns:

None.

Note:

None.

```
void XPlbArb_DisableInterrupt( XPlbArb * InstancePtr)
```

Disables the interrupt output from the arbiter

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None

Note:

The Arbiter hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt source with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void XPlbArb_EnableInterrupt( XPlbArb * InstancePtr)
```

Enables the interrupt output from the arbiter

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None.

Note:

The Arbiter hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt source with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

Xuint32 XPlbArb_GetErrorAddress(**XPlbArb** * *InstancePtr*)

Returns the PLB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XPlbArb_IsError()** is recommended to confirm that an error has occurred prior to calling this function to ensure that the data in the error address register is relevant.

Xuint32 XPlbArb_GetErrorStatus(**XPlbArb** * *InstancePtr*, **Xuint8** *Master*)

Returns the Error status for the specified master. These are bit masks.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

The current error status for the requested master on the PLB. The status is a bit-mask that can contain the following bit values:

XPA_DRIVING_BEAR_MASK	Indicates that an error has occurred and this master is driving the error address
XPA_ERROR_READ_MASK	Indicates the error was a read error (it is a write error otherwise).
XPA_ERROR_STATUS_LOCK_MASK	Indicates the error status and address are locked and cannot be overwritten.
XPA_PEAR_SIZE_MASK	Size of access that caused error
XPA_PEAR_TYPE_MASK	Type of access that caused error

Note:

None.

Xuint8 XPlbArb_GetNumMasters(**XPlbArb** * *InstancePtr*)

Returns the number of masters associated with the arbiter.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

The number of masters. This is a number from 1 to the maximum of 32.

Note:

The value returned from this call needs to be adjusted if it is to be used as the argument for other calls since the masters are numbered from 0 and this function returns values starting at 1.

```
XStatus XPlbArb_Initialize( XPlbArb * InstancePtr,  
                           Xuint16   DeviceId  
                           )
```

Initializes a specific **XPlbArb** instance. Looks up the configuration for the given device instance and initialize the instance structure.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPlbArb** component.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

```
Xboolean XPlbArb_IsError( XPlbArb * InstancePtr)
```

Returns XTRUE is there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

XPlbArb_Config* XPlbArb_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table XPlbArb_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or XNULL if no match is found.

Note:

None.

void XPlbArb_Reset(XPlbArb * InstancePtr)

Forces a software reset to occur in the arbiter. Disables interrupts in the process.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None.

Note:

Disables interrupts in the process.

XStatus XPlbArb_SelfTest(XPlbArb * InstancePtr, Xuint32 TestAddress)

Runs a self-test on the driver/device.

This tests reads the PACR to verify that the proper value is there.

XST_SUCCESS is returned if expected value is there, XST_PLBARB_FAIL_SELFTEST is returned otherwise.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

TestAddress is a location that could cause an error on read, not used - user definable for hw specific implementations.

Returns:

XST_SUCCESS if successful, or XST_PLBARB_FAIL_SELFTEST if the driver fails the self test.

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

XPIbArb Struct Reference

```
#include <xplbarb.h>
```

Detailed Description

The XPIbArb driver instance data. The user is required to allocate a variable of this type for every PLB arbiter device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `plbarb/v1_01_a/src/xplbarb.h`
-

plbarb/v1_01_a/src/xplbarb.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xplbarb.h,v 1.2 2002/06/26 22:58:36 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file plbarb/v1_01_a/src/xplbarb.h
00026 *
00027 * This component contains the implementation of the XPlbArb component. It is
the
00028 * driver for the PLB (Processor Local Bus) Arbiter. The arbiter performs bus
00029 * arbitration on the PLB transactions.
00030 *
00031 * This driver allows the user to access the PLB Arbiter registers to support
00032 * the handling of bus errors and other access errors and determine an
00033 * appropriate solution if possible.
00034 *
00035 * The Arbiter Hardware generates an interrupt in error conditions which is not
00036 * handled by the driver directly. It is the application's responsibility to
00037 * attach to the appropriate interrupt with a handler which then calls functions
00038 * provided by this driver to determine the cause of the error and take the
00039 * necessary actions to correct the situation.
00040 *
00041 * <b>Hardware Features</b>
```

```

00042 *
00043 * The Xilinx PLB Arbiter is a soft IP core designed for Xilinx FPGAs and
contains
00044 * the following features:
00045 * - PLB address and data steering support for up to eight masters. Number of
00046 *   PLB masters is configurable via a design parameter
00047 * - 64-bit and/or 32-bit support for masters and slaves
00048 * - PLB address pipelining
00049 * - PLB arbitration support for up to eight masters. Number of PLB masters is
00050 *   configurable via a design parameter
00051 * - Three cycle arbitration
00052 * - Four levels of dynamic master request priority
00053 * - PLB watchdog timer
00054 * - PLB architecture compliant
00055 *
00056 * <b>Device Configuration</b>
00057 *
00058 * The device can be configured in various ways during the FPGA implementation
00059 * process. The configuration data for each device is contained in xplbarb_g.c.
00060 * A table is defined where each entry contains configuration information for a
00061 * device. This information includes such things as the base address of the DCR
00062 * mapped device, and the number of masters on the bus.
00063 *
00064 * @note
00065 *
00066 * This driver is not thread-safe. Thread safety must be guaranteed by the layer
00067 * above this driver if there is a need to access the device from multiple
00068 * threads.
00069 * <br><br>
00070 * The Arbiter registers reside on the DCR address bus.
00071 * <br><br>
00072 * Any and all outstanding errors are cleared in the initialization function.
00073 *
00074 * <pre>
00075 * MODIFICATION HISTORY:
00076 *
00077 * Ver    Who    Date      Changes
00078 * -----
00079 * 1.00a  ecm    12/7/01   First release
00080 * 1.01a  rpm    05/13/02 Updated to match hw version, publicized LookupConfig
00081 * </pre>
00082 *
00083 * *****/
00084
00085 #ifndef XPLBARB_H /* prevent circular inclusions */
00086 #define XPLBARB_H /* by using protection macros */
00087
00088 /***** Include Files *****/
00089 #include "xbasic_types.h"
00090 #include "xstatus.h"
00091
00092 /***** Constant Definitions *****/

```

```

00093
00094 /***** Type Definitions *****/
00095
00096 /**
00097  * This typedef contains configuration information for the device. This
00098  * information would typically be extracted from Configuration ROM (CROM).
00099  */
00100 typedef struct
00101 {
00102     Xuint16 DeviceId;      /**< Unique ID of device */
00103     Xuint32 BaseAddress;   /**< Register base address */
00104     Xuint8  NumMasters;    /**< Number of masters on the bus */
00105 } XPlbArb_Config;
00106
00107 /**
00108  * The XPlbArb driver instance data. The user is required to allocate a
00109  * variable of this type for every PLB arbiter device in the system. A pointer
00110  * to a variable of this type is then passed to the driver API functions.
00111  */
00112 typedef struct
00113 {
00114     Xuint32 BaseAddress;    /* Base address of registers */
00115     Xuint32 IsReady;       /* Device is initialized and ready */
00116     Xuint8  NumMasters;    /* number of masters for this arbiter */
00117 } XPlbArb;
00118
00119
00120
00121 /***** Macros (Inline Functions) Definitions *****/
00122
00123
00124 /***** Function Prototypes *****/
00125
00126
00127 /*
00128  * Required functions in xplbarb.c
00129  */
00130
00131 /*
00132  * Initialization Functions
00133  */
00134 XStatus XPlbArb_Initialize(XPlbArb *InstancePtr, Xuint16 DeviceId);
00135 void XPlbArb_Reset(XPlbArb *InstancePtr);
00136 XPlbArb_Config *XPlbArb_LookupConfig(Xuint16 DeviceId);
00137
00138 /*
00139  * Access Functions
00140  */
00141 Xboolean XPlbArb_IsError(XPlbArb *InstancePtr);
00142 void XPlbArb_ClearErrors(XPlbArb *InstancePtr, Xuint8 Master);
00143

```

```
00144 Xuint32 XPlbArb_GetErrorStatus(XPlbArb *InstancePtr, Xuint8 Master);
00145 Xuint32 XPlbArb_GetErrorAddress(XPlbArb *InstancePtr);
00146
00147 /*
00148  * Configuration
00149  */
00150 Xuint8 XPlbArb_GetNumMasters(XPlbArb *InstancePtr);
00151 void XPlbArb_EnableInterrupt(XPlbArb *InstancePtr);
00152 void XPlbArb_DisableInterrupt(XPlbArb *InstancePtr);
00153
00154 /*
00155  * Self-test functions in xplbarb_selftest.c
00156  */
00157 XStatus XPlbArb_SelfTest(XPlbArb *InstancePtr, Xuint32 TestAddress);
00158
00159 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

XPIbArb_Config Struct Reference

```
#include <xplbarb.h>
```

Detailed Description

This typedef contains configuration information for the device. This information would typically be extracted from Configuration ROM (CROM).

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint8 NumMasters](#)

Field Documentation

[Xuint32 XPIbArb_Config::BaseAddress](#)

Register base address

[Xuint16 XPIbArb_Config::DeviceId](#)

Unique ID of device

[Xuint8 XPIbArb_Config::NumMasters](#)

Number of masters on the bus

The documentation for this struct was generated from the following file:

- `plbarb/v1_01_a/src/xplbarb.h`

Generated on 29 May 2003 for Xilinx Device Drivers

plbarb/v1_01_a/src/xplbarb.c File Reference

Detailed Description

Contains required functions for the [XPlbArb](#) component. See [xplbarb.h](#) for more information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	12/7/01	First release
1.01a	rpm	05/13/02	Updated to match hw version, publicized LookupConfig

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xplbarb.h"  
#include "xplbarb_i.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

Functions

```
XStatus XPlbArb_Initialize (XPlbArb *InstancePtr, Xuint16 DeviceId)  
Xboolean XPlbArb_IsError (XPlbArb *InstancePtr)  
void XPlbArb_ClearErrors (XPlbArb *InstancePtr, Xuint8 Master)  
Xuint32 XPlbArb_GetErrorStatus (XPlbArb *InstancePtr, Xuint8 Master)  
Xuint32 XPlbArb_GetErrorAddress (XPlbArb *InstancePtr)  
Xuint8 XPlbArb_GetNumMasters (XPlbArb *InstancePtr)  
void XPlbArb_EnableInterrupt (XPlbArb *InstancePtr)  
void XPlbArb_DisableInterrupt (XPlbArb *InstancePtr)  
void XPlbArb_Reset (XPlbArb *InstancePtr)  
XPlbArb_Config * XPlbArb_LookupConfig (Xuint16 DeviceId)
```

Function Documentation

```
void XPlbArb_ClearErrors( XPlbArb * InstancePtr,  
                          Xuint8   Master  
                          )
```

Clears the Errors for the specified master

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters - 1 on the bus

Returns:

None.

Note:

None.

```
void XPlbArb_DisableInterrupt( XPlbArb * InstancePtr)
```

Disables the interrupt output from the arbiter

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None

Note:

The Arbiter hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt source with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

```
void XPlbArb_EnableInterrupt( XPlbArb * InstancePtr)
```

Enables the interrupt output from the arbiter

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None.

Note:

The Arbiter hardware generates interrupts in error conditions. These interrupts are not handled by the driver directly. It is the application's responsibility to attach to the appropriate interrupt source with a handler which then calls functions provided by this driver to determine the cause of the error and take the necessary actions to correct the situation.

Xuint32 XPlbArb_GetErrorAddress(XPlbArb * InstancePtr)

Returns the PLB Address where the most recent error occurred. If there isn't an outstanding error, the last address in error is returned. 0x00000000 is the initial value coming out of reset.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

Address where error causing access occurred

Note:

Calling **XPlbArb_IsError()** is recommended to confirm that an error has occurred prior to calling this function to ensure that the data in the error address register is relevant.

**Xuint32 XPlbArb_GetErrorStatus(XPlbArb * InstancePtr,
Xuint8 Master
)**

Returns the Error status for the specified master. These are bit masks.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Master of which the indicated error is to be cleared, valid range is 0 - the number of masters on the bus

Returns:

The current error status for the requested master on the PLB. The status is a bit-mask that can contain the following bit values:

XPA_DRIVING_BEAR_MASK

Indicates that an error has occurred and this master is driving the error address

XPA_ERROR_READ_MASK	Indicates the error was a read error (it is a write error otherwise).
XPA_ERROR_STATUS_LOCK_MASK	Indicates the error status and address are locked and cannot be overwritten.
XPA_PEAR_SIZE_MASK	Size of access that caused error
XPA_PEAR_TYPE_MASK	Type of access that caused error

Note:

None.

Xuint8 XPlbArb_GetNumMasters(XPlbArb * InstancePtr)

Returns the number of masters associated with the arbiter.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

The number of masters. This is a number from 1 to the maximum of 32.

Note:

The value returned from this call needs to be adjusted if it is to be used as the argument for other calls since the masters are numbered from 0 and this function returns values starting at 1.

**XStatus XPlbArb_Initialize(XPlbArb * InstancePtr,
Xuint16 DeviceId
)**

Initializes a specific **XPlbArb** instance. Looks up the configuration for the given device instance and initialize the instance structure.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XPlbArb** component.

Returns:

- o XST_SUCCESS if everything starts up as expected.
- o XST_DEVICE_NOT_FOUND if the requested device is not found

Note:

None.

Xboolean XPlbArb_IsError(XPlbArb * InstancePtr)

Returns XTRUE if there is an error outstanding

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

Boolean XTRUE if there is an error, XFALSE if there is no current error.

Note:

None.

XPlbArb_Config* XPlbArb_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table XPlbArb_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or XNULL if no match is found.

Note:

None.

void XPlbArb_Reset(XPlbArb * InstancePtr)

Forces a software reset to occur in the arbiter. Disables interrupts in the process.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

Returns:

None.

Note:

Disables interrupts in the process.

plbarb/v1_01_a/src/xplbarb_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xplbarb_i.h,v 1.2 2002/06/26 22:58:36 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file plbarb/v1_01_a/src/xplbarb_i.h
00026 *
00027 * This file contains data which is shared between files and internal to the
00028 * XPlbArb component. It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date      Changes
00034 * -----
00035 * 1.00a  ecm    02/28/02  First release
00036 * 1.01a  rpm    05/13/02  Updated to match hw version, moved identifiers
00037 *                               to xplbarb_l.h
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XPLBARB_I_H /* prevent circular inclusions */
```

```
00043 #define XPLBARB_I_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046 #include "xplbarb_1.h"
00047
00048 /***** Constant Definitions *****/
00049
00050 /***** Type Definitions *****/
00051
00052 /***** Macros (Inline Functions) Definitions *****/
00053
00054 /***** Function Prototypes *****/
00055
00056 /***** Variables *****/
00057
00058 extern XPlbArb_Config XPlbArb_ConfigTable[];
00059
00060
00061 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

plbarb/v1_01_a/src/xplbarb_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XPlbArb** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	ecm	02/28/02	First release
1.01a	rpm	05/13/02	Updated to match hw version, moved identifiers to xplbarb_1.h

```
#include "xplbarb\_1.h"
```

[Go to the source code of this file.](#)

Variables

[XPlbArb_Config](#) [XPlbArb_ConfigTable](#) []

Variable Documentation

XPlbArb_Config XPlbArb_ConfigTable[]()

The PLB Arbiter configuration table, sized by the number of instances defined in [xparameters.h](#).

Generated on 29 May 2003 for Xilinx Device Drivers

plbarb/v1_01_a/src/xplbarb_l.h File Reference

Detailed Description

This file contains internal identifiers and low-level macros that can be used to access the device directly. See [xplbarb.h](#) for a description of the high-level driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.01a	rpm	05/10/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xio_dcr.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XPlbArb_mGetPesrMerrReg(BaseAddress)  
#define XPlbArb_mSetPesrMerrReg(BaseAddress, Mask)  
#define XPlbArb_mGetPesrMDriveReg(BaseAddress)  
#define XPlbArb_mGetPesrRnwReg(BaseAddress)  
#define XPlbArb_mGetPesrLockReg(BaseAddress)
```

```
#define XPlbArb_mGetPearAddrReg(BaseAddress)
#define XPlbArb_mGetPearByteEnReg(BaseAddress)
#define XPlbArb_mGetControlReg(BaseAddress)
#define XPlbArb_mEnableInterrupt(BaseAddress)
#define XPlbArb_mDisableInterrupt(BaseAddress)
#define XPlbArb_mReset(BaseAddress)
```

Define Documentation

```
#define XPlbArb_mDisableInterrupt( BaseAddress )
```

Disable interrupts in the bridge. Preserve the contents of the ctrl register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XPlbArb_mEnableInterrupt( BaseAddress )
```

Enable interrupts in the bridge. Preserve the contents of the ctrl register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XPlbArb_mGetControlReg( BaseAddress )
```

Get the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit control register contents.

Note:

None.

```
#define XPlbArb_mGetPearAddrReg( BaseAddress )
```

Get the error address (or PEAR), which is the address that just caused the error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit error address.

Note:

None.

```
#define XPlbArb_mGetPearByteEnReg( BaseAddress )
```

Get the error address byte enable register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit error address byte enable register contents.

Note:

None.

```
#define XPlbArb_mGetPesrLockReg( BaseAddress )
```

Get the value of the lock bit register, which indicates whether the master has locked the error registers.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the PESR Lock error register.

Note:

None.

```
#define XPlbArb_mGetPesrMDriveReg( BaseAddress )
```

Get the master driving the error, if any.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the PESR Master driving error register.

Note:

None.

```
#define XPlbArb_mGetPesrMerrReg( BaseAddress )
```

Get the error status register.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the error status register.

Note:

None.

```
#define XPIbArb_mGetPesrRnwReg( BaseAddress )
```

Get the value of the Read-Not-Write register, which indicates whether the error is a read error or write error.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value of the PESR RNW error register.

Note:

None.

```
#define XPIbArb_mReset( BaseAddress )
```

Reset the bridge. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XPlbArb_mSetPesrMerrReg( BaseAddress,  
                                Mask          )
```

Set the error status register.

Parameters:

BaseAddress is the base address of the device

Mask is the 32-bit value to write to the error status register.

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

plbarb/v1_01_a/src/xplbarb_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xplbarb_l.h,v 1.3 2002/07/26 20:25:30 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file plbarb/v1_01_a/src/xplbarb_l.h
00026 *
00027 * This file contains internal identifiers and low-level macros that can be
00028 * used to access the device directly.  See xplbarb.h for a description of
00029 * the high-level driver.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date        Changes
00035 * -----
00036 * 1.01a rpm    05/10/02    First release
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XPLBARB_L_H /* prevent circular inclusions */
00042 #define XPLBARB_L_H /* by using protection macros */
```



```

00043
00044 /***** Include Files *****/
00045
00046 #include "xbasic_types.h"
00047 #include "xio.h"
00048 #include "xio_dcr.h" /* DCR only, no choice */
00049
00050 /***** Constant Definitions *****/
00051
00052 /* PLB Arbiter Register offsets - DCR bus */
00053
00054 #define XPA_PESR_MERR_OFFSET 0x00 /* Master error register */
00055
00056 #define XPA_PESR_MDRIVE_OFFSET 0x01 /* Master driving error status */
00057 #define XPA_PESR_READ_OFFSET 0x02 /* set if read error,
00058 * clear if write error */
00059 #define XPA_PESR_LCK_ERR_OFFSET 0x03 /* error status locked if set */
00060 #define XPA_PEAR_ADDR_OFFSET 0x04 /* address of error */
00061 #define XPA_PEAR_BYTE_EN_OFFSET 0x05 /* byte lane(s) where error occurred
*/
00062 #define XPA_PEAR_SIZE_TYPE_OFFSET 0x06 /* size/type of error that occurred
*/
00063 #define XPA_PACR_OFFSET 0x07 /* control register */
00064
00065 /* PACR Register masks */
00066
00067 #define XPA_PACR_ENABLE_INTR_MASK 0x80000000 /* set to enable interrupts */
00068 #define XPA_PACR_SOFTWARE_RESET_MASK 0x40000000 /* set to force reset, */
00069 #define XPA_PACR_TEST_ENABLE_MASK 0x20000000 /* set to allow testing,
00070 * clear otherwise */
00071
00072 /* PEAR Size Register masks */
00073
00074 #define XPA_PEAR_SIZE_MASK 0x000000F0 /* size of access that
00075 * caused error */
00076 #define XPA_PEAR_TYPE_MASK 0x0000000E /* type of access that
00077 * caused error */
00078
00079 /* PLB Arbiter Register Masks */
00080
00081 #define XPA_DRIVING_BEAR_MASK 0x80000000 /* timeout error occurred */
00082 #define XPA_ERROR_READ_MASK 0x40000000 /* set if read error,
00083 * clear if write error */
00084 #define XPA_ERROR_STATUS_LOCK_MASK 0x20000000 /* status is locked in register
*/
00085
00086 /***** Type Definitions *****/
00087
00088 /***** Macros (Inline Functions) Definitions *****/
00089
00090 /* Define the appropriate I/O access method for the arbiter currently only
00091 * DCR

```

```

00092 */
00093 #define XPlbArb_In32    XIo_DcrIn
00094 #define XPlbArb_Out32  XIo_DcrOut
00095
00096 /*****
00097 *
00098 * Low-level driver macros and functions. The list below provides signatures
00099 * to help the user use the macros.
00100 *
00101 * Xuint32 XPlbArb_mGetPesrMerrReg(Xuint32 BaseAddress)
00102 * void XPlbArb_mSetPesrMerrReg(Xuint32 BaseAddress, Mask)
00103 *
00104 * Xuint32 XPlbArb_mGetPesrMDriveReg(Xuint32 BaseAddress)
00105 * Xuint32 XPlbArb_mGetPesrRnwReg(Xuint32 BaseAddress)
00106 * Xuint32 XPlbArb_mGetPesrLockReg(Xuint32 BaseAddress)
00107 *
00108 * Xuint32 XPlbArb_mGetPearAddrReg(Xuint32 BaseAddress)
00109 * Xuint32 XPlbArb_mGetPearByteEnReg(Xuint32 BaseAddress)
00110 * Xuint32 XPlbArb_mGetControlReg(Xuint32 BaseAddress)
00111 *
00112 * void XPlbArb_mEnableInterrupt(Xuint32 BaseAddress)
00113 * void XPlbArb_mDisableInterrupt(Xuint32 BaseAddress)
00114 * void XPlbArb_mReset(Xuint32 BaseAddress)
00115 *
00116 *****/
00117
00118 /*****
00119 /**
00120 *
00121 * Get the error status register.
00122 *
00123 * @param    BaseAddress is the base address of the device
00124 *
00125 * @return   The 32-bit value of the error status register.
00126 *
00127 * @note    None.
00128 *
00129 *****/
00130 #define XPlbArb_mGetPesrMerrReg(BaseAddress) \
00131         XPlbArb_In32((BaseAddress) + XPA_PESR_MERR_OFFSET)
00132
00133
00134 /*****
00135 /**
00136 *
00137 * Set the error status register.
00138 *
00139 * @param    BaseAddress is the base address of the device
00140 * @param    Mask is the 32-bit value to write to the error status register.
00141 *
00142 * @note    None.
00143 *

```

```
00144 *****/
00145 #define XPlbArb_mSetPesrMerrReg(BaseAddress, Mask) \
00146         XPlbArb_Out32((BaseAddress) + XPA_PESR_MERR_OFFSET, (Mask))
00147
00148
00149 *****/
00150 /**
00151 *
00152 * Get the master driving the error, if any.
00153 *
00154 * @param   BaseAddress is the base address of the device
00155 *
00156 * @return  The 32-bit value of the PESR Master driving error register.
00157 *
00158 * @note   None.
00159 *
00160 *****/
00161 #define XPlbArb_mGetPesrMDriveReg(BaseAddress) \
00162         XPlbArb_In32((BaseAddress) + XPA_PESR_MDRIVE_OFFSET)
00163
00164
00165 *****/
00166 /**
00167 *
00168 * Get the value of the Read-Not-Write register, which indicates whether the
00169 * error is a read error or write error.
00170 *
00171 * @param   BaseAddress is the base address of the device
00172 *
00173 * @return  The 32-bit value of the PESR RNW error register.
00174 *
00175 * @note   None.
00176 *
00177 *****/
00178 #define XPlbArb_mGetPesrRnwReg(BaseAddress) \
00179         XPlbArb_In32((BaseAddress) + XPA_PESR_READ_OFFSET)
00180
00181
00182 *****/
00183 /**
00184 *
00185 * Get the value of the lock bit register, which indicates whether the master
00186 * has locked the error registers.
00187 *
00188 * @param   BaseAddress is the base address of the device
00189 *
00190 * @return  The 32-bit value of the PESR Lock error register.
00191 *
00192 * @note   None.
00193 *
00194 *****/
00195 #define XPlbArb_mGetPesrLockReg(BaseAddress) \
```

```

00196             XPlbArb_In32((BaseAddress) + XPA_PESR_LCK_ERR_OFFSET)
00197
00198
00199 /*****
00200 /**
00201 *
00202 * Get the error address (or PEAR), which is the address that just caused the
00203 * error.
00204 *
00205 * @param    BaseAddress is the base address of the device
00206 *
00207 * @return   The 32-bit error address.
00208 *
00209 * @note    None.
00210 *
00211 *****/
00212 #define XPlbArb_mGetPearAddrReg(BaseAddress) \
00213             XPlbArb_In32((BaseAddress) + XPA_PEAR_ADDR_OFFSET)
00214
00215
00216 /*****
00217 /**
00218 *
00219 * Get the error address byte enable register.
00220 *
00221 * @param    BaseAddress is the base address of the device
00222 *
00223 * @return   The 32-bit error address byte enable register contents.
00224 *
00225 * @note    None.
00226 *
00227 *****/
00228 #define XPlbArb_mGetPearByteEnReg(BaseAddress) \
00229             XPlbArb_In32((BaseAddress) + XPA_PEAR_BYTE_EN_OFFSET)
00230
00231
00232 /*****
00233 /**
00234 *
00235 * Get the contents of the control register.
00236 *
00237 * @param    BaseAddress is the base address of the device
00238 *
00239 * @return   The 32-bit control register contents.
00240 *
00241 * @note    None.
00242 *
00243 *****/
00244 #define XPlbArb_mGetControlReg(BaseAddress) \
00245             XPlbArb_In32((BaseAddress) + XPA_PACR_OFFSET)
00246
00247 /*****

```

```

00248 /**
00249 *
00250 * Enable interrupts in the bridge. Preserve the contents of the ctrl register.
00251 *
00252 * @param    BaseAddress is the base address of the device
00253 *
00254 * @return   None.
00255 *
00256 * @note    None.
00257 *
00258 *****/
00259 #define XPlbArb_mEnableInterrupt(BaseAddress) \
00260     XPlbArb_Out32((BaseAddress) + XPA_PACR_OFFSET, \
00261     XPlbArb_In32((BaseAddress) + XPA_PACR_OFFSET) | \
00262     XPA_PACR_ENABLE_INTR_MASK)
00263
00264
00265 *****/
00266 /**
00267 *
00268 * Disable interrupts in the bridge. Preserve the contents of the ctrl register.
00269 *
00270 * @param    BaseAddress is the base address of the device
00271 *
00272 * @return   None.
00273 *
00274 * @note    None.
00275 *
00276 *****/
00277 #define XPlbArb_mDisableInterrupt(BaseAddress) \
00278     XPlbArb_Out32((BaseAddress) + XPA_PACR_OFFSET, \
00279     XPlbArb_In32((BaseAddress) + XPA_PACR_OFFSET) & \
00280     ~XPA_PACR_ENABLE_INTR_MASK)
00281
00282
00283 *****/
00284 /**
00285 *
00286 * Reset the bridge. Preserve the contents of the control register.
00287 *
00288 * @param    BaseAddress is the base address of the device
00289 *
00290 * @return   None.
00291 *
00292 * @note    None.
00293 *
00294 *****/
00295 #define XPlbArb_mReset(BaseAddress) \
00296     XPlbArb_Out32((BaseAddress) + XPA_PACR_OFFSET, \
00297     XPlbArb_In32((BaseAddress) + XPA_PACR_OFFSET) | \
00298     XPA_PACR_SOFTWARE_RESET_MASK)
00299

```

```
00300
00301 /***** Function Prototypes *****/
00302
00303
00304 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

plbarb/v1_01_a/src/xplbarb_selftest.c

File Reference

Detailed Description

Contains diagnostic self-test functions for the [XPlbArb](#) component. See [xplbarb.h](#) for more information about the component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	ecm	11/16/01	First release
1.01a	rpm	05/13/02	Updated to match hw version, removed _g.h

```
#include "xstatus.h"  
#include "xplbarb.h"  
#include "xplbarb_i.h"  
#include "xio_dcr.h"
```

Functions

[XStatus XPlbArb_SelfTest](#) ([XPlbArb](#) *InstancePtr, [Xuint32](#) TestAddress)

Function Documentation

```
XStatus XPlbArb_SelfTest( XPlbArb * InstancePtr,  
                          Xuint32   TestAddress  
                          )
```

Runs a self-test on the driver/device.

This tests reads the PACR to verify that the proper value is there.

XST_SUCCESS is returned if expected value is there, XST_PLBARB_FAIL_SELFTEST is returned otherwise.

Parameters:

InstancePtr is a pointer to the **XPlbArb** instance to be worked on.

TestAddress is a location that could cause an error on read, not used - user definable for hw specific implementations.

Returns:

XST_SUCCESS if successful, or XST_PLBARB_FAIL_SELFTEST if the driver fails the self test.

Note:

None.

plbarb/v1_01_a/src/xplbarb_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of PLB Arbiter devices in the system. Each arbiter device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	11/16/01	First release
1.01a	rpm	05/13/02	Updated to match hw version, removed _g.h

```
#include "xparameters.h"  
#include "xplbarb.h"
```

Variables

[XPlbArb_Config](#) [XPlbArb_ConfigTable](#) [XPAR_XPLBARB_NUM_INSTANCES]

Variable Documentation

[XPlbArb_Config](#) [XPlbArb_ConfigTable](#)[XPAR_XPLBARB_NUM_INSTANCES]

The PLB Arbiter configuration table, sized by the number of instances defined in [xparameters.h](#).

Generated on 29 May 2003 for Xilinx Device Drivers

ps2_ref/v1_00_a/src/xps2_stats.c File Reference

Detailed Description

This file contains the statistics functions for the PS/2 driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	06/24/02	First release

```
#include "xps2.h"  
#include "xps2_i.h"
```

Functions

```
void XPs2_GetStats (XPs2 *InstancePtr, XPs2Stats *StatsPtr)
```

```
void XPs2_ClearStats (XPs2 *InstancePtr)
```

Function Documentation

```
void XPs2_ClearStats( XPs2 * InstancePtr)
```

This function zeros the statistics for the given instance.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

None.

Note:

None.

```
void XPs2_GetStats( XPs2 * InstancePtr,  
                  XPs2Stats * StatsPtr  
                  )
```

This functions returns a snapshot of the current statistics in the area provided.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

StatsPtr is a pointer to a XPs2Stats structure to where the statistics are to be copied to.

Returns:

None.

Note:

None.

ps2_ref/v1_00_a/src/xps2.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xps2.h,v 1.5 2002/06/26 20:49:18 carsten Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file ps2_ref/v1_00_a/src/xps2.h
00026 *
00027 * This driver supports the following features:
00028 *
00029 * - Polled mode
00030 * - Interrupt driven mode
00031 *
00032 * <b>Interrupts</b>
00033 *
00034 * The device does not have any way to disable the receiver such that the
00035 * receiver may contain unwanted data. The IP is reset driver is initialized,
00036 *
00037 * The driver defaults to no interrupts at initialization such that interrupts
00038 * must be enabled if desired. An interrupt is generated for any of the
00039 * following
00040 * conditions.
00041 * - Data in the receiver
```

```

00042 * - Any receive status error detected
00043 * - Data byte transmitted
00044 * - Any transmit status error detected
00045 *
00046 * The application can control which interrupts are enabled using the SetOptions
00047 * function.
00048 *
00049 * In order to use interrupts, it is necessary for the user to connect the
00050 * driver interrupt handler, XPs2_InterruptHandler(), to the interrupt system of
00051 * the application. This function does not save and restore the processor
00052 * context such that the user must provide it. A handler must be set for the
00053 * driver such that the handler is called when interrupt events occur. The
00054 * handler is called from interrupt context and is designed to allow application
00055 * specific processing to be performed.
00056 *
00057 * The functions, XPs2_Send() and Ps2_Recv(), are provided in the driver to
00058 * allow data to be sent and received. They are designed to be used in polled
00059 * or interrupt modes.
00060 *
00061 * @note
00062 *
00063 * None.
00064 *
00065 * <pre>
00066 * MODIFICATION HISTORY:
00067 *
00068 * Ver    Who    Date      Changes
00069 * -----
00070 * 1.00a ch    06/18/02 First release
00071 * </pre>
00072 *
00073 *****/
00074
00075 #ifndef XPS2_H /* prevent circular inclusions */
00076 #define XPS2_H /* by using protection macros */
00077
00078 /***** Include Files *****/
00079
00080 #include "xbasic_types.h"
00081 #include "xstatus.h"
00082 #include "xps2_1.h"
00083
00084 /***** Constant Definitions *****/
00085
00086 /*
00087 * These constants specify the handler events that are passed to
00088 * a handler from the driver. These constants are not bit masks such that
00089 * only one will be passed at a time to the handler
00090 */
00091 #define XPS2_EVENT_RECV_DATA    1
00092 #define XPS2_EVENT_RECV_ERROR  2
00093 #define XPS2_EVENT_RECV_OVF    3

```



```

00146 /*
00147  * PS/2 statistics
00148  */
00149 typedef struct
00150 {
00151     Xuint16 TransmitInterrupts;
00152     Xuint16 ReceiveInterrupts;
00153     Xuint16 CharactersTransmitted;
00154     Xuint16 CharactersReceived;
00155     Xuint16 ReceiveErrors;
00156     Xuint16 ReceiveOverflowErrors;
00157     Xuint16 TransmitErrors;
00158 } XPs2Stats;
00159
00160 /*
00161  * The PS/2 driver instance data. The user is required to allocate a
00162  * variable of this type for every PS/2 device in the system.
00163  * If the last byte of a message was received then call the application
00164  * handler, this code should not use an else from the previous check of
00165  * the number of bytes to receive because the call to receive the buffer
00166  * updates the bytes to receive
00167  * A pointer to a variable of this type is then passed to the driver API
00168  * functions
00169  */
00170 typedef struct
00171 {
00172     XPs2Stats Stats;                /* Component Statistics */
00173     Xuint32 BaseAddress;            /* Base address of device (IPIF) */
00174     Xuint32 IsReady;               /* Device is initialized and ready */
00175     Xuint8  LastErrors;            /* the accumulated errors */
00176
00177     XPs2Buffer SendBuffer;
00178     XPs2Buffer ReceiveBuffer;
00179
00180     XPs2_Handler Handler;
00181     void *CallbackRef;             /* Callback reference for control handler */
00182 } XPs2;
00183
00184 /***** Macros (Inline Functions) Definitions *****/
00185
00186 /***** Function Prototypes *****/
00187
00188 /*
00189  * required functions is xps2.c
00190  */
00191 XStatus XPs2_Initialize(XPs2 *InstancePtr, Xuint16 DeviceId);
00192
00193 unsigned int XPs2_Send(XPs2 *InstancePtr, Xuint8 *BufferPtr,
00194                       unsigned int NumBytes);
00195 unsigned int XPs2_Recv(XPs2 *InstancePtr, Xuint8 *BufferPtr,
00196                       unsigned int NumBytes);

```



```
00197 XPs2_Config *XPs2_LookupConfig(Xuint16 DeviceId);
00198
00199 /*
00200  * options functions in xps2_options.c
00201  */
00202 Xuint8 XPs2_GetLastErrors(XPs2 *InstancePtr);
00203 Xboolean XPs2_IsSending(XPs2 *InstancePtr);
00204
00205 /*
00206  * interrupt functions in xps2_intr.c
00207  */
00208 void XPs2_SetHandler(XPs2 *InstancePtr, XPs2_Handler FuncPtr,
00209                    void *CallbackRef);
00210 void XPs2_InterruptHandler(XPs2 *InstancePtr);
00211 void XPs2_EnableInterrupt(XPs2 *InstancePtr);
00212 void XPs2_DisableInterrupt(XPs2 *InstancePtr);
00213
00214 #endif                /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

ps2_ref/v1_00_a/src/xps2.h File Reference

Detailed Description

This driver supports the following features:

- Polled mode
- Interrupt driven mode

Interrupts

The device does not have any way to disable the receiver such that the receiver may contain unwanted data. The IP is reset driver is initialized,

The driver defaults to no interrupts at initialization such that interrupts must be enabled if desired. An interrupt is generated for any of the following conditions.

- Data in the receiver
- Any receive status error detected
- Data byte transmitted
- Any transmit status error detected

The application can control which interrupts are enabled using the `SetOptions` function.

In order to use interrupts, it is necessary for the user to connect the driver interrupt handler, [XPs2_InterruptHandler\(\)](#), to the interrupt system of the application. This function does not save and restore the processor context such that the user must provide it. A handler must be set for the driver such that the handler is called when interrupt events occur. The handler is called from interrupt context and is designed to allow application specific processing to be performed.

The functions, **XPs2_Send()** and **Ps2_Recv()**, are provided in the driver to allow data to be sent and received. They are designed to be used in polled or interrupt modes.

Note:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	06/18/02	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xps2_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XPs2  
struct XPs2_Config  
struct XPs2Buffer  
struct XPs2Stats
```

Functions

XStatus XPs2_Initialize (XPs2 *InstancePtr, **Xuint16** DeviceId)

unsigned int **XPs2_Send** (XPs2 *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)

unsigned int **XPs2_Recv** (XPs2 *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)

XPs2_Config * **XPs2_LookupConfig** (**Xuint16** DeviceId)

Xuint8 XPs2_GetLastErrors (XPs2 *InstancePtr)

Xboolean XPs2_IsSending (XPs2 *InstancePtr)

void **XPs2_SetHandler** (XPs2 *InstancePtr, XPs2_Handler FuncPtr, void *CallBackRef)

void **XPs2_InterruptHandler** (XPs2 *InstancePtr)

void [XPs2_EnableInterrupt](#) (XPs2 *InstancePtr)

void [XPs2_DisableInterrupt](#) (XPs2 *InstancePtr)

Function Documentation

void XPs2_DisableInterrupt(XPs2 * *InstancePtr*)

void XPs2_DisableInterrupt

This function disables the PS/2 interrupts.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

None.

Note:

None.

void XPs2_EnableInterrupt(XPs2 * *InstancePtr*)

This function enables the PS/2 interrupts.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

None.

Note:

None.

Xuint8 XPs2_GetLastErrors(XPs2 * *InstancePtr*)

This function returns the last errors that have occurred in the specified PS/2 port. It also clears the errors such that they cannot be retrieved again. The errors include parity error, receive overrun error, framing error, and break detection.

The last errors is an accumulation of the errors each time an error is discovered in the driver. A status is checked for each received byte and this status is accumulated in the last errors.

If this function is called after receiving a buffer of data, it will indicate any errors that occurred for the bytes of the buffer. It does not indicate which bytes contained errors.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

The last errors that occurred. The errors are bit masks that are contained in the file [xps2.h](#) and named XPS2_ERROR_*.

Note:

None.

```
XStatus XPs2_Initialize( XPs2 * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initializes a specific PS/2 instance such that it is ready to be used. The default operating mode of the driver is polled mode.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

DeviceId is the unique id of the device controlled by this XPs2 instance. Passing in a device id associates the generic XPs2 instance to a specific device, as chosen by the caller or application developer.

Returns:

- o XST_SUCCESS if initialization was successful
- o XST_DEVICE_NOT_FOUND if the device ID could not be found in the configuration table

Note:

None.

void XPs2_InterruptHandler(XPs2 * *InstancePtr*)

This function is the interrupt handler for the PS/2 driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any PS/2 port occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the PS/2 port that the interrupt is for.

Returns:

None.

Note:

None.

Xboolean XPs2_IsSending(XPs2 * *InstancePtr*)

This function determines if the specified PS/2 port is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

A value of XTRUE if the transmitter is sending data, otherwise XFALSE.

Note:

None.

XPs2_Config* XPs2_LookupConfig(**Xuint16 *DeviceId*)**

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found.

Note:

None.

```
unsigned int XPs2_Recv( XPs2 *      InstancePtr,  
                      Xuint8 *    BufferPtr,  
                      unsigned int NumBytes  
                      )
```

This function will attempt to receive a specified number of bytes of data from PS/2 and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the PS/2 port.

In a polled mode, this function will only receive 1 byte which is as much data as the receiver can buffer. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```
unsigned int XPs2_Send( XPs2 *      InstancePtr,  
                      Xuint8 *    BufferPtr,  
                      unsigned int NumBytes  
                      )
```

This functions sends the specified buffer of data to the PS/2 port in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent thorough PS/2. If the port is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send 1 byte which is as much data as the transmitter can buffer. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.

NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

This function modifies shared data such that there may be a need for mutual exclusion in a multithreaded environment


```
void XPs2_SetHandler( XPs2 *      InstancePtr,
                    XPs2_Handler FuncPtr,
                    void *        CallbackRef
                    )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

ps2_ref/v1_00_a/src/xps2.c File Reference

Detailed Description

This file contains the required functions for the PS/2 driver. Refer to the header file [xps2.h](#) for more detailed information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	06/18/02	First release
1.00a	rmm	05/14/03	Fixed diab compiler warnings relating to asserts.

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xps2.h"  
#include "xps2_i.h"  
#include "xps2_l.h"  
#include "xio.h"
```

Functions

XStatus XPs2_Initialize (XPs2 *InstancePtr, **Xuint16** DeviceId)
unsigned int **XPs2_Send** (XPs2 *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)
unsigned int **XPs2_Recv** (XPs2 *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)

unsigned int **XPs2_SendBuffer** (XPs2 *InstancePtr)
unsigned int **XPs2_ReceiveBuffer** (XPs2 *InstancePtr)
XPs2_Config * **XPs2_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
XStatus XPs2_Initialize( XPs2 * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initializes a specific PS/2 instance such that it is ready to be used. The default operating mode of the driver is polled mode.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

DeviceId is the unique id of the device controlled by this XPs2 instance. Passing in a device id associates the generic XPs2 instance to a specific device, as chosen by the caller or application developer.

Returns:

- o XST_SUCCESS if initialization was successful
- o XST_DEVICE_NOT_FOUND if the device ID could not be found in the configuration table

Note:

None.

```
XPs2_Config* XPs2_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found.

Note:

None.

unsigned int XPs2_ReceiveBuffer(XPs2 * *InstancePtr*)

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the XPs2 component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from PS/2 and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received.

In a polled mode, this function will only receive 1 byte which is as much data as the receiver can buffer. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

```
unsigned int XPs2_Recv( XPs2 *      InstancePtr,
                       Xuint8 *    BufferPtr,
                       unsigned int NumBytes
                       )
```

This function will attempt to receive a specified number of bytes of data from PS/2 and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the PS/2 port.

In a polled mode, this function will only receive 1 byte which is as much data as the receiver can buffer. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```
unsigned int XPs2_Send( XPs2 *      InstancePtr,
                       Xuint8 *    BufferPtr,
                       unsigned int NumBytes
                       )
```

This function sends the specified buffer of data to the PS/2 port in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent through PS/2. If the port is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send 1 byte which is as much data as the transmitter can buffer. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.

NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

This function modifies shared data such that there may be a need for mutual exclusion in a multithreaded environment

```
unsigned int XPs2_SendBuffer( XPs2 * InstancePtr)
```

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the XPs2 component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the PS/2 port in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent.

In a polled mode, this function will only send 1 byte which is as much data transmitter can buffer. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent

Note:

None.

ps2_ref/v1_00_a/src/xps2_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xps2.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	06/18/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XPs2_mReset(BaseAddress)  
#define XPs2_mGetStatus(BaseAddress)  
#define XPs2_mGetIntrStatus(BaseAddress)  
#define XPs2_mClearIntr(BaseAddress, ClearMask)  
#define XPs2_mIsIntrEnabled(BaseAddress, EnabledMask)
```



```
#define XPs2_mEnableIntr(BaseAddress, EnableMask)
#define XPs2_mDisableIntr(BaseAddress, DisableMask)
#define XPs2_mIsReceiveEmpty(BaseAddress)
#define XPs2_mIsTransmitFull(BaseAddress)
```

Functions

```
void XPs2_SendByte (Xuint32 BaseAddress, Xuint8 Data)
Xuint8 XPs2_RecvByte (Xuint32 BaseAddress)
```

Define Documentation

```
#define XPs2_mClearIntr( BaseAddress,
                        ClearMask  )
```

Clear pending interrupts.

Parameters:

BaseAddress contains the base address of the device. Bitmask for interrupts to be cleared. A "1" clears the interrupt.

Returns:

None.

Note:

None.

```
#define XPs2_mDisableIntr( BaseAddress,
                          DisableMask  )
```

Disable Interrupts.

Parameters:

BaseAddress contains the base address of the device. Bitmask for interrupts to be disabled.

Returns:

None.

Note:

None.

```
#define XPs2_mEnableIntr( BaseAddress,  
                          EnableMask )
```

Enable Interrupts.

Parameters:

BaseAddress contains the base address of the device. Bitmask for interrupts to be enabled.

Returns:

None.

Note:

None.

```
#define XPs2_mGetIntrStatus( BaseAddress )
```

Read the interrupt status register.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XPs2_mGetStatus( BaseAddress )
```

Read the PS/2 status register.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XPs2_mIsIntrEnabled( BaseAddress,  
                             EnabledMask )
```

Check for enabled interrupts.

Parameters:

BaseAddress contains the base address of the device. Bitmask for interrupts to be checked.

Returns:

XTRUE if the interrupt is enabled, XFALSE otherwise.

Note:

None.

```
#define XPs2_mIsReceiveEmpty( BaseAddress )
```

Determine if there is receive data in the receiver.

Parameters:

BaseAddress contains the base address of the device.

Returns:

XTRUE if there is receive data, XFALSE otherwise.

Note:

None.

```
#define XPs2_mIsTransmitFull( BaseAddress )
```

Determine if a byte of data can be sent with the transmitter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

XTRUE if a byte can be sent, XFALSE otherwise.

Note:

None.

```
#define XPs2_mReset( BaseAddress )
```

Reset the PS/2 port.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

Function Documentation

Xuint8 XPs2_RecvByte(**Xuint32** *BaseAddress*)

This function receives a byte from PS/2. It operates in the polling mode and blocks until a byte of data is received.

Parameters:

BaseAddress contains the base address of the PS/2 port.

Returns:

The data byte received by PS/2.

Note:

None.

void XPs2_SendByte(**Xuint32** *BaseAddress*,
 Xuint8 *Data*
)

This function sends a data byte to PS/2. This function operates in the polling mode and blocks until the data has been put into the transmit holding register.

Parameters:

BaseAddress contains the base address of the PS/2 port.

Data contains the data byte to be sent.

Returns:

None.

Note:

None.

ps2_ref/v1_00_a/src/xps2_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xps2_l.h,v 1.7 2003/01/07 15:27:07 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file ps2_ref/v1_00_a/src/xps2_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  The user should refer to the
00029 * hardware device specification for more details of the device operation.
00030 * High-level driver functions are defined in xps2.h.
00031 *
00032 * <pre>
00033 * MODIFICATION HISTORY:
00034 *
00035 * Ver   Who   Date       Changes
00036 * -----
00037 * 1.00a ch   06/18/02 First release
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XPS2_L_H /* prevent circular inclusions */
```

```

00043 #define XPS2_L_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048 #include "xio.h"
00049
00050 /***** Constant Definitions *****/
00051
00052 /* PS/2 register offsets */
00053 #define XPS2_RESET_OFFSET      0 /* reset register, write only */
00054 #define XPS2_STATUS_OFFSET    4 /* status register, read only */
00055 #define XPS2_RX_REG_OFFSET    8 /* receive register, read only */
00056 #define XPS2_TX_REG_OFFSET   12 /* transmit register, write only */
00057 #define XPS2_INTSTA_REG_OFFSET 16 /* int status register, read only */
00058 #define XPS2_INTCLR_REG_OFFSET 20 /* int clear register, write only */
00059 #define XPS2_INTMSET_REG_OFFSET 24 /* mask set register, read/write */
00060 #define XPS2_INTMCLR_REG_OFFSET 28 /* mask clear register, write only */
00061
00062 /* reset register bit positions */
00063 #define XPS2_CLEAR_RESET      0x00
00064 #define XPS2_RESET           0x01
00065
00066 /* status register bit positions */
00067 #define XPS2_ST_RX_FULL      0x01
00068 #define XPS2_ST_TX_FULL     0x02
00069
00070 /* interrupt register bit positions */
00071 /* used for the INTSTA, INTCLR, INTMSET, INTMCLR register */
00072 #define XPS2_INT_WDT_TOUT    0x01
00073 #define XPS2_INT_TX_NOACK    0x02
00074 #define XPS2_INT_TX_ACK     0x04
00075 #define XPS2_INT_TX_ALL     0x06
00076 #define XPS2_INT_RX_OVF     0x08
00077 #define XPS2_INT_RX_ERR     0x10
00078 #define XPS2_INT_RX_FULL    0x20
00079 #define XPS2_INT_RX_ALL     0x38
00080 #define XPS2_INT_ALL        0x3f
00081
00082 /***** Type Definitions *****/
00083
00084 /***** Macros (Inline Functions) Definitions *****/
00085
00086 /*****
00087 *
00088 * Low-level driver macros. The list below provides signatures to help the
00089 * user use the macros.
00090 *
00091 * void XPs2_mReset(Xuint32 BaseAddress)
00092 * Xuint8 XPs2_mGetStatus(Xuint32 BaseAddress)
00093 *
00094 * Xuint8 XPs2_mGetIntrStatus(Xuint32 BaseAddress)

```

```

00095 * void XPs2_mClearIntr(Xuint32 BaseAddress, Xuint8 ClearMask)
00096 * Xboolean XPs2_mIsIntrEnabled(Xuint32 BaseAddress, Xuint8 EnabledMask)
00097 * void XPs2_mEnableIntr(Xuint32 BaseAddress, Xuint8 EnableMask)
00098 * void XPs2_mDisableIntr(Xuint32 BaseAddress, Xuint8 DisableMask)
00099 *
00100 * Xboolean XPs2_mIsReceiveEmpty(Xuint32 BaseAddress)
00101 * Xboolean XPs2_mIsTransmitFull(Xuint32 BaseAddress)
00102 *
00103 *****/
00104
00105 /***/
00106 /**
00107 * Reset the PS/2 port.
00108 *
00109 * @param BaseAddress contains the base address of the device.
00110 *
00111 * @return None.
00112 *
00113 * @note None.
00114 *
00115 *****/
00116 #define XPs2_mReset(BaseAddress) \
00117     XIo_Out8(((BaseAddress) + XPS2_RESET_OFFSET), XPS2_RESET); \
00118     XIo_Out8(((BaseAddress) + XPS2_RESET_OFFSET), XPS2_CLEAR_RESET)
00119
00120 *****/
00121 /**
00122 * Read the PS/2 status register.
00123 *
00124 * @param BaseAddress contains the base address of the device.
00125 *
00126 * @return The value read from the register.
00127 *
00128 * @note None.
00129 *
00130 *****/
00131 #define XPs2_mGetStatus(BaseAddress) \
00132     (XIo_In8((BaseAddress) + XPS2_STATUS_OFFSET))
00133
00134 *****/
00135 /**
00136 * Read the interrupt status register.
00137 *
00138 * @param BaseAddress contains the base address of the device.
00139 *
00140 * @return The value read from the register.
00141 *
00142 * @note None.
00143 *
00144 *****/
00145 #define XPs2_mGetIntrStatus(BaseAddress) \
00146     (XIo_In8((BaseAddress) + XPS2_INTSTA_REG_OFFSET))

```



```
00147
00148 /*****
00149 /**
00150 * Clear pending interrupts.
00151 *
00152 * @param   BaseAddress contains the base address of the device.
00153 *          Bitmask for interrupts to be cleared. A "1" clears the interrupt.
00154 *
00155 * @return  None.
00156 *
00157 * @note    None.
00158 *
00159 *****/
00160 #define XPs2_mClearIntr(BaseAddress, ClearMask) \
00161         XIo_Out8((BaseAddress) + XPS2_INTCLR_REG_OFFSET, (ClearMask))
00162
00163 /*****
00164 /**
00165 * Check for enabled interrupts.
00166 *
00167 * @param   BaseAddress contains the base address of the device.
00168 *          Bitmask for interrupts to be checked.
00169 *
00170 * @return  XTRUE if the interrupt is enabled, XFALSE otherwise.
00171 *
00172 * @note    None.
00173 *
00174 *****/
00175 #define XPs2_mIsIntrEnabled(BaseAddress, EnabledMask) \
00176         (XIo_In8((BaseAddress) + XPS2_INTMSET_REG_OFFSET) & (EnabledMask))
00177
00178 /*****
00179 /**
00180 * Enable Interrupts.
00181 *
00182 * @param   BaseAddress contains the base address of the device.
00183 *          Bitmask for interrupts to be enabled.
00184 *
00185 * @return  None.
00186 *
00187 * @note    None.
00188 *
00189 *****/
00190 #define XPs2_mEnableIntr(BaseAddress, EnableMask) \
00191         XIo_Out8((BaseAddress) + XPS2_INTMSET_REG_OFFSET, (EnableMask))
00192
00193 /*****
00194 /**
00195 * Disable Interrupts.
00196 *
00197 * @param   BaseAddress contains the base address of the device.
00198 *          Bitmask for interrupts to be disabled.
```

```

00199 *
00200 * @return    None.
00201 *
00202 * @note     None.
00203 *
00204 *****/
00205 #define XPs2_mDisableIntr(BaseAddress, DisableMask) \
00206         XIo_Out8((BaseAddress) + XPS2_INTMCLR_REG_OFFSET, (DisableMask))
00207
00208 *****/
00209 /**
00210 * Determine if there is receive data in the receiver.
00211 *
00212 * @param    BaseAddress contains the base address of the device.
00213 *
00214 * @return   XTRUE if there is receive data, XFALSE otherwise.
00215 *
00216 * @note     None.
00217 *
00218 *****/
00219 #define XPs2_mIsReceiveEmpty(BaseAddress) \
00220         (!(XPs2_mGetStatus(BaseAddress) & XPS2_ST_RX_FULL))
00221
00222 *****/
00223 /**
00224 * Determine if a byte of data can be sent with the transmitter.
00225 *
00226 * @param    BaseAddress contains the base address of the device.
00227 *
00228 * @return   XTRUE if a byte can be sent, XFALSE otherwise.
00229 *
00230 * @note     None.
00231 *
00232 *****/
00233 #define XPs2_mIsTransmitFull(BaseAddress) \
00234         (XPs2_mGetStatus(BaseAddress) & XPS2_ST_TX_FULL)
00235
00236 *****/ Variable Definitions *****/
00237
00238 *****/ Function Prototypes *****/
00239
00240 void XPs2_SendByte(Xuint32 BaseAddress, Xuint8 Data);
00241 Xuint8 XPs2_RecvByte(Xuint32 BaseAddress);
00242
00243 *****/
00244
00245 #endif

```

ps2_ref/v1_00_a/src/xps2_l.c File Reference

Detailed Description

This file contains low-level driver functions that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ch	06/18/02	First release

```
#include "xps2_l.h"
```

Functions

```
void XPs2_SendByte (Xuint32 BaseAddress, Xuint8 Data)  
Xuint8 XPs2_RecvByte (Xuint32 BaseAddress)
```

Function Documentation

```
Xuint8 XPs2_RecvByte( Xuint32 BaseAddress)
```

This function receives a byte from PS/2. It operates in the polling mode and blocks until a byte of data is received.

Parameters:

BaseAddress contains the base address of the PS/2 port.

Returns:

The data byte received by PS/2.

Note:

None.

```
void XPs2_SendByte( Xuint32 BaseAddress,  
                   Xuint8  Data  
                   )
```

This function sends a data byte to PS/2. This function operates in the polling mode and blocks until the data has been put into the transmit holding register.

Parameters:

BaseAddress contains the base address of the PS/2 port.

Data contains the data byte to be sent.

Returns:

None.

Note:

None.

ps2_ref/v1_00_a/src/xps2_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xps2_i.h,v 1.2 2002/06/24 23:31:39 carsten Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file ps2_ref/v1_00_a/src/xps2_i.h
00026 *
00027 * This header file contains internal identifiers, which are those shared
00028 * between the files of the driver. It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver   Who   Date       Changes
00034 * -----
00035 * 1.00a ch   06/18/02 First release
00036 * </pre>
00037 *
00038 *****/
00039 #ifndef XPS2_I_H /* prevent circular inclusions */
00040 #define XPS2_I_H /* by using protection macros */
00041
00042 /***** Include Files *****/
```

```

00043
00044 #include "xps2.h"
00045
00046 /***** Constant Definitions *****/
00047
00048 /***** Type Definitions *****/
00049
00050 /***** Macros (Inline Functions) Definitions *****/
00051
00052 /*****
00053 *
00054 * This macro clears the statistics of the component instance. The purpose of
00055 * this macro is to allow common processing between the modules of the
00056 * component with less overhead than a function in the required module.
00057 *
00058 * @param InstancePtr is a pointer to the XPs2 instance to be worked on.
00059 *
00060 * @return
00061 *
00062 * None.
00063 *
00064 * @note
00065 *
00066 * Signature: void XPs2_mClearStats(XPs2 *InstancePtr)
00067 *
00068 *****/
00069 #define XPs2_mClearStats(InstancePtr) \
00070 { \
00071     InstancePtr->Stats.TransmitInterrupts = 0UL; \
00072     InstancePtr->Stats.ReceiveInterrupts = 0UL; \
00073     InstancePtr->Stats.CharactersTransmitted = 0UL; \
00074     InstancePtr->Stats.CharactersReceived = 0UL; \
00075     InstancePtr->Stats.ReceiveErrors = 0UL; \
00076     InstancePtr->Stats.ReceiveOverflowErrors = 0UL; \
00077     InstancePtr->Stats.TransmitErrors = 0UL; \
00078 }
00079
00080 /***** Variable Definitions *****/
00081
00082 extern XPs2_Config XPs2_ConfigTable[];
00083
00084 /***** Function Prototypes *****/
00085
00086 unsigned int XPs2_SendBuffer(XPs2 *InstancePtr);
00087 unsigned int XPs2_ReceiveBuffer(XPs2 *InstancePtr);
00088
00089 #endif

```

ps2_ref/v1_00_a/src/xps2_i.h File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between the files of the driver. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ch	06/18/02	First release

```
#include "xps2.h"
```

[Go to the source code of this file.](#)

Functions

unsigned int [XPs2_SendBuffer](#) (XPs2 *InstancePtr)

unsigned int [XPs2_ReceiveBuffer](#) (XPs2 *InstancePtr)

Function Documentation

```
unsigned int XPs2_ReceiveBuffer( XPs2 * InstancePtr)
```

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the XPs2 component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from PS/2 and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received.

In a polled mode, this function will only receive 1 byte which is as much data as the receiver can buffer. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

unsigned int XPs2_SendBuffer(XPs2 * *InstancePtr*)

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the XPs2 component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the PS/2 port in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent.

In a polled mode, this function will only send 1 byte which is as much data transmitter can buffer. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the

application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

ps2_ref/v1_00_a/src/xps2_options.c File Reference

Detailed Description

The implementation of the options functions for the PS/2 driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	06/24/02	First release.

```
#include "xps2.h"  
#include "xio.h"
```

Functions

[Xuint8 XPs2_GetLastErrors](#) (XPs2 *InstancePtr)

[Xboolean XPs2_IsSending](#) (XPs2 *InstancePtr)

Function Documentation

[Xuint8 XPs2_GetLastErrors](#)(XPs2 * *InstancePtr*)

This function returns the last errors that have occurred in the specified PS/2 port. It also clears the errors such that they cannot be retrieved again. The errors include parity error, receive overrun error, framing error, and break detection.

The last errors is an accumulation of the errors each time an error is discovered in the driver. A status is checked for each received byte and this status is accumulated in the last errors.

If this function is called after receiving a buffer of data, it will indicate any errors that occurred for the bytes of the buffer. It does not indicate which bytes contained errors.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

The last errors that occurred. The errors are bit masks that are contained in the file [xps2.h](#) and named XPS2_ERROR_*.

Note:

None.

Xboolean XPs2_IsSending(XPs2 * *InstancePtr*)

This function determines if the specified PS/2 port is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

A value of XTRUE if the transmitter is sending data, otherwise XFALSE.

Note:

None.

ps2_ref/v1_00_a/src/xps2_intr.c File Reference

Detailed Description

This file contains the functions that are related to interrupt processing for the PS/2 driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	06/18/02	First release

```
#include "xps2.h"  
#include "xps2_i.h"  
#include "xio.h"
```

Functions

```
void XPs2_SetHandler (XPs2 *InstancePtr, XPs2_Handler FuncPtr, void *CallBackRef)  
void XPs2_InterruptHandler (XPs2 *InstancePtr)  
void XPs2_EnableInterrupt (XPs2 *InstancePtr)  
void XPs2_DisableInterrupt (XPs2 *InstancePtr)
```

Function Documentation

```
void XPs2_DisableInterrupt( XPs2 * InstancePtr)
```

```
void XPs2_DisableInterrupt
```

This function disables the PS/2 interrupts.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

None.

Note:

None.

```
void XPs2_EnableInterrupt( XPs2 * InstancePtr)
```

This function enables the PS/2 interrupts.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

Returns:

None.

Note:

None.

```
void XPs2_InterruptHandler( XPs2 * InstancePtr)
```

This function is the interrupt handler for the PS/2 driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any PS/2 port occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the PS/2 port that the interrupt is for.

Returns:

None.

Note:

None.

```
void XPs2_SetHandler( XPs2 *      InstancePtr,
                    XPs2_Handler FuncPtr,
                    void *        CallbackRef
                    )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the XPs2 instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

rapidio/v1_00_a/src/xrapidio_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	12/13/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XRapidIo_mReadReg(BaseAddress, RegOffset)  
#define XRapidIo_mWriteReg(BaseAddress, RegOffset, Data)  
#define XRapidIo_mReset(BaseAddress)  
#define XRapidIo_mGetLinkStatus(BaseAddress)
```

Functions

unsigned [XRapidIo_SendPkt](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *PktPtr, unsigned ByteCount)

unsigned [XRapidIo_RecvPkt](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *PktPtr)

Define Documentation

```
#define XRapidIo_mGetLinkStatus( BaseAddress )
```

Get the status of the PHY link.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XRapidIo_mReadReg( BaseAddress,  
                           RegOffset  )
```

Read a 32-bit value from a register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the offset of the register to read

Returns:

The 32-bit register value

Note:

None.


```
#define X RapidIo_mReset( BaseAddress )
```

Reset the device using the IPIF reset register. Also reset the static bin counters.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define X RapidIo_mWriteReg( BaseAddress,  
                             RegOffset,  
                             Data      )
```

Write a 32-bit value to a register.

Parameters:

BaseAddress is the base address of the device

RegOffset is the offset of the register to write

Data is the value to write

Returns:

None.

Note:

None.

Function Documentation

```
unsigned X RapidIo_RecvPkt( Xuint32 BaseAddress,  
                           Xuint8 * PktPtr  
                           )
```

Receive a packet. Wait for a packet to arrive.

Parameters:

BaseAddress is the base address of the device

PktPtr is a pointer to a 64-bit word-aligned buffer where the packet will be stored.

Returns:

The size, in bytes, of the packet received, or 0 if the incoming buffer is not 64-bit address aligned.

Note:

TODO: enforce 64-bit address alignment?

```
unsigned XRapidIo_SendPkt( Xuint32 BaseAddress,  
                          Xuint8 * PktPtr,  
                          unsigned ByteCount  
                          )
```

Send a RapidIO packet. The byte count is the total packet size, including header. This function blocks waiting for the packet to be transmitted.

Parameters:

BaseAddress is the base address of the device

PktPtr is a pointer to 64-bit word-aligned packet

ByteCount is the number of bytes in the packet

Returns:

The number of bytes sent. If an error occurs, such as the data is not 64-bit word aligned, a value of 0 is returned.

Note:

The data is written to the packet buffer in 32-bit chunks.

rapidio/v1_00_a/src/xrapidio_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xrapidio_l.h,v 1.1 2002/12/19 17:28:46 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file rapidio/v1_00_a/src/xrapidio_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date        Changes
00034 * -----
00035 * 1.00a rpm   12/13/02 First release
00036 * </pre>
00037 *
00038 *****/
00039
00040 #ifndef XRAPIDIO_L_H /* prevent circular inclusions */
00041 #define XRAPIDIO_L_H /* by using protection macros */
00042
```

```

00043 /***** Include Files *****/
00044
00045 #include "xbasic_types.h"
00046 #include "xio.h"
00047
00048 /***** Constant Definitions *****/
00049
00050 /*
00051  * RapidIO Register offsets
00052  */
00053 #define XRI_DGER_OFFSET          0x1C    /* Device global interrupt enable */
00054 #define XRI_IISR_OFFSET         0x20    /* IP interrupt status */
00055 #define XRI_IIER_OFFSET         0x28    /* IP interrupt enable */
00056 #define XRI_DRST_OFFSET         0x40    /* Device reset */
00057
00058 #define XRI_LINK_OFFSET         0x50    /* link status */
00059 #define XRI_TXL_OFFSET          0x100   /* transmit length */
00060 #define XRI_TXS_OFFSET          0x120   /* transmit status */
00061 #define XRI_RXL_OFFSET          0x200   /* receive length */
00062 #define XRI_RXS_OFFSET          0x220   /* receive status */
00063
00064 #define XRI_PHY_MTCE_BLK_OFFSET  0x1000  /* PHY Port mtce block */
00065 #define XRI_PHY_LINK_TMOUT_OFFSET 0x1020 /* PHY Port link timeout CSR */
00066 #define XRI_PHY_RESP_TMOUT_OFFSET 0x1024 /* PHY Port response timeout CSR */
00067 #define XRI_PHY_GENERAL_OFFSET  0x103C  /* PHY Port general CSR */
00068 #define XRI_PHY_PORT0_OFFSET    0x1040  /* PHY Port 0 */
00069 #define XRI_PHY_PORT0_ERR_OFFSET 0x1058  /* PHY Port 0 error and status CSR
*/
00070 #define XRI_PHY_PORT0_CTL_OFFSET 0x105C  /* PHY Port 0 control CSR */
00071
00072 #define XRI_TX_OFFSET            0x2000  /* transmit packet buffer */
00073 #define XRI_RX_OFFSET            0x3000  /* receive packet buffer */
00074
00075 /*
00076  * Define the size (and therefore offset) of each transmit or receive bin.
00077  * There are technically 64 double words in each bin, but the device reserves
00078  * the last one for its own internal use.
00079  */
00080 #define XRI_BIN_SIZE_WORDS       64      /* Number of 64-bit words */
00081 #define XRI_BIN_SIZE_BYTES       512    /* Number of bytes */
00082 #define XRI_BIN_NUMBER_OF        8      /* Number of bins */
00083
00084 /*
00085  * Link status (LINK) bit masks (8-bit register)
00086  */
00087 #define XRI_LINK_TX_READY_MASK   0x80    /* PHY Tx link ready */
00088 #define XRI_LINK_RX_READY_MASK   0x40    /* PHY Rx link ready */
00089 #define XRI_LINK_RX_RESET_MASK   0x20    /* PHY Rx detecting a bus reset request
*/
00090 #define XRI_LINK_TX_NEXT_MASK    0x07    /* Next tx packet bin used by PHY */
00091
00092 /*

```

```

00093 * Transmit Length (TXL) bit masks (32-bit register)
00094 */
00095 #define XRI_TXL_COMPLETE_MASK    0x80000000 /* Packet is complete (not partial)
*/
00096 #define XRI_TXL_UNUSED1_MASK     0x7F000000 /* Unused */
00097 #define XRI_TXL_BIN_MASK         0x00FC0000 /* Packet bin number */
00098 #define XRI_TXL_BIN_SHIFT        18          /* Packet bin number shift */
00099 #define XRI_TXL_UNUSED2_MASK     0x0003FE00 /* Unused */
00100 #define XRI_TXL_LENGTH_MASK      0x000001FF /* Packet length */
00101
00102 /*
00103 * Transmit Status (TXS) bit masks (32-bit register)
00104 */
00105 #define XRI_TXS_VALID_MASK        0x80000000 /* Packet valid (not empty) */
00106 #define XRI_TXS_BIN_MASK         0x0000003F /* Packet bin number */
00107
00108 /*
00109 * Receive Length (RXL) bit masks (32-bit register)
00110 */
00111 #define XRI_RXL_COMPLETE_MASK    0x80000000 /* Packet is complete (not partial)
*/
00112 #define XRI_RXL_UNUSED1_MASK     0x7F000000 /* Unused */
00113 #define XRI_RXL_BIN_MASK         0x00FC0000 /* Packet bin number */
00114 #define XRI_RXL_BIN_SHIFT        18          /* Packet bin number shift */
00115 #define XRI_RXL_UNUSED2_MASK     0x0003FE00 /* Unused */
00116 #define XRI_RXL_LENGTH_MASK      0x000001FF /* Packet length */
00117
00118 /*
00119 * Receive Status (RXS) bit masks (32-bit register)
00120 */
00121 #define XRI_RXS_VALID_MASK        0x00000001 /* Packet valid (not empty) */
00122
00123 /*
00124 * IPIF IP Interrupt Enable/Status Register bit masks.
00125 */
00126 #define XRI_IIXR_RX_RESET_MASK   0x00000001 /* Rx link reset request from PHY
*/
00127 #define XRI_IIXR_RX_PACKET_MASK  0x00000002 /* Rx packet available (Rx Length
00128 * FIFO is non-empty) */
00129 #define XRI_IIXR_TX_ACK_MASK     0x00000004 /* Tx ack available (Tx Status
00130 * FIFO is non-empty) */
00131 #define XRI_IIXR_PHY_MGMT_MASK   0x00000008 /* PHY management interrupt */
00132 #define XRI_IIXR_RXL_OVER_MASK   0x00000010 /* Rx length FIFO overrun */
00133 #define XRI_IIXR_TXS_UNDER_MASK  0x00000020 /* Tx status FIFO underrun */
00134 #define XRI_IIXR_TXL_OVER_MASK   0x00000040 /* Tx length FIFO overrun */
00135
00136 /***** Type Definitions *****/
00137
00138
00139 /***** Macros (Inline Functions) Definitions *****/
00140
00141 /*****
00142 *

```

```

00143 * Low-level driver macros. The list below provides signatures
00144 * to help the user use the macros.
00145 *
00146 * Xuint32 XRapidIo_mReadReg(Xuint32 BaseAddress, int RegOffset)
00147 * void XRapidIo_mWriteReg(Xuint32 BaseAddress, int RegOffset, Xuint32 Data)
00148 *
00149 * void XRapidIo_mReset(Xuint32 BaseAddress)
00150 * Xuint8 XRapidIo_mGetLinkStatus(Xuint32 BaseAddress)
00151 *
00152 * unsigned XRapidIo_SendPkt(Xuint32 BaseAddress, Xuint8 *PktPtr,
00153 *                          unsigned ByteCount)
00154 * unsigned XRapidIo_RecvPkt(Xuint32 BaseAddress, Xuint8 *PktPtr)
00155 *
00156 *****/
00157
00158 /***
00159 /**
00160 *
00161 * Read a 32-bit value from a register.
00162 *
00163 * @param   BaseAddress is the base address of the device
00164 * @param   RegOffset is the offset of the register to read
00165 *
00166 * @return  The 32-bit register value
00167 *
00168 * @note    None.
00169 *
00170 *****/
00171 #define XRapidIo_mReadReg(BaseAddress, RegOffset) \
00172     XIo_In32((BaseAddress) + (RegOffset))
00173
00174
00175 *****/
00176 /**
00177 *
00178 * Write a 32-bit value to a register.
00179 *
00180 * @param   BaseAddress is the base address of the device
00181 * @param   RegOffset is the offset of the register to write
00182 * @param   Data is the value to write
00183 *
00184 * @return  None.
00185 *
00186 * @note    None.
00187 *
00188 *****/
00189 #define XRapidIo_mWriteReg(BaseAddress, RegOffset, Data) \
00190     XIo_Out32((BaseAddress) + (RegOffset), (Data))
00191
00192
00193 *****/
00194 /**

```

```

00195 *
00196 * Reset the device using the IPIF reset register. Also reset the static bin
00197 * counters.
00198 *
00199 * @param    BaseAddress is the base address of the device
00200 *
00201 * @return   None.
00202 *
00203 * @note     None.
00204 *
00205 *****/
00206 #define XRapidIo_mReset(BaseAddress) \
00207 { \
00208     extern int XRapidIo_TxBin; \
00209     XIo_Out32((BaseAddress) + XRI_DRST_OFFSET, 0x0000000A); \
00210     XRapidIo_TxBin = 0; \
00211 }
00212
00213 *****/
00214 /**
00215 *
00216 * Get the status of the PHY link.
00217 *
00218 * @param    BaseAddress is the base address of the device
00219 *
00220 * @return   None.
00221 *
00222 * @note     None.
00223 *
00224 *****/
00225 #define XRapidIo_mGetLinkStatus(BaseAddress) \
00226     XIo_In8((BaseAddress) + XRI_LINK_OFFSET)
00227
00228
00229 *****/ Variable Definitions *****/
00230
00231
00232 *****/ Function Prototypes *****/
00233
00234 unsigned XRapidIo_SendPkt(Xuint32 BaseAddress, Xuint8 *PktPtr,
00235                          unsigned ByteCount);
00236 unsigned XRapidIo_RecvPkt(Xuint32 BaseAddress, Xuint8 *PktPtr);
00237
00238
00239 #endif          /* end of protection macro */

```

rapidio/v1_00_a/src/xrapidio_l.c File Reference

Detailed Description

This file contains low-level polled functions to send and receive RapidIO frames.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	12/13/02	First release

```
#include "xrapidio_l.h"  
#include "xio.h"
```

Functions

unsigned [XRapidIo_SendPkt](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *PktPtr, unsigned ByteCount)

unsigned [XRapidIo_RecvPkt](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *PktPtr)

Function Documentation

```
unsigned XRapidIo_RecvPkt( Xuint32 BaseAddress,  
                          Xuint8 * PktPtr  
                          )
```


Receive a packet. Wait for a packet to arrive.

Parameters:

BaseAddress is the base address of the device

PktPtr is a pointer to a 64-bit word-aligned buffer where the packet will be stored.

Returns:

The size, in bytes, of the packet received, or 0 if the incoming buffer is not 64-bit address aligned.

Note:

TODO: enforce 64-bit address alignment?

```
unsigned XRapidIo_SendPkt( Xuint32 BaseAddress,  
                          Xuint8 * PktPtr,  
                          unsigned ByteCount  
                          )
```

Send a RapidIO packet. The byte count is the total packet size, including header. This function blocks waiting for the packet to be transmitted.

Parameters:

BaseAddress is the base address of the device

PktPtr is a pointer to 64-bit word-aligned packet

ByteCount is the number of bytes in the packet

Returns:

The number of bytes sent. If an error occurs, such as the data is not 64-bit word aligned, a value of 0 is returned.

Note:

The data is written to the packet buffer in 32-bit chunks.

sysace/v1_00_a/src/xsysace_l.h File Reference

Detailed Description

Defines identifiers and low-level macros/functions for the **XSysAce** driver. These identifiers include register offsets and bit masks. A high-level driver interface is defined in [xsysace.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	06/14/02	work in progress

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Register Offsets

System ACE register offsets

```
#define XSA_BMR_OFFSET  
#define XSA_SR_OFFSET  
#define XSA_ER_OFFSET  
#define XSA_CLR_OFFSET
```

```
#define XSA_MLR_OFFSET
#define XSA_SCCR_OFFSET
#define XSA_VR_OFFSET
#define XSA_CR_OFFSET
#define XSA_FSR_OFFSET
#define XSA_DBR_OFFSET
```

Status Values

Status Register masks

```
#define XSA_SR_CFGLOCK_MASK
#define XSA_SR_MPULOCK_MASK
#define XSA_SR_CFGERROR_MASK
#define XSA_SR_CFCERROR_MASK
#define XSA_SR_CFDETECT_MASK
#define XSA_SR_DATABUFRDY_MASK
#define XSA_SR_DATABUFMODE_MASK
#define XSA_SR_CFGDONE_MASK
#define XSA_SR_RDYFORCMD_MASK
#define XSA_SR_CFGMODE_MASK
#define XSA_SR_CFGADDR_MASK
#define XSA_SR_CFBSY_MASK
#define XSA_SR_CFRDY_MASK
#define XSA_SR_CFDWF_MASK
#define XSA_SR_CFDSC_MASK
#define XSA_SR_CFDQRQ_MASK
#define XSA_SR_CFCORR_MASK
#define XSA_SR_CFERR_MASK
```

Error Values

Error Register masks.

```
#define XSA_ER_CARD_RESET
#define XSA_ER_CARD_READY
```

```
#define XSA_ER_CARD_READ
#define XSA_ER_CARD_WRITE
#define XSA_ER_SECTOR_READY
#define XSA_ER_CFG_ADDR
#define XSA_ER_CFG_FAIL
#define XSA_ER_CFG_READ
#define XSA_ER_CFG_INSTR
#define XSA_ER_CFG_INIT
#define XSA_ER_RESERVED
#define XSA_ER_BAD_BLOCK
#define XSA_ER_UNCORRECTABLE
#define XSA_ER_SECTOR_ID
#define XSA_ER_ABORT
#define XSA_ER_GENERAL
```

Sector Count/Command Values

Sector Count Command Register masks

```
#define XSA_SCCR_COUNT_MASK
#define XSA_SCCR_RESET_MASK
#define XSA_SCCR_IDENTIFY_MASK
#define XSA_SCCR_READDATA_MASK
#define XSA_SCCR_WRITEDATA_MASK
#define XSA_SCCR_ABORT_MASK
#define XSA_SCCR_CMD_MASK
```

Control Values

Control Register masks

```
#define XSA_CR_FORCELOCK_MASK
#define XSA_CR_LOCKREQ_MASK
#define XSA_CR_FORCECFGADDR_MASK
#define XSA_CR_FORCECFGMODE_MASK
#define XSA_CR_CFGMODE_MASK
```

```
#define XSA_CR_CFGSTART_MASK
#define XSA_CR_CFGSEL_MASK
#define XSA_CR_CFGRESET_MASK
#define XSA_CR_DATARDYIRQ_MASK
#define XSA_CR_ERRORIRQ_MASK
#define XSA_CR_CFGDONEIRQ_MASK
#define XSA_CR_RESETIRQ_MASK
#define XSA_CR_CFGPROG_MASK
#define XSA_CR_CFGADDR_MASK
#define XSA_CR_CFGADDR_SHIFT
```

FAT Status

FAT filesystem status masks. The first valid partition of the CF is a FAT partition.

```
#define XSA_FAT_VALID_BOOT_REC
#define XSA_FAT_VALID_PART_REC
#define XSA_FAT_12_BOOT_REC
#define XSA_FAT_12_PART_REC
#define XSA_FAT_16_BOOT_REC
#define XSA_FAT_16_PART_REC
#define XSA_FAT_12_CALC
#define XSA_FAT_16_CALC
```

Defines

```
#define XSA_BMR_16BIT_MASK
#define XSA_CLR_LBA_MASK
#define XSA_MLR_LBA_MASK
#define XSA_DATA_BUFFER_SIZE
#define XSA_CF_SECTOR_SIZE
#define XSysAce_mGetControlReg(BaseAddress)
#define XSysAce_mSetControlReg(BaseAddress, Data)
#define XSysAce_mOrControlReg(BaseAddress, Data)
#define XSysAce_mAndControlReg(BaseAddress, Data)
#define XSysAce_mGetErrorReg(BaseAddress)
```

```
#define XSysAce_mGetStatusReg(BaseAddress)
#define XSysAce_mSetCfgAddr(BaseAddress, Address)
#define XSysAce_mWaitForLock(BaseAddress)
#define XSysAce_mEnableIntr(BaseAddress, Mask)
#define XSysAce_mDisableIntr(BaseAddress, Mask)
#define XSysAce_mIsReadyForCmd(BaseAddress)
#define XSysAce_mIsMpuLocked(BaseAddress)
#define XSysAce_mIsCfgDone(BaseAddress)
#define XSysAce_mIsIntrEnabled(BaseAddress)
```

Functions

```
int XSysAce_ReadSector (Xuint32 BaseAddress, Xuint32 SectorId, Xuint8 *BufferPtr)
int XSysAce_WriteSector (Xuint32 BaseAddress, Xuint32 SectorId, Xuint8 *BufferPtr)
Xuint32 XSysAce_RegRead32 (Xuint32 Address)
Xuint16 XSysAce_RegRead16 (Xuint32 Address)
void XSysAce_RegWrite32 (Xuint32 Address, Xuint32 Data)
void XSysAce_RegWrite16 (Xuint32 Address, Xuint16 Data)
int XSysAce_ReadDataBuffer (Xuint32 BaseAddress, Xuint8 *BufferPtr, int NumBytes)
int XSysAce_WriteDataBuffer (Xuint32 BaseAddress, Xuint8 *BufferPtr, int NumBytes)
```

Define Documentation

```
#define XSA_BMR_16BIT_MASK
```

16-bit access to ACE controller

```
#define XSA_BMR_OFFSET
```

Bus mode (BUSMODEREG)

```
#define XSA_CF_SECTOR_SIZE
```

Number of bytes in a CF sector

```
#define XSA_CLR_LBA_MASK
```

Config LBA Register - address mask

#define XSA_CLR_OFFSET

Config LBA (CFGLBAREG)

#define XSA_CR_CFGADDR_MASK

Config address mask

#define XSA_CR_CFGADDR_SHIFT

Config address shift

#define XSA_CR_CFGDONEIRQ_MASK

Enable CFG done IRQ

#define XSA_CR_CFGMODE_MASK

CFG mode

#define XSA_CR_CFGPROG_MASK

Inverted CFGPROG pin

#define XSA_CR_CFGRESET_MASK

CFG reset

#define XSA_CR_CFGSEL_MASK

CFG select

#define XSA_CR_CFGSTART_MASK

CFG start

#define XSA_CR_DATARDYIRQ_MASK

Enable data ready IRQ

#define XSA_CR_ERRORIRQ_MASK

Enable error IRQ

#define XSA_CR_FORCECFGADDR_MASK

Force CFG address

#define XSA_CR_FORCECFGMODE_MASK

Force CFG mode

#define XSA_CR_FORCELOCK_MASK

Force lock request

#define XSA_CR_LOCKREQ_MASK

MPU lock request

#define XSA_CR_OFFSET

Control (CONTROLREG)

#define XSA_CR_RESETIRQ_MASK

Reset IRQ line

#define XSA_DATA_BUFFER_SIZE

Size of System ACE data buffer

#define XSA_DBR_OFFSET

Data buffer (DATABUFREG)

#define XSA_ER_ABORT

CF command aborted

#define XSA_ER_BAD_BLOCK

CF bad block detected

#define XSA_ER_CARD_READ

CF read command failed

#define XSA_ER_CARD_READY

CF card failed to ready

#define XSA_ER_CARD_RESET

CF card failed to reset

#define XSA_ER_CARD_WRITE

CF write command failed

#define XSA_ER_CFG_ADDR

Cfg address is invalid

#define XSA_ER_CFG_FAIL

Failed to configure a device

#define XSA_ER_CFG_INIT

CFGINIT pin error - did not go high within 500ms of start

#define XSA_ER_CFG_INSTR

Invalid instruction during cfg

#define XSA_ER_CFG_READ

Cfg read of CF failed

#define XSA_ER_GENERAL

CF general error

#define XSA_ER_OFFSET

Error (ERRORREG)

#define XSA_ER_RESERVED

reserved

#define XSA_ER_SECTOR_ID

CF sector ID not found

#define XSA_ER_SECTOR_READY

CF sector failed to ready

#define XSA_ER_UNCORRECTABLE

CF uncorrectable error

#define XSA_FAT_12_BOOT_REC

FAT12 in master boot rec

#define XSA_FAT_12_CALC

Calculated FAT12 from clusters

#define XSA_FAT_12_PART_REC

FAT12 in partition boot rec

#define XSA_FAT_16_BOOT_REC

FAT16 in master boot rec

#define XSA_FAT_16_CALC

Calculated FAT16 from clusters

#define XSA_FAT_16_PART_REC

FAT16 in partition boot rec

#define XSA_FAT_VALID_BOOT_REC

Valid master boot record

#define XSA_FAT_VALID_PART_REC

Valid partition boot record

#define XSA_FSR_OFFSET

FAT status (FATSTATREG)

#define XSA_MLR_LBA_MASK

MPU LBA Register - address mask

#define XSA_MLR_OFFSET

MPU LBA (MPULBAREG)

#define XSA_SCCR_ABORT_MASK

Abort CF command

#define XSA_SCCR_CMD_MASK

Command mask

#define XSA_SCCR_COUNT_MASK

Sector count mask

#define XSA_SCCR_IDENTIFY_MASK

Identify CF card command

#define XSA_SCCR_OFFSET

Sector cnt (SECCNTCMDREG)

#define XSA_SCCR_READDATA_MASK

Read CF card command

#define XSA_SCCR_RESET_MASK

Reset CF card command

#define XSA_SCCR_WRITEDATA_MASK

Write CF card command

#define XSA_SR_CFBSY_MASK

CF busy (BSY bit)

#define XSA_SR_CFCERROR_MASK

CF error status

#define XSA_SR_CFCORR_MASK

CF correctable error (CORR bit)

#define XSA_SR_CFDETECT_MASK

CF detect flag

#define XSA_SR_CFDRQ_MASK

CF data request (DRQ)

#define XSA_SR_CFDSC_MASK

CF ready (DSC bit)

#define XSA_SR_CFDWF_MASK

CF data write fault (DWF bit)

#define XSA_SR_CFERR_MASK

CF error (ERR bit)

#define XSA_SR_CFGADDR_MASK

Configuration address

#define XSA_SR_CFGDONE_MASK

Configuration done status

```
#define XSA_SR_CFGERROR_MASK
```

Config port error status

```
#define XSA_SR_CFGLOCK_MASK
```

Config port lock status

```
#define XSA_SR_CFGMODE_MASK
```

Configuration mode status

```
#define XSA_SR_CFRDY_MASK
```

CF ready (RDY bit)

```
#define XSA_SR_DATABUFMODE_MASK
```

Data buffer mode status

```
#define XSA_SR_DATABUFRDY_MASK
```

Data buffer ready status

```
#define XSA_SR_MPULOCK_MASK
```

MPU port lock status

```
#define XSA_SR_OFFSET
```

Status (STATUSREG)

```
#define XSA_SR_RDYFORCMD_MASK
```

Ready for CF command

```
#define XSA_VR_OFFSET
```

Version (VERSIONREG)

```
#define XSysAce_mAndControlReg( BaseAddress,  
                                Data          )
```

Set the contents of the control register to the value specified AND'ed with its current contents.

Parameters:

BaseAddress is the base address of the device.

Data is the 32-bit value to AND with the register.

Returns:

None.

Note:

None.

```
#define XSysAce_mDisableIntr( BaseAddress,  
                             Mask          )
```

Disable ACE controller interrupts.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

Note:

None.

```
#define XSysAce_mEnableIntr( BaseAddress,  
                             Mask          )
```

Enable ACE controller interrupts.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

Note:

None.

```
#define XSysAce_mGetControlReg( BaseAddress )
```

Get the contents of the control register.

Parameters:

BaseAddress is the base address of the device.

Returns:

The 32-bit value of the control register.

Note:

None.

```
#define XSysAce_mGetErrorReg( BaseAddress )
```

Get the contents of the error register.

Parameters:

BaseAddress is the base address of the device.

Returns:

The 32-bit value of the register.

Note:

None.

#define XSysAce_mGetStatusReg(BaseAddress)

Get the contents of the status register.

Parameters:

BaseAddress is the base address of the device.

Returns:

The 32-bit value of the register.

Note:

None.

#define XSysAce_mIsCfgDone(BaseAddress)

Is the CompactFlash configuration of the target FPGA chain complete?

Parameters:

BaseAddress is the base address of the device.

Returns:

XTRUE if it is ready, XFALSE otherwise.

Note:

None.

#define XSysAce_mIsIntrEnabled(BaseAddress)

Have interrupts been enabled by the user? We look for the interrupt reset bit to be clear (meaning interrupts are armed, even though none may be individually enabled).

Parameters:

BaseAddress is the base address of the device.

Returns:

XTRUE if it is enabled, XFALSE otherwise.

Note:

None.


```
#define XSysAce_mIsMpuLocked( BaseAddress )
```

Is the ACE controller locked for MPU access?

Parameters:

BaseAddress is the base address of the device.

Returns:

XTRUE if it is locked, XFALSE otherwise.

Note:

None.

```
#define XSysAce_mIsReadyForCmd( BaseAddress )
```

Is the CompactFlash ready for a command?

Parameters:

BaseAddress is the base address of the device.

Returns:

XTRUE if it is ready, XFALSE otherwise.

Note:

None.

```
#define XSysAce_mOrControlReg( BaseAddress,  
                               Data          )
```

Set the contents of the control register to the value specified OR'ed with its current contents.

Parameters:

BaseAddress is the base address of the device.

Data is the 32-bit value to OR with the register.

Returns:

None.

Note:

None.

```
#define XSysAce_mSetCfgAddr( BaseAddress,  
                             Address      )
```

Set the configuration address, or file, of the CompactFlash. This address indicates which .ace bitstream to use to configure the target FPGA chain.

Parameters:

BaseAddress is the base address of the device.

Address ranges from 0 to 7 and represents the eight possible .ace bitstreams that can reside on the CompactFlash.

Returns:

None.

Note:

Used cryptic var names to avoid conflict with caller's var names.

```
#define XSysAce_mSetControlReg( BaseAddress,  
                                Data        )
```

Set the contents of the control register.

Parameters:

BaseAddress is the base address of the device.

Data is the 32-bit value to write to the register.

Returns:

None.

Note:

None.

```
#define XSysAce_mWaitForLock( BaseAddress )
```

Request then wait for the MPU lock. This is not a forced lock, so we must contend with the configuration controller.

Parameters:

BaseAddress is the base address of the device.

Returns:

None.

Note:

None.

Function Documentation

```
int XSysAce_ReadDataBuffer( Xuint32 BaseAddress,  
                           Xuint8 * BufferPtr,  
                           int Size  
                           )
```

Read the specified number of bytes from the data buffer of the ACE controller. The data buffer, which is 32 bytes, can only be read two bytes at a time. Once the data buffer is read, we wait for it to be filled again before reading the next buffer's worth of data.

Parameters:

BaseAddress is the base address of the device
BufferPtr is a pointer to a buffer in which to store data.
Size is the number of bytes to read

Returns:

The total number of bytes read, or 0 if an error occurred.

Note:

If *Size* is not aligned with the size of the data buffer (32 bytes), this function will read the entire data buffer, dropping the extra bytes on the floor since the user did not request them. This is necessary to get the data buffer to be ready again.

```
int XSysAce_ReadSector( Xuint32 BaseAddress,  
                        Xuint32 SectorId,  
                        Xuint8 * BufferPtr  
                        )
```

Read a CompactFlash sector. This is a blocking, low-level function which does not return until the specified sector is read.

Parameters:

BaseAddress is the base address of the device
SectorId is the id of the sector to read
BufferPtr is a pointer to a buffer where the data will be stored.

Returns:

The number of bytes read. If this number is not equal to the sector size, 512 bytes, then an error occurred.

Note:

None.

```
Xuint16 XSysAce_RegRead16( Xuint32 Address)
```

Read a 16-bit value from the given address. Based on a compile-time constant, do the read in one 16-bit read or two 8-bit reads.

Parameters:

Address is the address to read from.

Returns:

The 16-bit value of the address.

Note:

No need for endian conversion in 8-bit mode since this function gets the bytes into their proper lanes in the 16-bit word.

```
Xuint32 XSysAce_RegRead32( Xuint32 Address )
```

Read a 32-bit value from the given address. Based on a compile-time constant, do the read in two 16-bit reads or four 8-bit reads.

Parameters:

Address is the address to read from.

Returns:

The 32-bit value of the address.

Note:

No need for endian conversion in 8-bit mode since this function gets the bytes into their proper lanes in the 32-bit word.

```
void XSysAce_RegWrite16( Xuint32 Address,  
                        Xuint16 Data  
                        )
```

Write a 16-bit value to the given address. Based on a compile-time constant, do the write in one 16-bit write or two 8-bit writes.

Parameters:

Address is the address to write to.

Data is the value to write

Returns:

None.

Note:

No need for endian conversion in 8-bit mode since this function writes the bytes into their proper lanes based on address.

```
void XSysAce_RegWrite32( Xuint32 Address,  
                        Xuint32 Data  
                        )
```

Write a 32-bit value to the given address. Based on a compile-time constant, do the write in two 16-bit writes or four 8-bit writes.

Parameters:

Address is the address to write to.

Data is the value to write

Returns:

None.

Note:

No need for endian conversion in 8-bit mode since this function writes the bytes into their proper lanes based on address.

```
int XSysAce_WriteDataBuffer( Xuint32 BaseAddress,  
                             Xuint8 * BufferPtr,  
                             int Size  
                             )
```

Write the specified number of bytes to the data buffer of the ACE controller. The data buffer, which is 32 bytes, can only be written two bytes at a time. Once the data buffer is written, we wait for it to be empty again before writing the next buffer's worth of data. If the size of the incoming buffer is not aligned with the System ACE data buffer size (32 bytes), then this routine pads out the data buffer with zeros so the entire data buffer is written. This is necessary for the ACE controller to process the data buffer.

Parameters:

BaseAddress is the base address of the device

BufferPtr is a pointer to a buffer used to write to the controller.

Size is the number of bytes to write

Returns:

The total number of bytes written (not including pad bytes), or 0 if an error occurs.

Note:

None.

```
int XSysAce_WriteSector( Xuint32 BaseAddress,  
                        Xuint32 SectorId,  
                        Xuint8 * BufferPtr  
                        )
```

Write a CompactFlash sector. This is a blocking, low-level function which does not return until the specified sector is written in its entirety.

Parameters:

BaseAddress is the base address of the device

SectorId is the id of the sector to write

BufferPtr is a pointer to a buffer used to write the sector.

Returns:

The number of bytes written. If this number is not equal to the sector size, 512 bytes, then an error occurred.

Note:

None.

XSysAce Struct Reference

```
#include <xsysace.h>
```

Detailed Description

The XSysAce driver instance data. The user is required to allocate a variable of this type for every System ACE device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `sysace/v1_00_a/src/xsysace.h`

sysace/v1_00_a/src/xsysace.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xsysace.h,v 1.8 2002/11/04 23:44:21 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file sysace/v1_00_a/src/xsysace.h
00026 *
00027 * The Xilinx System ACE driver. This driver supports the Xilinx System Advanced
00028 * Configuration Environment (ACE) controller. It currently supports only the
00029 * CompactFlash solution. The driver makes use of the Microprocessor (MPU)
00030 * interface to communicate with the device.
00031 *
00032 * The driver provides a user the ability to access the CompactFlash through
00033 * the System ACE device. The user can read and write CompactFlash sectors,
00034 * identify the flash device, and reset the flash device. Also, the driver
00035 * provides a user the ability to configure FPGA devices by selecting a
00036 * configuration file (.ace file) resident on the CompactFlash, or directly
00037 * configuring the FPGA devices via the MPU port and the configuration JTAG
00038 * port of the controller.
00039 *
00040 * <b>Bus Mode</b>
00041 *
00042 * The System ACE device supports both 8-bit and 16-bit access to its registers.
```

00043 * The driver defaults to 8-bit access, but can be changed to use 16-bit access
00044 * at compile-time. The compile-time constant XPAR_XSYSACE_MEM_WIDTH must be
00045 * defined equal to 16 to make the driver use 16-bit access. This constant is
00046 * typically defined in xparameters.h.
00047 *
00048 * **Endianness**
00049 *
00050 * The System ACE device is little-endian. If being accessed by a big-endian
00051 * processor, the endian conversion will be done by the device driver. The
00052 * endian conversion is encapsulated inside the XSysAce_RegRead/Write functions
00053 * so that it can be removed if the endian conversion is moved to hardware.
00054 *
00055 * **Hardware Access**
00056 *
00057 * The device driver expects the System ACE controller to be a memory-mapped
00058 * device. Access to the System ACE controller is typically achieved through
00059 * the External Memory Controller (EMC) IP core. The EMC is simply a pass-
through
00060 * device that allows access to the off-chip System ACE device. There is no
00061 * software-based setup or configuration necessary for the EMC.
00062 *
00063 * The System ACE registers are expected to be byte-addressable. If for some
00064 * reason this is not possible, the register offsets defined in xsysace_l.h must
00065 * be changed accordingly.
00066 *
00067 * **Reading or Writing CompactFlash**
00068 *
00069 * The smallest unit that can be read from or written to CompactFlash is one
00070 * sector. A sector is 512 bytes. The functions provided by this driver allow
00071 * the user to specify a starting sector ID and the number of sectors to be read
00072 * or written. At most 256 sectors can be read or written in one operation. The
00073 * user must ensure that the buffer passed to the functions is big enough to
00074 * hold (512 * NumSectors), where NumSectors is the number of sectors specified.
00075 *
00076 * **Interrupt Mode**
00077 *
00078 * By default, the device and driver are in polled mode. The user is required to
00079 * enable interrupts using XSysAce_EnableInterrupt(). In order to use
interrupts,
00080 * it is necessary for the user to connect the driver's interrupt handler,
00081 * XSysAce_InterruptHandler(), to the interrupt system of the application. This
00082 * function does not save and restore the processor context. An event handler
00083 * must also be set by the user, using XSysAce_SetEventHandler(), for the driver
00084 * such that the handler is called when interrupt events occur. The handler is
00085 * called from interrupt context and allows application-specific processing to
00086 * be performed.
00087 *
00088 * In interrupt mode, the only available interrupt is data buffer ready, so
00089 * the size of a data transfer between interrupts is 32 bytes (the size of the
00090 * data buffer).
00091 *
00092 * **Polled Mode**
00093 *

```

00094 * The sector read and write functions are blocking when in polled mode. This
00095 * choice was made over non-blocking since sector transfer rates are high
00096 * (>20Mbps) and the user can limit the number of sectors transferred in a
single
00097 * operation to 1 when in polled mode, plus the API for non-blocking polled
00098 * functions was a bit awkward. Below is some more information on the sector
00099 * transfer rates given the current state of technology (year 2002). Although
00100 * the seek times for CompactFlash cards is high, this average hit needs to be
00101 * taken every time a new read/write operation is invoked by the user. So the
00102 * additional few microseconds to transfer an entire sector along with seeking
00103 * is miniscule.
00104 *
00105 * - Microdrives are slower than CompactFlash cards by a significant factor,
00106 * especially if the MD is asleep.
00107 *   - Microdrive:
00108 *     - Power-up/wake-up time is approx. 150 to 1000 ms.
00109 *     - Average seek time is approx. 15 to 20 ms.
00110 *   - CompactFlash:
00111 *     - Power-up/reset time is approx. 50 to 400 ms and wake-up time is
00112 *       approx. 3 ms.
00113 *     - "Seek time" here means how long it takes the internal controller
00114 *       to process the command until the sector data is ready for
transfer
00115 *       by the ACE controller. This time is approx. 2 ms per sector.
00116 *
00117 * - Once the sector data is ready in the CF device buffer (i.e., "seek time"
is
00118 *   over) the ACE controller can read 2 bytes from the MD/CF device every 11
00119 *   clock cycles, assuming no wait cycles happen. For instance, if the clock
00120 *   is 33 MHz, then then the max. rate that the ACE controller can transfer is
00121 *   6 MB/sec. However, due to other overhead (e.g., time for data buffer
00122 *   transfers over MPU port, etc.), a better estimate is 3-5 MB/sec.
00123 *
00124 * <b>Mutual Exclusion</b>
00125 *
00126 * This driver is not thread-safe. The System ACE device has a single data
00127 * buffer and therefore only one operation can be active at a time. The device
00128 * driver does not prevent the user from starting an operation while a previous
00129 * operation is still in progress. It is up to the user to provide this mutual
00130 * exclusion.
00131 *
00132 * <b>Errors</b>
00133 *
00134 * Error causes are defined in xsysace_l.h using the prefix XSA_ER_*. The
00135 * user can use XSysAce_GetErrors() to retrieve all outstanding errors.
00136 *
00137 * <pre>
00138 * MODIFICATION HISTORY:
00139 *
00140 * Ver    Who    Date      Changes
00141 * -----
00142 * 1.00a rpm 06/17/02 work in progress

```

```

00143 * </pre>
00144 *
00145 *****/
00146
00147 #ifndef XSYSACE_H /* prevent circular inclusions */
00148 #define XSYSACE_H /* by using protection macros */
00149
00150 /***** Include Files *****/
00151
00152 #include "xbasic_types.h"
00153 #include "xstatus.h"
00154 #include "xsysace_l.h"
00155
00156 /***** Constant Definitions *****/
00157
00158 /** @name Asynchronous Events
00159 *
00160 * Asynchronous events passed to the event handler when in interrupt mode.
00161 *
00162 * Note that when an error event occurs, the only way to clear this condition
00163 * is to reset the CompactFlash or the System ACE configuration controller,
00164 * depending on where the error occurred. The driver does not reset either
00165 * and leaves this task to the user.
00166 * @{
00167 */
00168 #define XSA_EVENT_CFG_DONE 1 /**< Configuration of JTAG chain is done */
00169 #define XSA_EVENT_DATA_DONE 2 /**< Data transfer to/from CompactFlash is done
*/
00170 #define XSA_EVENT_ERROR 3 /**< An error occurred. Use XSysAce_GetErrors()
00171 * to determine the cause of the error(s).
00172 */
00173 /*@}*/
00174
00175
00176 /***** Type Definitions *****/
00177
00178 /**
00179 * Typedef for CompactFlash identify drive parameters. Use XSysAce_IdentifyCF()
00180 * to retrieve this information from the CompactFlash storage device.
00181 */
00182 typedef struct
00183 {
00184     Xuint16 Signature; /**< CompactFlash signature is 0x848a */
00185     Xuint16 NumCylinders; /**< Default number of cylinders */
00186     Xuint16 Reserved;
00187     Xuint16 NumHeads; /**< Default number of heads */
00188     Xuint16 NumBytesPerTrack; /**< Number of unformatted bytes per track */
00189     Xuint16 NumBytesPerSector; /**< Number of unformatted bytes per sector */
00190     Xuint16 NumSectorsPerTrack; /**< Default number of sectors per track */
00191     Xuint32 NumSectorsPerCard; /**< Default number of sectors per card */
00192     Xuint16 VendorUnique; /**< Vendor unique */

```

```

00193     Xuint8 SerialNo[20];           /**< ASCII serial number */
00194     Xuint16 BufferType;            /**< Buffer type */
00195     Xuint16 BufferSize;            /**< Buffer size in 512-byte increments */
00196     Xuint16 NumEccBytes;           /**< Number of ECC bytes on R/W Long cmds */
00197     Xuint8 FwVersion[8];          /**< ASCII firmware version */
00198     Xuint8 ModelNo[40];           /**< ASCII model number */
00199     Xuint16 MaxSectors;           /**< Max sectors on R/W Multiple cmds */
00200     Xuint16 DblWord;              /**< Double Word not supported */
00201     Xuint16 Capabilities;         /**< Device capabilities */
00202     Xuint16 Reserved2;
00203     Xuint16 PioMode;              /**< PIO data transfer cycle timing mode */
00204     Xuint16 DmaMode;              /**< DMA data transfer cycle timing mode */
00205     Xuint16 TranslationValid;     /**< Translation parameters are valid */
00206     Xuint16 CurNumCylinders;      /**< Current number of cylinders */
00207     Xuint16 CurNumHeads;          /**< Current number of heads */
00208     Xuint16 CurSectorsPerTrack;   /**< Current number of sectors per track */
00209     Xuint32 CurSectorsPerCard;    /**< Current capacity in sectors */
00210     Xuint16 MultipleSectors;      /**< Multiple sector setting */
00211     Xuint32 LbaSectors;           /**< Number of addressable sectors in LBA mode
*/
00212     Xuint8 Reserved3[132];
00213     Xuint16 SecurityStatus;       /**< Security status */
00214     Xuint8 VendorUniqueBytes[62]; /**< Vendor unique bytes */
00215     Xuint16 PowerDesc;            /**< Power requirement description */
00216     Xuint8 Reserved4[190];
00217
00218 } XSysAce_CFPParameters;
00219
00220
00221 /**
00222  * Callback when an asynchronous event occurs during interrupt mode.
00223  *
00224  * @param CallBackRef is a callback reference passed in by the upper layer
00225  *         when setting the callback functions, and passed back to the upper
00226  *         layer when the callback is invoked.
00227  * @param Event is the event that occurred. See xsysace.h and the event
00228  *         identifiers prefixed with XSA_EVENT_* for a description of possible
00229  *         events.
00230  */
00231 typedef void (*XSysAce_EventHandler)(void *CallBackRef, int Event);
00232
00233 /**
00234  * This typedef contains configuration information for the device.
00235  */
00236 typedef struct
00237 {
00238     Xuint16 DeviceId;              /**< Unique ID of device */
00239     Xuint32 BaseAddress;           /**< Register base address */
00240

```



```

00290 Xuint16 XSysAce_GetFatStatus(XSysAce *InstancePtr);
00291
00292 /*
00293  * JTAG configuration interface functions in xsysace_jtagcfg.c
00294  */
00295 void XSysAce_ResetCfg(XSysAce *InstancePtr);
00296 void XSysAce_SetCfgAddr(XSysAce *InstancePtr, unsigned int Address);
00297 void XSysAce_SetStartMode(XSysAce *InstancePtr, Xboolean ImmedOnReset,
00298                          Xboolean SetStart);
00299 Xboolean XSysAce_IsCfgDone(XSysAce *InstancePtr);
00300 Xuint32 XSysAce_GetCfgSector(XSysAce *InstancePtr);
00301 XStatus XSysAce_ProgramChain(XSysAce *InstancePtr, Xuint8 *BufferPtr, int
NumBytes);
00302
00303 /*
00304  * General interrupt-related functions in xsysace_intr.c
00305  */
00306 void XSysAce_EnableInterrupt(XSysAce *InstancePtr);
00307 void XSysAce_DisableInterrupt(XSysAce *InstancePtr);
00308 void XSysAce_SetEventHandler(XSysAce *InstancePtr, XSysAce_EventHandler
FuncPtr,
00309                             void *CallBackRef);
00310 void XSysAce_InterruptHandler(void *InstancePtr);      /* interrupt handler */
00311
00312 /*
00313  * Diagnostic functions in xsysace_selftest.c
00314  */
00315 XStatus XSysAce_SelfTest(XSysAce *InstancePtr);
00316 Xuint16 XSysAce_GetVersion(XSysAce *InstancePtr);
00317
00318
00319 #endif          /* end of protection macro */
00320

```

Generated on 29 May 2003 for Xilinx Device Drivers

sysace/v1_00_a/src/xsysace.h File Reference

Detailed Description

The Xilinx System ACE driver. This driver supports the Xilinx System Advanced Configuration Environment (ACE) controller. It currently supports only the CompactFlash solution. The driver makes use of the Microprocessor (MPU) interface to communicate with the device.

The driver provides a user the ability to access the CompactFlash through the System ACE device. The user can read and write CompactFlash sectors, identify the flash device, and reset the flash device. Also, the driver provides a user the ability to configure FPGA devices by selecting a configuration file (.ace file) resident on the CompactFlash, or directly configuring the FPGA devices via the MPU port and the configuration JTAG port of the controller.

Bus Mode

The System ACE device supports both 8-bit and 16-bit access to its registers. The driver defaults to 8-bit access, but can be changed to use 16-bit access at compile-time. The compile-time constant `XPAR_XSYSACE_MEM_WIDTH` must be defined equal to 16 to make the driver use 16-bit access. This constant is typically defined in [xparameters.h](#).

Endianness

The System ACE device is little-endian. If being accessed by a big-endian processor, the endian conversion will be done by the device driver. The endian conversion is encapsulated inside the `XSysAce_RegRead/Write` functions so that it can be removed if the endian conversion is moved to hardware.

Hardware Access

The device driver expects the System ACE controller to be a memory-mapped device. Access to the System ACE controller is typically achieved through the External Memory Controller (EMC) IP core. The EMC is simply a pass-through device that allows access to the off-chip System ACE device. There is no software-based setup or configuration necessary for the EMC.

The System ACE registers are expected to be byte-addressable. If for some reason this is not possible, the register offsets defined in [xsysace_1.h](#) must be changed accordingly.

Reading or Writing CompactFlash

The smallest unit that can be read from or written to CompactFlash is one sector. A sector is 512 bytes. The functions provided by this driver allow the user to specify a starting sector ID and the number of sectors to be read or written. At most 256 sectors can be read or written in one operation. The user must ensure that the buffer passed to the functions is big enough to hold $(512 * \text{NumSectors})$, where NumSectors is the number of sectors specified.

Interrupt Mode

By default, the device and driver are in polled mode. The user is required to enable interrupts using [XSysAce_EnableInterrupt\(\)](#). In order to use interrupts, it is necessary for the user to connect the driver's interrupt handler, [XSysAce_InterruptHandler\(\)](#), to the interrupt system of the application. This function does not save and restore the processor context. An event handler must also be set by the user, using [XSysAce_SetEventHandler\(\)](#), for the driver such that the handler is called when interrupt events occur. The handler is called from interrupt context and allows application-specific processing to be performed.

In interrupt mode, the only available interrupt is data buffer ready, so the size of a data transfer between interrupts is 32 bytes (the size of the data buffer).

Polled Mode

The sector read and write functions are blocking when in polled mode. This choice was made over non-blocking since sector transfer rates are high (>20Mbps) and the user can limit the number of sectors transferred in a single operation to 1 when in polled mode, plus the API for non-blocking polled functions was a bit awkward. Below is some more information on the sector transfer rates given the current state of technology (year 2002). Although the seek times for CompactFlash cards is high, this average hit needs to be taken every time a new read/write operation is invoked by the user. So the additional few microseconds to transfer an entire sector along with seeking is miniscule.

- Microdrives are slower than CompactFlash cards by a significant factor, especially if the MD is asleep.

- Microdrive:
 - Power-up/wake-up time is approx. 150 to 1000 ms.
 - Average seek time is approx. 15 to 20 ms.
 - CompactFlash:
 - Power-up/reset time is approx. 50 to 400 ms and wake-up time is approx. 3 ms.
 - "Seek time" here means how long it takes the internal controller to process the command until the sector data is ready for transfer by the ACE controller. This time is approx. 2 ms per sector.
- Once the sector data is ready in the CF device buffer (i.e., "seek time" is over) the ACE controller can read 2 bytes from the MD/CF device every 11 clock cycles, assuming no wait cycles happen. For instance, if the clock is 33 MHz, then then the max. rate that the ACE controller can transfer is 6 MB/sec. However, due to other overhead (e.g., time for data buffer transfers over MPU port, etc.), a better estimate is 3-5 MB/sec.

Mutual Exclusion

This driver is not thread-safe. The System ACE device has a single data buffer and therefore only one operation can be active at a time. The device driver does not prevent the user from starting an operation while a previous operation is still in progress. It is up to the user to provide this mutual exclusion.

Errors

Error causes are defined in [xsysace_1.h](#) using the prefix XSA_ER_*. The user can use [XSysAce_GetErrors\(\)](#) to retrieve all outstanding errors.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	06/17/02	work in progress

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xsysace_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XSysAce  
struct XSysAce_CFPParameters  
struct XSysAce_Config
```

Asynchronous Events

Asynchronous events passed to the event handler when in interrupt mode.

Note that when an error event occurs, the only way to clear this condition is to reset the CompactFlash or the System ACE configuration controller, depending on where the error occurred. The driver does not reset either and leaves this task to the user.

```
#define XSA_EVENT_CFG_DONE  
#define XSA_EVENT_DATA_DONE  
#define XSA_EVENT_ERROR
```

Typedefs

```
typedef void(* XSysAce_EventHandler )(void *CallBackRef, int Event)
```

Functions

```
XStatus XSysAce_Initialize (XSysAce *InstancePtr, Xuint16 DeviceId)
```

```
XStatus XSysAce_Lock (XSysAce *InstancePtr, Xboolean Force)
```

```
void XSysAce_Unlock (XSysAce *InstancePtr)
```

```
Xuint32 XSysAce_GetErrors (XSysAce *InstancePtr)
```

```
XSysAce_Config * XSysAce_LookupConfig (Xuint16 DeviceId)
```

```
XStatus XSysAce_ResetCF (XSysAce *InstancePtr)
```

```
XStatus XSysAce_AbortCF (XSysAce *InstancePtr)
```

```
XStatus XSysAce_IdentifyCF (XSysAce *InstancePtr, XSysAce_CFPParameters
```

```
*ParamPtr)
```

Xboolean XSysAce_IsCFReady (**XSysAce** *InstancePtr)
XStatus XSysAce_SectorRead (**XSysAce** *InstancePtr, **Xuint32** StartSector, int NumSectors, **Xuint8** *BufferPtr)
XStatus XSysAce_SectorWrite (**XSysAce** *InstancePtr, **Xuint32** StartSector, int NumSectors, **Xuint8** *BufferPtr)
Xuint16 XSysAce_GetFatStatus (**XSysAce** *InstancePtr)
void **XSysAce_ResetCfg** (**XSysAce** *InstancePtr)
void **XSysAce_SetCfgAddr** (**XSysAce** *InstancePtr, unsigned int Address)
void **XSysAce_SetStartMode** (**XSysAce** *InstancePtr, **Xboolean** ImmedOnReset, **Xboolean** SetStart)
Xboolean XSysAce_IsCfgDone (**XSysAce** *InstancePtr)
Xuint32 XSysAce_GetCfgSector (**XSysAce** *InstancePtr)
XStatus XSysAce_ProgramChain (**XSysAce** *InstancePtr, **Xuint8** *BufferPtr, int NumBytes)
void **XSysAce_EnableInterrupt** (**XSysAce** *InstancePtr)
void **XSysAce_DisableInterrupt** (**XSysAce** *InstancePtr)
void **XSysAce_SetEventHandler** (**XSysAce** *InstancePtr, **XSysAce_EventHandler** FuncPtr, void *CallbackRef)
void **XSysAce_InterruptHandler** (void *InstancePtr)
XStatus XSysAce_SelfTest (**XSysAce** *InstancePtr)
Xuint16 XSysAce_GetVersion (**XSysAce** *InstancePtr)

Define Documentation

```
#define XSA_EVENT_CFG_DONE
```

Configuration of JTAG chain is done

```
#define XSA_EVENT_DATA_DONE
```

Data transfer to/from CompactFlash is done

```
#define XSA_EVENT_ERROR
```

An error occurred. Use **XSysAce_GetErrors**() to determine the cause of the error(s).

Typedef Documentation

```
typedef void(* XSysAce_EventHandler)(void *CallbackRef, int Event)
```

Callback when an asynchronous event occurs during interrupt mode.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked.

Event is the event that occurred. See [xsysace.h](#) and the event identifiers prefixed with XSA_EVENT_* for a description of possible events.

Function Documentation

```
XStatus XSysAce_AbortCF( XSysAce * InstancePtr)
```

Abort the CompactFlash operation currently in progress.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

- XST_SUCCESS if the abort was done successfully
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

According to the ASIC designer, the abort command has not been well tested.

```
void XSysAce_DisableInterrupt( XSysAce * InstancePtr)
```

Disable all System ACE interrupts and hold the interrupt request line of the device in reset.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance that just interrupted.

Returns:

None.

Note:

None.

```
void XSysAce_EnableInterrupt( XSysAce * InstancePtr)
```

Enable System ACE interrupts. There are three interrupts that can be enabled. The error interrupt enable serves as the driver's means to determine whether interrupts have been enabled or not. The configuration-done interrupt is not enabled here, instead it is enabled during a reset - which can cause a configuration process to start. The data-buffer-ready interrupt is not enabled here either. It is enabled when a read or write operation is started. The reason for not enabling the latter two interrupts are because the status bits may be set as a leftover of an earlier occurrence of the interrupt.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to work on.

Returns:

None.

Note:

None.

```
Xuint32 XSysAce_GetCfgSector( XSysAce * InstancePtr)
```

Get the sector ID of the CompactFlash sector being used for configuration of the target FPGA chain. This sector ID (or logical block address) only affects transfers between the ACE configuration logic and the CompactFlash card. This function is typically used for debug purposes to determine which sector was being accessed when an error occurred.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

The sector ID (logical block address) being used for data transfers between the ACE configuration logic and the CompactFlash. Sector IDs range from 0 to 0x10000000.

Note:

None.

Xuint32 XSysAce_GetErrors(XSysAce * InstancePtr)

Get all outstanding errors. Errors include the inability to read or write CompactFlash and the inability to successfully configure FPGA devices along the target FPGA chain.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

A 32-bit mask of error values. See [xsysace_1.h](#) for a description of possible values. The error identifiers are prefixed with XSA_ER_*.

Note:

None.

Xuint16 XSysAce_GetFatStatus(XSysAce * InstancePtr)

Get the status of the FAT filesystem on the first valid partition of the CompactFlash device such as the boot record and FAT types found.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

A 16-bit mask of status values. These values are defined in **xsysace_1.h** with the prefix XSA_FAT_*.

Note:

None.

```
Xuint16 XSysAce_GetVersion( XSysAce * InstancePtr )
```

Retrieve the version of the System ACE device.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

A 16-bit version where the 4 most significant bits are the major version number, the next four bits are the minor version number, and the least significant 8 bits are the revision or build number.

Note:

None.

```
XStatus XSysAce_IdentifyCF( XSysAce * InstancePtr,  
                           XSysAce_CFPParameters * ParamPtr  
                           )
```

Identify the CompactFlash device. Retrieves the parameters for the CompactFlash storage device. Note that this is a polled read of one sector of data. The data is read from the CompactFlash into a byte buffer, which is then copied into the [XSysAce_CFPParameters](#) structure passed in by the user. The copy is necessary since we don't know how the compiler packs the [XSysAce_CFPParameters](#) structure.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

ParamPtr is a pointer to a [XSysAce_CFPParameters](#) structure where the information for the CompactFlash device will be stored. See [xsysace.h](#) for details on the [XSysAce_CFPParameters](#) structure.

Returns:

- XST_SUCCESS if the identify was done successfully
- XST_FAILURE if an error occurs. Use [XSysAce_GetErrors\(\)](#) to determine cause.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

None.

```
XStatus XSysAce_Initialize( XSysAce * InstancePtr,  
                          Xuint16 DeviceId  
                          )
```

Initialize a specific [XSysAce](#) instance. The configuration information for the given device ID is found and the driver instance data is initialized appropriately.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

DeviceId is the unique id of the device controlled by this [XSysAce](#) instance.

Returns:

XST_SUCCESS if successful, or XST_DEVICE_NOT_FOUND if the device was not found in the configuration table in [xsysace_g.c](#).

Note:

We do not want to reset the configuration controller here since this could cause a reconfiguration of the JTAG target chain, depending on how the CFGMODEPIN of the device is wired.

void XSysAce_InterruptHandler(void * *InstancePtr*)

The interrupt handler for the System ACE driver. This handler must be connected by the user to an interrupt controller or source. This function does not save or restore context.

This function continues reading or writing to the compact flash if such an operation is in progress, and notifies the upper layer software through the event handler once the operation is complete or an error occurs. On an error, any command currently in progress is aborted.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance that just interrupted.

Returns:

None.

Note:

None.

Xboolean XSysAce_IsCfgDone(XSysAce * *InstancePtr*)

Check to see if the configuration of the target FPGA chain is complete. This function is typically only used in polled mode. In interrupt mode, an event (XSA_EVENT_CFG_DONE) is returned to the user in the asynchronous event handler.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

XTRUE if the configuration is complete. XFALSE otherwise.

Note:

None.

Xboolean **XSysAce_IsCFReady**(**XSysAce** * *InstancePtr*)

Check to see if the CompactFlash is ready for a command. The CompactFlash may delay after one operation before it is ready for the next. This function helps the user determine when it is ready before invoking a CompactFlash operation such as **XSysAce_SectorRead**() or **XSysAce_SectorWrite**();

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

XTRUE if the CompactFlash is ready for a command, and XFALSE otherwise.

Note:

None.

XStatus **XSysAce_Lock**(**XSysAce** * *InstancePtr*, **Xboolean** *Force*)

Attempt to lock access to the CompactFlash. The CompactFlash may be accessed by the MPU port as well as the JTAG configuration port within the System ACE device. This function requests exclusive access to the CompactFlash for the MPU port. This is a non-blocking request. If access cannot be locked (because the configuration controller has the lock), an appropriate status is returned. In this case, the user should call this function again until successful.

If the user requests a forced lock, the JTAG configuration controller will be put into a reset state in case it currently has a lock on the CompactFlash. This effectively aborts any operation the configuration controller had in progress and makes the configuration controller restart its process the next time it is able to get a lock.

A lock must be granted to the user before attempting to read or write the CompactFlash device.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Force is a boolean value that, when set to XTRUE, will force the MPU lock to occur in the System ACE. When set to XFALSE, the lock is requested and the device arbitrates between the MPU request and JTAG requests. Forcing the MPU lock resets the configuration controller, thus aborting any configuration operations in progress.

Returns:

XST_SUCCESS if the lock was granted, or XST_DEVICE_BUSY if the lock was not granted because the configuration controller currently has access to the CompactFlash.

Note:

If the lock is not granted to the MPU immediately, this function removes its request for a lock so that a lock is not later granted at a time when the application is (a) not ready for the lock, or (b) cannot be informed asynchronously about the granted lock since there is no such interrupt event.

XSysAce_Config* XSysAce_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. The table XSysAce_ConfigTable contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or XNULL if no match is found.

Note:

None.

```
XStatus XSysAce_ProgramChain( XSysAce * InstancePtr,  
                               Xuint8 * BufferPtr,  
                               int NumBytes  
                               )
```

Program the target FPGA chain through the configuration JTAG port. This allows the user to program the devices on the target FPGA chain from the MPU port instead of from CompactFlash. The user specifies a buffer and the number of bytes to write. The buffer should be equivalent to an ACE (.ace) file.

Note that when loading the ACE file via the MPU port, the first sector of the ACE file is discarded. The CF filesystem controller in the System ACE device knows to skip the first sector when the ACE file comes from the CF, but the CF filesystem controller is bypassed when the ACE file comes from the MPU port. For this reason, this function skips the first sector of the buffer passed in.

In polled mode, the write is blocking. In interrupt mode, the write is non-blocking and an event, XSA_EVENT_CFG_DONE, is returned to the user in the asynchronous event handler when the configuration is complete.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

BufferPtr is a pointer to a buffer that will be used to program the configuration JTAG devices.

NumBytes is the number of bytes in the buffer. We assume that there is at least one sector of data in the .ace file, which is the information sector.

Returns:

- XST_SUCCESS if the write was successful. In interrupt mode, this does not mean the write is complete, only that it has begun. An event is returned to the user when the write is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_FAILURE if an error occurred during the write. The user should call [XSysAce_GetErrors\(\)](#) to determine the cause of the error.

Note:

None.

```
XStatus XSysAce_ResetCF( XSysAce * InstancePtr)
```

Reset the CompactFlash device. This function does not reset the System ACE controller. An ATA soft-reset of the CompactFlash is performed.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

- XST_SUCCESS if the reset was done successfully
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

None.

```
void XSysAce_ResetCfg( XSysAce * InstancePtr)
```

Reset the JTAG configuration controller. This comprises a reset of the JTAG configuration controller and the CompactFlash controller (if it is currently being accessed by the configuration controller). Note that the MPU controller is not reset, meaning the MPU registers remain unchanged. The configuration controller is reset then released from reset in this function.

The CFGDONE status (and therefore interrupt) is cleared when the configuration controller is reset. If interrupts have been enabled, we go ahead and enable the CFGDONE interrupt here. This means that if and when a configuration process starts as a result of this reset, an interrupt will be received when it is complete.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

None.

Note:

This function is not thread-safe.

```
XStatus XSysAce_SectorRead( XSysAce * InstancePtr,  
                             Xuint32   StartSector,  
                             int       NumSectors,  
                             Xuint8 * BufferPtr  
                             )
```

Read at least one sector of data from the CompactFlash. The user specifies the starting sector ID and the number of sectors to be read. The minimum unit that can be read from the CompactFlash is a sector, which is 512 bytes.

In polled mode, this read is blocking. If there are other tasks in the system that must run, it is best to keep the number of sectors to be read to a minimum (e.g., 1). In interrupt mode, this read is non-blocking and an event, XSA_EVENT_DATA_DONE, is returned to the user in the asynchronous event handler when the read is complete. The user must call **XSysAce_EnableInterrupt()** to put the driver/device into interrupt mode.

An MPU lock, obtained using **XSysAce_Lock()**, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

StartSector is the starting sector ID from where data will be read. Sector IDs range from 0 (first sector) to 0x10000000.

NumSectors is the number of sectors to read. The range can be from 1 to 256.

BufferPtr is a pointer to a buffer where the data will be stored. The user must ensure it is big enough to hold (512 * NumSectors) bytes.

Returns:

- XST_SUCCESS if the read was successful. In interrupt mode, this does not mean the read is complete, only that it has begun. An event is returned to the user when the read is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the ACE controller is not ready for a command
- XST_FAILURE if an error occurred during the read. The user should call **XSysAce_GetErrors()** to determine the cause of the error.

Note:

None.


```
XStatus XSysAce_SectorWrite( XSysAce * InstancePtr,  
                             Xuint32   StartSector,  
                             int       NumSectors,  
                             Xuint8 * BufferPtr  
                             )
```

Write data to the CompactFlash. The user specifies the starting sector ID and the number of sectors to be written. The minimum unit that can be written to the CompactFlash is a sector, which is 512 bytes.

In polled mode, this write is blocking. If there are other tasks in the system that must run, it is best to keep the number of sectors to be written to a minimum (e.g., 1). In interrupt mode, this write is non-blocking and an event, XSA_EVENT_DATA_DONE, is returned to the user in the asynchronous event handler when the write is complete. The user must call **XSysAce_EnableInterrupt()** to put the driver/device into interrupt mode.

An MPU lock, obtained using **XSysAce_Lock()**, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

StartSector is the starting sector ID from where data will be written. Sector IDs range from 0 (first sector) to 0x10000000.

NumSectors is the number of sectors to write. The range can be from 1 to 256.

BufferPtr is a pointer to the data buffer to be written. This buffer must have at least (512 * NumSectors) bytes.

Returns:

- XST_SUCCESS if the write was successful. In interrupt mode, this does not mean the write is complete, only that it has begun. An event is returned to the user when the write is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the ACE controller is not ready for a command
- XST_FAILURE if an error occurred during the write. The user should call **XSysAce_GetErrors()** to determine the cause of the error.

Note:

None.

```
XStatus XSysAce_SelfTest( XSysAce * InstancePtr)
```

A self-test that simply proves communication to the ACE controller from the device driver by obtaining an MPU lock, verifying it, then releasing it.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

XST_SUCCESS if self-test passes, or XST_FAILURE if an error occurs.

Note:

None.

```
void XSysAce_SetCfgAddr( XSysAce * InstancePtr,  
                        unsigned int Address  
                        )
```

Select the configuration address (or file) from the CompactFlash to be used for configuration of the target FPGA chain.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Address is the address or file number to be used as the bitstream to configure the target FPGA devices. There are 8 possible files, so the value of this parameter can range from 0 to 7.

Returns:

None.

Note:

None.

```
void XSysAce_SetEventHandler( XSysAce * InstancePtr,  
                             XSysAce_EventHandler FuncPtr,  
                             void * CallbackRef  
                             )
```

Set the callback function for handling events. The upper layer software should call this function during initialization. The events are passed asynchronously to the upper layer software. The events are described in [xsysace.h](#) and are named XSA_EVENT_*.

Note that the callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

FuncPtr is the pointer to the callback function.

CallBackRef is a reference pointer to be passed back to the upper layer.

Returns:

None.

Note:

None.

```
void XSysAce_SetStartMode( XSysAce * InstancePtr,  
                           Xboolean ImmedOnReset,  
                           Xboolean StartCfg  
                           )
```

Set the start mode for configuration of the target FPGA chain from CompactFlash. The configuration process only starts after a reset. The user can indicate that the configuration should start immediately after a reset, or the configuration process can be delayed until the user commands it to start (using this function). The configuration controller can be reset using [XSysAce_ResetCfg\(\)](#).

The user can select which configuration file on the CompactFlash to use using the [XSysAce_SetCfgAddr\(\)](#) function. If the user intends to configure the target FPGA chain directly from the MPU port, this function is not needed. Instead, the user would simply call [XSysAce_ProgramChain\(\)](#).

The user can use [XSysAce_IsCfgDone\(\)](#) when in polled mode to determine if the configuration is complete. If in interrupt mode, the event XSA_EVENT_CFG_DONE will be returned asynchronously to the user when the configuration is complete. The user must call [XSysAce_EnableInterrupt\(\)](#) to put the device/driver into interrupt mode.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

ImmedOnReset can be set to XTRUE to indicate the configuration process will start immediately after a reset of the ACE configuration controller, or it can be set to XFALSE to indicate the configuration process is delayed after a reset until the user starts it (using this function).

StartCfg is a boolean indicating whether to start the configuration process or not. When ImmedOnReset is set to XTRUE, this value is ignored. When ImmedOnReset is set to XFALSE, then this value controls when the configuration process is started. When set to XTRUE the configuration process starts (assuming a reset of the device has occurred), and when set to XFALSE the configuration process does not start.

Returns:

None.

Note:

None.

```
void XSysAce_Unlock( XSysAce * InstancePtr)
```

Release the MPU lock to the CompactFlash. If a lock is not currently granted to the MPU port, this function has no effect.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

None.

Note:

None.

sysace/v1_00_a/src/xsysace.c File Reference

Detailed Description

The Xilinx System ACE driver component. This driver supports the Xilinx System Advanced Configuration Environment (ACE) controller. It currently supports only the CompactFlash solution. See [xsysace.h](#) for a detailed description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	06/17/02	work in progress
1.00a	rmm	05/14/03	Fixed diab compiler warnings relating to asserts

```
#include "xparameters.h"  
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

XStatus **XSysAce_Initialize** (**XSysAce** *InstancePtr, **Xuint16** DeviceId)

XStatus **XSysAce_Lock** (**XSysAce** *InstancePtr, **Xboolean** Force)

void **XSysAce_Unlock** (**XSysAce** *InstancePtr)

Xuint32 **XSysAce_GetErrors** (**XSysAce** *InstancePtr)

XSysAce_Config * **XSysAce_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
Xuint32 XSysAce_GetErrors( XSysAce * InstancePtr )
```

Get all outstanding errors. Errors include the inability to read or write CompactFlash and the inability to successfully configure FPGA devices along the target FPGA chain.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

A 32-bit mask of error values. See [xsysace_1.h](#) for a description of possible values. The error identifiers are prefixed with XSA_ER_*.

Note:

None.

```
XStatus XSysAce_Initialize( XSysAce * InstancePtr,  
                           Xuint16   DeviceId  
                           )
```

Initialize a specific **XSysAce** instance. The configuration information for the given device ID is found and the driver instance data is initialized appropriately.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XSysAce** instance.

Returns:

XST_SUCCESS if successful, or XST_DEVICE_NOT_FOUND if the device was not found in the configuration table in [xsysace_g.c](#).

Note:

We do not want to reset the configuration controller here since this could cause a reconfiguration of the JTAG target chain, depending on how the CFGMODEPIN of the device is wired.

```
XStatus XSysAce_Lock( XSysAce * InstancePtr,  
                    Xboolean Force  
                    )
```

Attempt to lock access to the CompactFlash. The CompactFlash may be accessed by the MPU port as well as the JTAG configuration port within the System ACE device. This function requests exclusive access to the CompactFlash for the MPU port. This is a non-blocking request. If access cannot be locked (because the configuration controller has the lock), an appropriate status is returned. In this case, the user should call this function again until successful.

If the user requests a forced lock, the JTAG configuration controller will be put into a reset state in case it currently has a lock on the CompactFlash. This effectively aborts any operation the configuration controller had in progress and makes the configuration controller restart its process the next time it is able to get a lock.

A lock must be granted to the user before attempting to read or write the CompactFlash device.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Force is a boolean value that, when set to XTRUE, will force the MPU lock to occur in the System ACE. When set to XFALSE, the lock is requested and the device arbitrates between the MPU request and JTAG requests. Forcing the MPU lock resets the configuration controller, thus aborting any configuration operations in progress.

Returns:

XST_SUCCESS if the lock was granted, or XST_DEVICE_BUSY if the lock was not granted because the configuration controller currently has access to the CompactFlash.

Note:

If the lock is not granted to the MPU immediately, this function removes its request for a lock so that a lock is not later granted at a time when the application is (a) not ready for the lock, or (b) cannot be informed asynchronously about the granted lock since there is no such interrupt event.

```
XSysAce_Config* XSysAce_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID. The table `XSysAce_ConfigTable` contains the configuration info for each device in the system.

Parameters:

DeviceId is the unique device ID to look for.

Returns:

A pointer to the configuration data for the device, or `XNULL` if no match is found.

Note:

None.

void XSysAce_Unlock(XSysAce * InstancePtr)

Release the MPU lock to the CompactFlash. If a lock is not currently granted to the MPU port, this function has no effect.

Parameters:

InstancePtr is a pointer to the `XSysAce` instance to be worked on.

Returns:

None.

Note:

None.

sysace/v1_00_a/src/xsysace_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xsysace_i.h,v 1.13 2002/10/24 20:51:00 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file sysace/v1_00_a/src/xsysace_i.h
00026 *
00027 * Defines identifiers and low-level macros/functions for the XSysAce driver.
00028 * These identifiers include register offsets and bit masks. A high-level driver
00029 * interface is defined in xsysace.h.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date        Changes
00035 * -----
00036 * 1.00a rpm    06/14/02    work in progress
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XSYSACE_L_H /* prevent circular inclusions */
00042 #define XSYSACE_L_H /* by using protection macros */
```

```

00043
00044 /***** Include Files *****/
00045
00046 #include "xbasic_types.h"
00047 #include "xio.h"
00048
00049 /***** Constant Definitions *****/
00050
00051 /*
00052  * Constant used to align the register offsets to the proper address. This was
00053  * used during development to handle both byte-addressable (alignment=1) and
00054  * word addressable (alignment=4) registers. The #ifndef allows the user to
00055  * modify this at compile-time.
00056  */
00057 #ifndef XSA_ADDR_ALIGN
00058 #define XSA_ADDR_ALIGN      1
00059 #endif
00060
00061 /** @name Register Offsets
00062  * System ACE register offsets
00063  * @{
00064  */
00065 #define XSA_BMR_OFFSET      (XSA_ADDR_ALIGN * 0)  /**< Bus mode (BUSMODEREG) */
00066 #define XSA_SR_OFFSET       (XSA_ADDR_ALIGN * 4)  /**< Status (STATUSREG) */
00067 #define XSA_ER_OFFSET       (XSA_ADDR_ALIGN * 8)  /**< Error (ERRORREG) */
00068 #define XSA_CLR_OFFSET      (XSA_ADDR_ALIGN * 12) /**< Config LBA (CFGLBAREG) */
00069 #define XSA_MLR_OFFSET      (XSA_ADDR_ALIGN * 16) /**< MPU LBA (MPULBAREG) */
00070 #define XSA_SCCR_OFFSET     (XSA_ADDR_ALIGN * 20) /**< Sector cnt (SECCNTCMDREG)
*/
00071 #define XSA_VR_OFFSET       (XSA_ADDR_ALIGN * 22) /**< Version (VERSIONREG) */
00072 #define XSA_CR_OFFSET       (XSA_ADDR_ALIGN * 24) /**< Control (CONTROLREG) */
00073 #define XSA_FSR_OFFSET      (XSA_ADDR_ALIGN * 28) /**< FAT status (FATSTATREG) */
00074 #define XSA_DBR_OFFSET      (XSA_ADDR_ALIGN * 64) /**< Data buffer (DATABUFREG) */
00075 /*@}*/
00076
00077 /*
00078  * Bus Mode Register masks
00079  */
00080 #define XSA_BMR_16BIT_MASK  0x0101  /**< 16-bit access to ACE controller */
00081
00082
00083 /** @name Status Values
00084  * Status Register masks
00085  * @{
00086  */
00087 #define XSA_SR_CFGLOCK_MASK 0x00000001  /**< Config port lock status */
00088 #define XSA_SR_MPULOCK_MASK 0x00000002  /**< MPU port lock status */
00089 #define XSA_SR_CFGERROR_MASK 0x00000004  /**< Config port error status */
00090 #define XSA_SR_CFCERROR_MASK 0x00000008  /**< CF error status */
00091 #define XSA_SR_CFDTECT_MASK 0x00000010  /**< CF detect flag */

```

```

00092 #define XSA_SR_DATABUFRDY_MASK 0x00000020 /**< Data buffer ready status */
00093 #define XSA_SR_DATABUFMODE_MASK 0x00000040 /**< Data buffer mode status */
00094 #define XSA_SR_CFGDONE_MASK 0x00000080 /**< Configuration done status */
00095 #define XSA_SR_RDYFORCMD_MASK 0x00000100 /**< Ready for CF command */
00096 #define XSA_SR_CFGMODE_MASK 0x00000200 /**< Configuration mode status */
00097 #define XSA_SR_CFGADDR_MASK 0x0000E000 /**< Configuration address */
00098 #define XSA_SR_CFBSY_MASK 0x00020000 /**< CF busy (BSY bit) */
00099 #define XSA_SR_CFRDY_MASK 0x00040000 /**< CF ready (RDY bit) */
00100 #define XSA_SR_CFDWF_MASK 0x00080000 /**< CF data write fault (DWF bit)
*/
00101 #define XSA_SR_CFDSC_MASK 0x00100000 /**< CF ready (DSC bit) */
00102 #define XSA_SR_CFDRQ_MASK 0x00200000 /**< CF data request (DRQ) */
00103 #define XSA_SR_CFCORR_MASK 0x00400000 /**< CF correctable error (CORR
bit) */
00104 #define XSA_SR_CFERR_MASK 0x00800000 /**< CF error (ERR bit) */
00105 /*@}*/
00106
00107
00108 /** @name Error Values
00109 * Error Register masks.
00110 * @{
00111 */
00112 #define XSA_ER_CARD_RESET 0x00000001 /**< CF card failed to reset */
00113 #define XSA_ER_CARD_READY 0x00000002 /**< CF card failed to ready */
00114 #define XSA_ER_CARD_READ 0x00000004 /**< CF read command failed */
00115 #define XSA_ER_CARD_WRITE 0x00000008 /**< CF write command failed */
00116 #define XSA_ER_SECTOR_READY 0x00000010 /**< CF sector failed to ready */
00117 #define XSA_ER_CFG_ADDR 0x00000020 /**< Cfg address is invalid */
00118 #define XSA_ER_CFG_FAIL 0x00000040 /**< Failed to configure a device */
00119 #define XSA_ER_CFG_READ 0x00000080 /**< Cfg read of CF failed */
00120 #define XSA_ER_CFG_INSTR 0x00000100 /**< Invalid instruction during cfg */
00121 #define XSA_ER_CFG_INIT 0x00000200 /**< CFGINIT pin error - did not
00122 * go high within 500ms of start */
00123 #define XSA_ER_RESERVED 0x00000400 /**< reserved */
00124 #define XSA_ER_BAD_BLOCK 0x00000800 /**< CF bad block detected */
00125 #define XSA_ER_UNCORRECTABLE 0x00001000 /**< CF uncorrectable error */
00126 #define XSA_ER_SECTOR_ID 0x00002000 /**< CF sector ID not found */
00127 #define XSA_ER_ABORT 0x00004000 /**< CF command aborted */
00128 #define XSA_ER_GENERAL 0x00008000 /**< CF general error */
00129 /*@}*/
00130
00131
00132 /**
00133 * Config LBA Register - address mask
00134 */
00135 #define XSA_CLR_LBA_MASK 0xFFFFFFFF /* Logical Block Address mask */
00136
00137 /**
00138 * MPU LBA Register - address mask

```

```

00139 */
00140 #define XSA_MLR_LBA_MASK      0xFFFFFFFF /* Logical Block Address mask */
00141
00142
00143 /** @name Sector Count/Command Values
00144 * Sector Count Command Register masks
00145 * @{
00146 */
00147 #define XSA_SCCR_COUNT_MASK    0x00FF /*< Sector count mask */
00148 #define XSA_SCCR_RESET_MASK   0x0100 /*< Reset CF card command */
00149 #define XSA_SCCR_IDENTIFY_MASK 0x0200 /*< Identify CF card command */
00150 #define XSA_SCCR_READDATA_MASK 0x0300 /*< Read CF card command */
00151 #define XSA_SCCR_WRITEDATA_MASK 0x0400 /*< Write CF card command */
00152 #define XSA_SCCR_ABORT_MASK   0x0600 /*< Abort CF command */
00153 #define XSA_SCCR_CMD_MASK     0x0700 /*< Command mask */
00154 /*@}*/
00155
00156
00157 /*
00158 * Version Register masks
00159 */
00160 #define XSA_VR_BUILD_MASK     0x00FF /* Revision/build number */
00161 #define XSA_VR_MINOR_MASK     0x0F00 /* Minor version number */
00162 #define XSA_VR_MAJOR_MASK     0xF000 /* Major version number */
00163
00164
00165 /** @name Control Values
00166 * Control Register masks
00167 * @{
00168 */
00169 #define XSA_CR_FORCELOCK_MASK 0x00000001 /*< Force lock request */
00170 #define XSA_CR_LOCKREQ_MASK   0x00000002 /*< MPU lock request */
00171 #define XSA_CR_FORCECFGADDR_MASK 0x00000004 /*< Force CFG address */
00172 #define XSA_CR_FORCECFGMODE_MASK 0x00000008 /*< Force CFG mode */
00173 #define XSA_CR_CFGMODE_MASK   0x00000010 /*< CFG mode */
00174 #define XSA_CR_CFGSTART_MASK  0x00000020 /*< CFG start */
00175 #define XSA_CR_CFGSEL_MASK     0x00000040 /*< CFG select */
00176 #define XSA_CR_CFGRESET_MASK  0x00000080 /*< CFG reset */
00177 #define XSA_CR_DATARDYIRQ_MASK 0x00000100 /*< Enable data ready IRQ */
00178 #define XSA_CR_ERRORIRQ_MASK  0x00000200 /*< Enable error IRQ */
00179 #define XSA_CR_CFGDONEIRQ_MASK 0x00000400 /*< Enable CFG done IRQ */
00180 #define XSA_CR_RESETIRQ_MASK  0x00000800 /*< Reset IRQ line */
00181 #define XSA_CR_CFGPROG_MASK    0x00001000 /*< Inverted CFGPROG pin */
00182 #define XSA_CR_CFGADDR_MASK    0x0000E000 /*< Config address mask */
00183 #define XSA_CR_CFGADDR_SHIFT  13 /*< Config address shift */
00184 /*@}*/
00185
00186
00187 /** @name FAT Status
00188 *

```

```
00189 * FAT filesystem status masks. The first valid partition of the CF
00190 * is a FAT partition.
00191 * @{
00192 */
00193 #define XSA_FAT_VALID_BOOT_REC 0x0001 /**< Valid master boot record */
00194 #define XSA_FAT_VALID_PART_REC 0x0002 /**< Valid partition boot record */
00195 #define XSA_FAT_12_BOOT_REC 0x0004 /**< FAT12 in master boot rec */
00196 #define XSA_FAT_12_PART_REC 0x0008 /**< FAT12 in partition boot rec */
00197 #define XSA_FAT_16_BOOT_REC 0x0010 /**< FAT16 in master boot rec */
00198 #define XSA_FAT_16_PART_REC 0x0020 /**< FAT16 in partition boot rec */
00199 #define XSA_FAT_12_CALC 0x0040 /**< Calculated FAT12 from clusters */
00200 #define XSA_FAT_16_CALC 0x0080 /**< Calculated FAT16 from clusters */
00201 /*@}*/
00202
00203
00204 #define XSA_DATA_BUFFER_SIZE 32 /**< Size of System ACE data buffer */
00205 #define XSA_CF_SECTOR_SIZE 512 /**< Number of bytes in a CF sector */
00206
00207 /***** Type Definitions *****/
00208
00209
00210 /***** Macros (Inline Functions) Definitions *****/
00211
00212 /*****
00213 *
00214 * Low-level driver macros and functions. The list below provides signatures
00215 * to help the user use the macros.
00216 *
00217 * Xuint32 XSysAce_mGetControlReg(Xuint32 BaseAddress)
00218 * void XSysAce_mSetControlReg(Xuint32 BaseAddress, Xuint32 Data)
00219 * void XSysAce_mOrControlReg(Xuint32 BaseAddress, Xuint32 Data)
00220 * void XSysAce_mAndControlReg(Xuint32 BaseAddress, Xuint32 Data)
00221 * Xuint32 XSysAce_mGetErrorReg(Xuint32 BaseAddress)
00222 * Xuint32 XSysAce_mGetStatusReg(Xuint32 BaseAddress)
00223 *
00224 * void XSysAce_mSetCfgAddr(Xuint32 BaseAddress, unsigned int Address)
00225 * void XSysAce_mWaitForLock(Xuint32 BaseAddress)
00226 * void XSysAce_mEnableIntr(Xuint32 BaseAddress, Xuint32 Mask)
00227 * void XSysAce_mDisableIntr(Xuint32 BaseAddress, Xuint32 Mask)
00228 *
00229 * Xboolean XSysAce_mIsReadyForCmd(Xuint32 BaseAddress)
00230 * Xboolean XSysAce_mIsCfgDone(Xuint32 BaseAddress)
00231 * Xboolean XSysAce_mIsMpuLocked(Xuint32 BaseAddress)
00232 * Xboolean XSysAce_mIsIntrEnabled(Xuint32 BaseAddress)
00233 *
00234 *****/
00235
00236
00237 /*****
00238 /**
00239 *
```

```

00240 * Get the contents of the control register.
00241 *
00242 * @param    BaseAddress is the base address of the device.
00243 *
00244 * @return    The 32-bit value of the control register.
00245 *
00246 * @note      None.
00247 *
00248 *****/
00249 #define XSysAce_mGetControlReg(BaseAddress) \
00250         XSysAce_RegRead32((BaseAddress) + XSA_CR_OFFSET)
00251
00252
00253 *****/
00254 /**
00255 *
00256 * Set the contents of the control register.
00257 *
00258 * @param    BaseAddress is the base address of the device.
00259 * @param    Data is the 32-bit value to write to the register.
00260 *
00261 * @return    None.
00262 *
00263 * @note      None.
00264 *
00265 *****/
00266 #define XSysAce_mSetControlReg(BaseAddress, Data) \
00267         XSysAce_RegWrite32((BaseAddress) + XSA_CR_OFFSET, (Data))
00268
00269
00270 *****/
00271 /**
00272 *
00273 * Set the contents of the control register to the value specified OR'ed with
00274 * its current contents.
00275 *
00276 * @param    BaseAddress is the base address of the device.
00277 * @param    Data is the 32-bit value to OR with the register.
00278 *
00279 * @return    None.
00280 *
00281 * @note      None.
00282 *
00283 *****/
00284 #define XSysAce_mOrControlReg(BaseAddress, Data) \
00285         XSysAce_mSetControlReg((BaseAddress), \
00286         XSysAce_mGetControlReg(BaseAddress) | (Data))
00287
00288
00289 *****/
00290 /**
00291 *

```

```
00292 * Set the contents of the control register to the value specified AND'ed with
00293 * its current contents.
00294 *
00295 * @param BaseAddress is the base address of the device.
00296 * @param Data is the 32-bit value to AND with the register.
00297 *
00298 * @return None.
00299 *
00300 * @note None.
00301 *
00302 *****/
00303 #define XSysAce_mAndControlReg(BaseAddress, Data) \
00304     XSysAce_mSetControlReg((BaseAddress), \
00305     XSysAce_mGetControlReg(BaseAddress) & (Data))
00306
00307
00308 *****/
00309 /**
00310 *
00311 * Get the contents of the error register.
00312 *
00313 * @param BaseAddress is the base address of the device.
00314 *
00315 * @return The 32-bit value of the register.
00316 *
00317 * @note None.
00318 *
00319 *****/
00320 #define XSysAce_mGetErrorReg(BaseAddress) \
00321     XSysAce_RegRead32((BaseAddress) + XSA_ER_OFFSET)
00322
00323
00324 *****/
00325 /**
00326 *
00327 * Get the contents of the status register.
00328 *
00329 * @param BaseAddress is the base address of the device.
00330 *
00331 * @return The 32-bit value of the register.
00332 *
00333 * @note None.
00334 *
00335 *****/
00336 #define XSysAce_mGetStatusReg(BaseAddress) \
00337     XSysAce_RegRead32((BaseAddress) + XSA_SR_OFFSET)
00338
00339
00340 *****/
00341 /**
00342 *
00343 * Set the configuration address, or file, of the CompactFlash. This address
```

```

00344 * indicates which .ace bitstream to use to configure the target FPGA chain.
00345 *
00346 * @param BaseAddress is the base address of the device.
00347 * @param Address ranges from 0 to 7 and represents the eight possible .ace
00348 * bitstreams that can reside on the CompactFlash.
00349 *
00350 * @return None.
00351 *
00352 * @note Used cryptic var names to avoid conflict with caller's var names.
00353 *
00354 *****/
00355 #define XSysAce_mSetCfgAddr(BaseAddress, Address) \
00356 { \
00357     Xuint32 A66rMask = ((Address) << XSA_CR_CFGADDR_SHIFT) & \
XSA_CR_CFGADDR_MASK; \
00358     Xuint32 C0ntr01 = XSysAce_mGetControlReg(BaseAddress); \
00359     C0ntr01 &= ~XSA_CR_CFGADDR_MASK; /* clear current address */ \
00360     C0ntr01 |= (A66rMask | XSA_CR_FORCECFGADDR_MASK); \
00361     XSysAce_mSetControlReg((BaseAddress), C0ntr01); \
00362 }
00363
00364
00365 *****/
00366 /**
00367 *
00368 * Request then wait for the MPU lock. This is not a forced lock, so we must
00369 * contend with the configuration controller.
00370 *
00371 * @param BaseAddress is the base address of the device.
00372 *
00373 * @return None.
00374 *
00375 * @note None.
00376 *
00377 *****/
00378 #define XSysAce_mWaitForLock(BaseAddress) \
00379 { \
00380     XSysAce_mOrControlReg((BaseAddress), XSA_CR_LOCKREQ_MASK); \
00381     while ((XSysAce_mGetStatusReg(BaseAddress) & XSA_SR_MPULOCK_MASK) == 0); \
00382 }
00383
00384 *****/
00385 /**
00386 *
00387 * Enable ACE controller interrupts.
00388 *
00389 * @param BaseAddress is the base address of the device.
00390 *
00391 * @return None.
00392 *
00393 * @note None.
00394 *

```



```
00395 *****/
00396 #define XSysAce_mEnableIntr(BaseAddress, Mask) \
00397     XSysAce_mOrControlReg((BaseAddress), (Mask));
00398
00399
00400 *****/
00401 /**
00402 *
00403 * Disable ACE controller interrupts.
00404 *
00405 * @param   BaseAddress is the base address of the device.
00406 *
00407 * @return  None.
00408 *
00409 * @note   None.
00410 *
00411 *****/
00412 #define XSysAce_mDisableIntr(BaseAddress, Mask) \
00413     XSysAce_mAndControlReg((BaseAddress), ~(Mask));
00414
00415
00416 *****/
00417 /**
00418 *
00419 * Is the CompactFlash ready for a command?
00420 *
00421 * @param   BaseAddress is the base address of the device.
00422 *
00423 * @return  XTRUE if it is ready, XFALSE otherwise.
00424 *
00425 * @note   None.
00426 *
00427 *****/
00428 #define XSysAce_mIsReadyForCmd(BaseAddress) \
00429     (XSysAce_mGetStatusReg(BaseAddress) & XSA_SR_RDYFORCMD_MASK)
00430
00431
00432 *****/
00433 /**
00434 *
00435 * Is the ACE controller locked for MPU access?
00436 *
00437 * @param   BaseAddress is the base address of the device.
00438 *
00439 * @return  XTRUE if it is locked, XFALSE otherwise.
00440 *
00441 * @note   None.
00442 *
00443 *****/
00444 #define XSysAce_mIsMpuLocked(BaseAddress) \
00445     (XSysAce_mGetStatusReg(BaseAddress) & XSA_SR_MPULOCK_MASK)
00446
```

```

00447
00448 /*****
00449 /**
00450 *
00451 * Is the CompactFlash configuration of the target FPGA chain complete?
00452 *
00453 * @param   BaseAddress is the base address of the device.
00454 *
00455 * @return   XTRUE if it is ready, XFALSE otherwise.
00456 *
00457 * @note     None.
00458 *
00459 *****/
00460 #define XSysAce_mIsCfgDone(BaseAddress) \
00461     (XSysAce_mGetStatusReg(BaseAddress) & XSA_SR_CFGDONE_MASK)
00462
00463
00464 /*****
00465 /**
00466 *
00467 * Have interrupts been enabled by the user? We look for the interrupt reset
00468 * bit to be clear (meaning interrupts are armed, even though none may be
00469 * individually enabled).
00470 *
00471 * @param   BaseAddress is the base address of the device.
00472 *
00473 * @return   XTRUE if it is enabled, XFALSE otherwise.
00474 *
00475 * @note     None.
00476 *
00477 *****/
00478 #define XSysAce_mIsIntrEnabled(BaseAddress) \
00479     ((XSysAce_mGetControlReg(BaseAddress) & XSA_CR_RESETIRQ_MASK) == 0)
00480
00481
00482 /***** Function Prototypes *****/
00483
00484 int XSysAce_ReadSector(Xuint32 BaseAddress, Xuint32 SectorId,
00485                       Xuint8 *BufferPtr);
00486 int XSysAce_WriteSector(Xuint32 BaseAddress, Xuint32 SectorId,
00487                        Xuint8 *BufferPtr);
00488
00489 /*
00490 * Utility functions to read and write registers and data buffer
00491 */
00492 Xuint32 XSysAce_RegRead32(Xuint32 Address);
00493 Xuint16 XSysAce_RegRead16(Xuint32 Address);
00494 void XSysAce_RegWrite32(Xuint32 Address, Xuint32 Data);
00495 void XSysAce_RegWrite16(Xuint32 Address, Xuint16 Data);
00496
00497 int XSysAce_ReadDataBuffer(Xuint32 BaseAddress, Xuint8 *BufferPtr,

```

```
00498             int NumBytes);
00499 int XSysAce_WriteDataBuffer(Xuint32 BaseAddress, Xuint8 *BufferPtr,
00500             int NumBytes);
00501
00502 #endif             /* end of protection macro */
00503
```

Generated on 29 May 2003 for Xilinx Device Drivers

sysace/v1_00_a/src/xsysace_l.c File Reference

Detailed Description

This file contains low-level functions to read and write CompactFlash sectors and ACE controller registers. These functions can be used when only the low-level functionality of the driver is desired. The user would typically use the high-level driver functions defined in [xsysace.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	06/14/02	work in progress

```
#include "xsysace\_l.h"
```

Functions

[Xuint32 XSysAce_RegRead32](#) ([Xuint32](#) Address)

[Xuint16 XSysAce_RegRead16](#) ([Xuint32](#) Address)

void [XSysAce_RegWrite32](#) ([Xuint32](#) Address, [Xuint32](#) Data)

void [XSysAce_RegWrite16](#) ([Xuint32](#) Address, [Xuint16](#) Data)

int [XSysAce_ReadSector](#) ([Xuint32](#) BaseAddress, [Xuint32](#) SectorId, [Xuint8](#) *BufferPtr)

int [XSysAce_WriteSector](#) ([Xuint32](#) BaseAddress, [Xuint32](#) SectorId, [Xuint8](#) *BufferPtr)

int [XSysAce_ReadDataBuffer](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *BufferPtr, int Size)

int [XSysAce_WriteDataBuffer](#) ([Xuint32](#) BaseAddress, [Xuint8](#) *BufferPtr, int Size)

Function Documentation

```
int XSysAce_ReadDataBuffer( Xuint32 BaseAddress,  
                           Xuint8 * BufferPtr,  
                           int      Size  
                           )
```

Read the specified number of bytes from the data buffer of the ACE controller. The data buffer, which is 32 bytes, can only be read two bytes at a time. Once the data buffer is read, we wait for it to be filled again before reading the next buffer's worth of data.

Parameters:

BaseAddress is the base address of the device

BufferPtr is a pointer to a buffer in which to store data.

Size is the number of bytes to read

Returns:

The total number of bytes read, or 0 if an error occurred.

Note:

If *Size* is not aligned with the size of the data buffer (32 bytes), this function will read the entire data buffer, dropping the extra bytes on the floor since the user did not request them. This is necessary to get the data buffer to be ready again.

```
int XSysAce_ReadSector( Xuint32 BaseAddress,  
                       Xuint32 SectorId,  
                       Xuint8 * BufferPtr  
                       )
```

Read a CompactFlash sector. This is a blocking, low-level function which does not return until the specified sector is read.

Parameters:

BaseAddress is the base address of the device

SectorId is the id of the sector to read

BufferPtr is a pointer to a buffer where the data will be stored.

Returns:

The number of bytes read. If this number is not equal to the sector size, 512 bytes, then an error occurred.

Note:

None.

Xuint16 XSysAce_RegRead16(**Xuint32** *Address*)

Read a 16-bit value from the given address. Based on a compile-time constant, do the read in one 16-bit read or two 8-bit reads.

Parameters:

Address is the address to read from.

Returns:

The 16-bit value of the address.

Note:

No need for endian conversion in 8-bit mode since this function gets the bytes into their proper lanes in the 16-bit word.

Xuint32 XSysAce_RegRead32(**Xuint32** *Address*)

Read a 32-bit value from the given address. Based on a compile-time constant, do the read in two 16-bit reads or four 8-bit reads.

Parameters:

Address is the address to read from.

Returns:

The 32-bit value of the address.

Note:

No need for endian conversion in 8-bit mode since this function gets the bytes into their proper lanes in the 32-bit word.

```
void XSysAce_RegWrite16( Xuint32 Address,  
                        Xuint16 Data  
                        )
```

Write a 16-bit value to the given address. Based on a compile-time constant, do the write in one 16-bit write or two 8-bit writes.

Parameters:

Address is the address to write to.

Data is the value to write

Returns:

None.

Note:

No need for endian conversion in 8-bit mode since this function writes the bytes into their proper lanes based on address.

```
void XSysAce_RegWrite32( Xuint32 Address,  
                       Xuint32 Data  
                       )
```

Write a 32-bit value to the given address. Based on a compile-time constant, do the write in two 16-bit writes or four 8-bit writes.

Parameters:

Address is the address to write to.

Data is the value to write

Returns:

None.

Note:

No need for endian conversion in 8-bit mode since this function writes the bytes into their proper lanes based on address.

```
int XSysAce_WriteDataBuffer( Xuint32 BaseAddress,  
                             Xuint8 * BufferPtr,  
                             int      Size  
                             )
```

Write the specified number of bytes to the data buffer of the ACE controller. The data buffer, which is 32 bytes, can only be written two bytes at a time. Once the data buffer is written, we wait for it to be empty again before writing the next buffer's worth of data. If the size of the incoming buffer is not aligned with the System ACE data buffer size (32 bytes), then this routine pads out the data buffer with zeros so the entire data buffer is written. This is necessary for the ACE controller to process the data buffer.

Parameters:

BaseAddress is the base address of the device

BufferPtr is a pointer to a buffer used to write to the controller.

Size is the number of bytes to write

Returns:

The total number of bytes written (not including pad bytes), or 0 if an error occurs.

Note:

None.


```
int XSysAce_WriteSector( Xuint32 BaseAddress,  
                        Xuint32 SectorId,  
                        Xuint8 * BufferPtr  
                        )
```

Write a CompactFlash sector. This is a blocking, low-level function which does not return until the specified sector is written in its entirety.

Parameters:

BaseAddress is the base address of the device

SectorId is the id of the sector to write

BufferPtr is a pointer to a buffer used to write the sector.

Returns:

The number of bytes written. If this number is not equal to the sector size, 512 bytes, then an error occurred.

Note:

None.

XSysAce_Config Struct Reference

```
#include <xsysace.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

Field Documentation

[Xuint32 XSysAce_Config::BaseAddress](#)

Register base address

[Xuint16 XSysAce_Config::DeviceId](#)

Unique ID of device

The documentation for this struct was generated from the following file:

- `sysace/v1_00_a/src/xsysace.h`
-

sysace/v1_00_a/src/xsysace_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of System ACE devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xparameters.h"
```

Variables

[XSysAce_Config](#) [XSysAce_ConfigTable](#) [XPAR_XSYSACE_NUM_INSTANCES]

Variable Documentation

[XSysAce_Config](#) [XSysAce_ConfigTable](#)[XPAR_XSYSACE_NUM_INSTANCES]

The configuration table for System ACE devices in the system. Each device should have an entry in this table.

Generated on 29 May 2003 for Xilinx Device Drivers

XSysAce_CFPParameters Struct Reference

```
#include <xsysace.h>
```

Detailed Description

Typedef for CompactFlash identify drive parameters. Use [XSysAce_IdentifyCF\(\)](#) to retrieve this information from the CompactFlash storage device.

Data Fields

[Xuint16 Signature](#)

[Xuint16 NumCylinders](#)

[Xuint16 NumHeads](#)

[Xuint16 NumBytesPerTrack](#)

[Xuint16 NumBytesPerSector](#)

[Xuint16 NumSectorsPerTrack](#)

[Xuint32 NumSectorsPerCard](#)

[Xuint16 VendorUnique](#)

[Xuint8 SerialNo](#) [20]

[Xuint16 BufferType](#)

[Xuint16 BufferSize](#)

[Xuint16 NumEccBytes](#)

[Xuint8 FwVersion](#) [8]

[Xuint8 ModelNo](#) [40]

[Xuint16 MaxSectors](#)

Xuint16 DblWord
Xuint16 Capabilities
Xuint16 PioMode
Xuint16 DmaMode
Xuint16 TranslationValid
Xuint16 CurNumCylinders
Xuint16 CurNumHeads
Xuint16 CurSectorsPerTrack
Xuint32 CurSectorsPerCard
Xuint16 MultipleSectors
Xuint32 LbaSectors
Xuint16 SecurityStatus
 Xuint8 VendorUniqueBytes [62]
Xuint16 PowerDesc

Field Documentation

Xuint16 XSysAce_CFPParameters::BufferSize

Buffer size in 512-byte increments

Xuint16 XSysAce_CFPParameters::BufferType

Buffer type

Xuint16 XSysAce_CFPParameters::Capabilities

Device capabilities

Xuint16 XSysAce_CFPParameters::CurNumCylinders

Current number of cylinders

Xuint16 XSysAce_CFPParameters::CurNumHeads

Current number of heads

Xuint32 XSysAce_CFPParameters::CurSectorsPerCard

Current capacity in sectors

Xuint16 XSysAce_CFPParameters::CurSectorsPerTrack

Current number of sectors per track

Xuint16 XSysAce_CFPParameters::DbiWord

Double Word not supported

Xuint16 XSysAce_CFPParameters::DmaMode

DMA data transfer cycle timing mode

Xuint8 XSysAce_CFPParameters::FwVersion[8]

ASCII firmware version

Xuint32 XSysAce_CFPParameters::LbaSectors

Number of addressable sectors in LBA mode

Xuint16 XSysAce_CFPParameters::MaxSectors

Max sectors on R/W Multiple cmds

Xuint8 XSysAce_CFPParameters::ModelNo[40]

ASCII model number

Xuint16 XSysAce_CFPParameters::MultipleSectors

Multiple sector setting

Xuint16 XSysAce_CFPParameters::NumBytesPerSector

Number of unformatted bytes per sector

Xuint16 XSysAce_CFPParameters::NumBytesPerTrack

Number of unformatted bytes per track

Xuint16 XSysAce_CFPParameters::NumCylinders

Default number of cylinders

Xuint16 XSysAce_CFPParameters::NumEccBytes

Number of ECC bytes on R/W Long cmds

Xuint16 XSysAce_CFPParameters::NumHeads

Default number of heads

Xuint32 XSysAce_CFPParameters::NumSectorsPerCard

Default number of sectors per card

Xuint16 XSysAce_CFPParameters::NumSectorsPerTrack

Default number of sectors per track

Xuint16 XSysAce_CFPParameters::PioMode

PIO data transfer cycle timing mode

Xuint16 XSysAce_CFPParameters::PowerDesc

Power requirement description

Xuint16 XSysAce_CFPParameters::SecurityStatus

Security status

Xuint8 XSysAce_CFPParameters::SerialNo[20]

ASCII serial number

Xuint16 XSysAce_CFPParameters::Signature

CompactFlash signature is 0x848a

Xuint16 XSysAce_CFPParameters::TranslationValid

Translation parameters are valid

Xuint16 XSysAce_CFPParameters::VendorUnique

Vendor unique

Xuint8 XSysAce_CFPParameters::VendorUniqueBytes[62]

Vendor unique bytes

The documentation for this struct was generated from the following file:

- [sysace/v1_00_a/src/xsysace.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

sysace/v1_00_a/src/xsysace_compactflash.c

File Reference

Detailed Description

Contains functions to reset, read, and write the CompactFlash device via the System ACE controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

```
XStatus XSysAce_ResetCF (XSysAce *InstancePtr)  
XStatus XSysAce_AbortCF (XSysAce *InstancePtr)  
XStatus XSysAce_IdentifyCF (XSysAce *InstancePtr, XSysAce_CFParameters *ParamPtr)  
Xboolean XSysAce_IsCFReady (XSysAce *InstancePtr)  
XStatus XSysAce_SectorRead (XSysAce *InstancePtr, Xuint32 StartSector, int NumSectors, Xuint8 *BufferPtr)  
XStatus XSysAce_SectorWrite (XSysAce *InstancePtr, Xuint32 StartSector, int NumSectors, Xuint8 *BufferPtr)  
Xuint16 XSysAce_GetFatStatus (XSysAce *InstancePtr)
```

Function Documentation

XStatus XSysAce_AbortCF(XSysAce * InstancePtr)

Abort the CompactFlash operation currently in progress.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

- XST_SUCCESS if the abort was done successfully
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

According to the ASIC designer, the abort command has not been well tested.

Xuint16 XSysAce_GetFatStatus(XSysAce * InstancePtr)

Get the status of the FAT filesystem on the first valid partition of the CompactFlash device such as the boot record and FAT types found.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

A 16-bit mask of status values. These values are defined in [xsysace_1.h](#) with the prefix XSA_FAT_*.

Note:

None.

XStatus XSysAce_IdentifyCF(XSysAce * InstancePtr, XSysAce_CFPParameters * ParamPtr)

Identify the CompactFlash device. Retrieves the parameters for the CompactFlash storage device. Note that this is a polled read of one sector of data. The data is read from the CompactFlash into a byte buffer, which is then copied into the **XSysAce_CFPParameters** structure passed in by the user. The copy is necessary since we don't know how the compiler packs the **XSysAce_CFPParameters** structure.

An MPU lock, obtained using **XSysAce_Lock()**, must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

ParamPtr is a pointer to a **XSysAce_CFPParameters** structure where the information for the CompactFlash device will be stored. See **xsysace.h** for details on the **XSysAce_CFPParameters** structure.

Returns:

- XST_SUCCESS if the identify was done successfully
- XST_FAILURE if an error occurs. Use **XSysAce_GetErrors()** to determine cause.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

None.

Xboolean XSysAce_IsCFReady(XSysAce * InstancePtr)

Check to see if the CompactFlash is ready for a command. The CompactFlash may delay after one operation before it is ready for the next. This function helps the user determine when it is ready before invoking a CompactFlash operation such as **XSysAce_SectorRead()** or **XSysAce_SectorWrite()**;

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

XTRUE if the CompactFlash is ready for a command, and XFALSE otherwise.

Note:

None.

XStatus XSysAce_ResetCF(XSysAce * InstancePtr)

Reset the CompactFlash device. This function does not reset the System ACE controller. An ATA soft-reset of the CompactFlash is performed.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

Returns:

- XST_SUCCESS if the reset was done successfully
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the CompactFlash is not ready for a command

Note:

None.

```
XStatus XSysAce_SectorRead( XSysAce * InstancePtr,  
                           Xuint32   StartSector,  
                           int       NumSectors,  
                           Xuint8 * BufferPtr  
                           )
```

Read at least one sector of data from the CompactFlash. The user specifies the starting sector ID and the number of sectors to be read. The minimum unit that can be read from the CompactFlash is a sector, which is 512 bytes.

In polled mode, this read is blocking. If there are other tasks in the system that must run, it is best to keep the number of sectors to be read to a minimum (e.g., 1). In interrupt mode, this read is non-blocking and an event, XSA_EVENT_DATA_DONE, is returned to the user in the asynchronous event handler when the read is complete. The user must call [XSysAce_EnableInterrupt\(\)](#) to put the driver/device into interrupt mode.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

StartSector is the starting sector ID from where data will be read. Sector IDs range from 0 (first sector) to 0x10000000.

NumSectors is the number of sectors to read. The range can be from 1 to 256.

BufferPtr is a pointer to a buffer where the data will be stored. The user must ensure it is big enough to hold (512 * NumSectors) bytes.

Returns:

- XST_SUCCESS if the read was successful. In interrupt mode, this does not mean the read is complete, only that it has begun. An event is returned to the user when the read is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the ACE controller is not ready for a command
- XST_FAILURE if an error occurred during the read. The user should call [XSysAce_GetErrors\(\)](#) to determine the cause of the error.

Note:

None.

```
XStatus XSysAce_SectorWrite( XSysAce * InstancePtr,  
                             Xuint32   StartSector,  
                             int       NumSectors,  
                             Xuint8 * BufferPtr  
                             )
```

Write data to the CompactFlash. The user specifies the starting sector ID and the number of sectors to be written. The minimum unit that can be written to the CompactFlash is a sector, which is 512 bytes.

In polled mode, this write is blocking. If there are other tasks in the system that must run, it is best to keep the number of sectors to be written to a minimum (e.g., 1). In interrupt mode, this write is non-blocking and an event, XSA_EVENT_DATA_DONE, is returned to the user in the asynchronous event handler when the write is complete. The user must call [XSysAce_EnableInterrupt\(\)](#) to put the driver/device into interrupt mode.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

StartSector is the starting sector ID from where data will be written. Sector IDs range from 0 (first sector) to 0x10000000.

NumSectors is the number of sectors to write. The range can be from 1 to 256.

BufferPtr is a pointer to the data buffer to be written. This buffer must have at least (512 * NumSectors) bytes.

Returns:

- XST_SUCCESS if the write was successful. In interrupt mode, this does not mean the write is complete, only that it has begun. An event is returned to the user when the write is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_DEVICE_BUSY if the ACE controller is not ready for a command
- XST_FAILURE if an error occurred during the write. The user should call [XSysAce_GetErrors\(\)](#) to determine the cause of the error.

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

sysace/v1_00_a/src/xsysace_jtagcfg.c

File Reference

Detailed Description

Contains functions to control the configuration of the target FPGA chain on the System ACE via the JTAG configuration port.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

```
void XSysAce_ResetCfg (XSysAce *InstancePtr)  
void XSysAce_SetCfgAddr (XSysAce *InstancePtr, unsigned int Address)  
void XSysAce_SetStartMode (XSysAce *InstancePtr, Xboolean ImmedOnReset, Xboolean  
StartCfg)  
XStatus XSysAce_ProgramChain (XSysAce *InstancePtr, Xuint8 *BufferPtr, int NumBytes)  
Xboolean XSysAce_IsCfgDone (XSysAce *InstancePtr)  
Xuint32 XSysAce_GetCfgSector (XSysAce *InstancePtr)
```

Function Documentation

Xuint32 XSysAce_GetCfgSector(**XSysAce** * *InstancePtr*)

Get the sector ID of the CompactFlash sector being used for configuration of the target FPGA chain. This sector ID (or logical block address) only affects transfers between the ACE configuration logic and the CompactFlash card. This function is typically used for debug purposes to determine which sector was being accessed when an error occurred.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

The sector ID (logical block address) being used for data transfers between the ACE configuration logic and the CompactFlash. Sector IDs range from 0 to 0x10000000.

Note:

None.

Xboolean XSysAce_IsCfgDone(**XSysAce** * *InstancePtr*)

Check to see if the configuration of the target FPGA chain is complete. This function is typically only used in polled mode. In interrupt mode, an event (XSA_EVENT_CFG_DONE) is returned to the user in the asynchronous event handler.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

XTRUE if the configuration is complete. XFALSE otherwise.

Note:

None.

XStatus XSysAce_ProgramChain(**XSysAce** * *InstancePtr*,
 Xuint8 * *BufferPtr*,
 int *NumBytes*
)

Program the target FPGA chain through the configuration JTAG port. This allows the user to program the devices on the target FPGA chain from the MPU port instead of from CompactFlash. The user specifies a buffer and the number of bytes to write. The buffer should be equivalent to an ACE (.ace) file.

Note that when loading the ACE file via the MPU port, the first sector of the ACE file is discarded. The CF filesystem controller in the System ACE device knows to skip the first sector when the ACE file comes from the CF, but the CF filesystem controller is bypassed when the ACE file comes from the MPU port. For this reason, this function skips the first sector of the buffer passed in.

In polled mode, the write is blocking. In interrupt mode, the write is non-blocking and an event, XSA_EVENT_CFG_DONE, is returned to the user in the asynchronous event handler when the configuration is complete.

An MPU lock, obtained using [XSysAce_Lock\(\)](#), must be granted before calling this function. If a lock has not been granted, no action is taken and an error is returned.

Parameters:

InstancePtr is a pointer to the [XSysAce](#) instance to be worked on.

BufferPtr is a pointer to a buffer that will be used to program the configuration JTAG devices.

NumBytes is the number of bytes in the buffer. We assume that there is at least one sector of data in the .ace file, which is the information sector.

Returns:

- XST_SUCCESS if the write was successful. In interrupt mode, this does not mean the write is complete, only that it has begun. An event is returned to the user when the write is complete.
- XST_SYSACE_NO_LOCK if no MPU lock has yet been granted
- XST_FAILURE if an error occurred during the write. The user should call [XSysAce_GetErrors\(\)](#) to determine the cause of the error.

Note:

None.

```
void XSysAce_ResetCfg( XSysAce * InstancePtr)
```

Reset the JTAG configuration controller. This comprises a reset of the JTAG configuration controller and the CompactFlash controller (if it is currently being accessed by the configuration controller). Note that the MPU controller is not reset, meaning the MPU registers remain unchanged. The configuration controller is reset then released from reset in this function.

The CFGDONE status (and therefore interrupt) is cleared when the configuration controller is reset. If interrupts have been enabled, we go ahead and enable the CFGDONE interrupt here. This means that if and when a configuration process starts as a result of this reset, an interrupt will be received when it is complete.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

None.

Note:

This function is not thread-safe.

```
void XSysAce_SetCfgAddr( XSysAce * InstancePtr,  
                        unsigned int Address  
                        )
```

Select the configuration address (or file) from the CompactFlash to be used for configuration of the target FPGA chain.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Address is the address or file number to be used as the bitstream to configure the target FPGA devices. There are 8 possible files, so the value of this parameter can range from 0 to 7.

Returns:

None.

Note:

None.

```
void XSysAce_SetStartMode( XSysAce * InstancePtr,  
                          Xboolean ImmedOnReset,  
                          Xboolean StartCfg  
                          )
```

Set the start mode for configuration of the target FPGA chain from CompactFlash. The configuration process only starts after a reset. The user can indicate that the configuration should start immediately after a reset, or the configuration process can be delayed until the user commands it to start (using this function). The configuration controller can be reset using [XSysAce_ResetCfg\(\)](#).

The user can select which configuration file on the CompactFlash to use using the [XSysAce_SetCfgAddr\(\)](#) function. If the user intends to configure the target FPGA chain directly from the MPU port, this function is not needed. Instead, the user would simply call [XSysAce_ProgramChain\(\)](#).

The user can use [XSysAce_IsCfgDone\(\)](#) when in polled mode to determine if the configuration is complete. If in interrupt mode, the event XSA_EVENT_CFG_DONE will be returned asynchronously to the user when the configuration is complete. The user must call [XSysAce_EnableInterrupt\(\)](#) to put the device/driver into interrupt mode.

Parameters:

- InstancePtr* is a pointer to the [XSysAce](#) instance to be worked on.
- ImmedOnReset* can be set to XTRUE to indicate the configuration process will start immediately after a reset of the ACE configuration controller, or it can be set to XFALSE to indicate the configuration process is delayed after a reset until the user starts it (using this function).
- StartCfg* is a boolean indicating whether to start the configuration process or not. When ImmedOnReset is set to XTRUE, this value is ignored. When ImmedOnReset is set to XFALSE, then this value controls when the configuration process is started. When set to XTRUE the configuration process starts (assuming a reset of the device has occurred), and when set to XFALSE the configuration process does not start.

Returns:

None.

Note:

None.

sysace/v1_00_a/src/xsysace_intr.c File Reference

Detailed Description

Contains functions related to System ACE interrupt mode. The driver's interrupt handler, **[XSysAce_InterruptHandler\(\)](#)**, must be connected by the user to the interrupt controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

```
void XSysAce\_EnableInterrupt (XSysAce *InstancePtr)  
void XSysAce\_DisableInterrupt (XSysAce *InstancePtr)  
void XSysAce\_InterruptHandler (void *InstancePtr)  
void XSysAce\_SetEventHandler (XSysAce *InstancePtr, XSysAce\_EventHandler FuncPtr, void *CallbackRef)
```

Function Documentation

void XSysAce_DisableInterrupt(XSysAce * InstancePtr)

Disable all System ACE interrupts and hold the interrupt request line of the device in reset.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance that just interrupted.

Returns:

None.

Note:

None.

void XSysAce_EnableInterrupt(XSysAce * InstancePtr)

Enable System ACE interrupts. There are three interrupts that can be enabled. The error interrupt enable serves as the driver's means to determine whether interrupts have been enabled or not. The configuration-done interrupt is not enabled here, instead it is enabled during a reset - which can cause a configuration process to start. The data-buffer-ready interrupt is not enabled here either. It is enabled when a read or write operation is started. The reason for not enabling the latter two interrupts are because the status bits may be set as a leftover of an earlier occurrence of the interrupt.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to work on.

Returns:

None.

Note:

None.

void XSysAce_IRQHandler(void * InstancePtr)

The interrupt handler for the System ACE driver. This handler must be connected by the user to an interrupt controller or source. This function does not save or restore context.

This function continues reading or writing to the compact flash if such an operation is in progress, and notifies the upper layer software through the event handler once the operation is complete or an error occurs. On an error, any command currently in progress is aborted.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance that just interrupted.

Returns:

None.

Note:

None.

```
void XSysAce_SetEventHandler( XSysAce *           InstancePtr,  
                             XSysAce_EventHandler FuncPtr,  
                             void *             CallBackRef  
                             )
```

Set the callback function for handling events. The upper layer software should call this function during initialization. The events are passed asynchronously to the upper layer software. The events are described in **xsysace.h** and are named XSA_EVENT_*

Note that the callback is invoked by the driver within interrupt context, so it needs to do its job quickly. If there are potentially slow operations within the callback, these should be done at task-level.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallBackRef is a reference pointer to be passed back to the upper layer.

Returns:

None.

Note:

None.

sysace/v1_00_a/src/xsysace_selftest.c

File Reference

Detailed Description

Contains diagnostic functions for the System ACE device and driver. This includes a self-test to make sure communication to the device is possible and the ability to retrieve the ACE controller version.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rpm	06/17/02	work in progress

```
#include "xsysace.h"  
#include "xsysace_1.h"
```

Functions

XStatus [XSysAce_SelfTest](#) ([XSysAce](#) *InstancePtr)

Xuint16 [XSysAce_GetVersion](#) ([XSysAce](#) *InstancePtr)

Function Documentation

Xuint16 [XSysAce_GetVersion](#)([XSysAce](#) * *InstancePtr*)

Retrieve the version of the System ACE device.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

A 16-bit version where the 4 most significant bits are the major version number, the next four bits are the minor version number, and the least significant 8 bits are the revision or build number.

Note:

None.

XStatus XSysAce_SelfTest(**XSysAce** * *InstancePtr*)

A self-test that simply proves communication to the ACE controller from the device driver by obtaining an MPU lock, verifying it, then releasing it.

Parameters:

InstancePtr is a pointer to the **XSysAce** instance to be worked on.

Returns:

XST_SUCCESS if self-test passes, or XST_FAILURE if an error occurs.

Note:

None.

spi/v1_00_b/src/xspi.h File Reference

Detailed Description

This component contains the implementation of the **XSpi** component. It is the driver for an SPI master or slave device. User documentation for the driver functions is contained in this file in the form of comment blocks at the front of each function.

SPI is a 4-wire serial interface. It is a full-duplex, synchronous bus that facilitates communication between one master and one slave. The device is always full-duplex, which means that for every byte sent, one is received, and vice-versa. The master controls the clock, so it can regulate when it wants to send or receive data. The slave is under control of the master, it must respond quickly since it has no control of the clock and must send/receive data as fast or as slow as the master does.

The application software between master and slave must implement a higher layer protocol so that slaves know what to transmit to the master and when.

Multiple Masters

More than one master can exist, but arbitration is the responsibility of the higher layer software. The device driver does not perform any type of arbitration.

Multiple Slaves

Multiple slaves are supported by adding additional slave select (SS) signals to each device, one for each slave on the bus. The driver ensures that only one slave can be selected at any one time.

FIFOs

The SPI hardware is parameterized such that it can be built with or without FIFOs. When using FIFOs, both send and receive must have FIFOs. The driver will not function correctly if one direction has a FIFO but the other direction does not. The frequency of the interrupts which occur is proportional to the data rate such that high data rates without the FIFOs could cause the software to consume large amounts of processing time. The driver is designed to work with or without the FIFOs.

Interrupts

The user must connect the interrupt handler of the driver, `XSpi_IRQHandler` to an interrupt system such that it will be called when an interrupt occurs. This function does not save and restore the processor context such that the user must

provide this processing.

The driver handles the following interrupts:

- Data Transmit Register/FIFO Empty
- Data Transmit Register/FIFO Underrun
- Data Receive Register/FIFO Overrun
- Mode Fault Error
- Slave Mode Fault Error

The Data Transmit Register/FIFO Empty interrupt indicates that the SPI device has transmitted all the data available to transmit, and now its data register (or FIFO) is empty. The driver uses this interrupt to indicate progress while sending data. The driver may have more data to send, in which case the data transmit register (or FIFO) is filled for subsequent transmission. When this interrupt arrives and all the data has been sent, the driver invokes the status callback with a value of `XST_SPI_TRANSFER_DONE` to inform the upper layer software that all data has been sent.

The Data Transmit Register/FIFO Underrun interrupt indicates that, as slave, the SPI device was required to transmit but there was no data available to transmit in the transmit register (or FIFO). This may not be an error if the master is not expecting data, but in the case where the master is expecting data this serves as a notification of such a condition. The driver reports this condition to the upper layer software through the status handler.

The Data Receive Register/FIFO Overrun interrupt indicates that the SPI device received data and subsequently dropped the data because the data receive register (or FIFO) was full. The interrupt applies to both master and slave operation. The driver reports this condition to the upper layer software through the status handler. This likely indicates a problem with the higher layer protocol, or a problem with the slave performance.

The Mode Fault Error interrupt indicates that while configured as a master, the device was selected as a slave by another master. This can be used by the application for arbitration in a multimaster environment or to indicate a problem with arbitration. When this interrupt occurs, the driver invokes the status callback with a status value of `XST_SPI_MODE_FAULT`. It is up to the application to resolve the conflict.

The Slave Mode Fault Error interrupt indicates that a slave device was selected as a slave by a master, but the slave device was disabled. This can be used during system debugging or by the slave application to learn when the slave application has not prepared for a master operation in a timely fashion. This likely indicates a problem with the higher layer protocol, or a problem with the slave performance.

Note that during the FPGA implementation process, the interrupt registers of the IPIF can be parameterized away. This driver is currently dependent on those interrupt registers and will not function without them.

Polled Operation

Currently there is no support for polled operation.

Device Busy

Some operations are disallowed when the device is busy. The driver tracks whether a device is busy. The device is considered busy when a data transfer request is outstanding, and is considered not busy only when that transfer completes (or is aborted with a mode fault error). This applies to both master and slave devices.

Device Configuration

The device can be configured in various ways during the FPGA implementation process. Configuration parameters are stored in the `xspi_g.c` file. A table is defined where each entry contains configuration information for an SPI device. This information includes such things as the base address of the memory-mapped device, the base address of the IPIF module within the device, the number of slave select bits in the device, and whether the device has FIFOs and is configured as slave-only.

RTOS Independence

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	10/11/01	First release
1.00b	jhl	03/14/02	Repartitioned driver for smaller files.

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XSpi  
struct XSpi_Config  
struct XSpi_Stats
```

Configuration options

The following options may be specified or retrieved for the device and enable/disable additional features of the SPI. Each of the options are bit fields, so more than one may be specified.

```
#define XSP_MASTER_OPTION  
#define XSP_CLK_ACTIVE_LOW_OPTION  
#define XSP_CLK_PHASE_1_OPTION  
#define XSP_LOOPBACK_OPTION  
#define XSP_MANUAL_SSELECT_OPTION
```

Typedefs

```
typedef void(* XSpi_StatusHandler )(void *CallBackRef, Xuint32 StatusEvent, unsigned int ByteCount)
```

Functions

```
XStatus XSpi_Initialize (XSpi *InstancePtr, Xuint16 DeviceId)
```

```
XStatus XSpi_Start (XSpi *InstancePtr)
```

```
XStatus XSpi_Stop (XSpi *InstancePtr)
```

```
void XSpi_Reset (XSpi *InstancePtr)
```

```
XStatus XSpi_SetSlaveSelect (XSpi *InstancePtr, Xuint32 SlaveMask)
```

```
Xuint32 XSpi_GetSlaveSelect (XSpi *InstancePtr)
```

```
XStatus XSpi_Transfer (XSpi *InstancePtr, Xuint8 *SendBufPtr, Xuint8 *RecvBufPtr, unsigned int  
ByteCount)
```

```
void XSpi_SetStatusHandler (XSpi *InstancePtr, void *CallBackRef, XSpi_StatusHandler FuncPtr)
```

```
void XSpi_InterruptHandler (void *InstancePtr)
```

```
XSpi_Config * XSpi_LookupConfig (Xuint16 DeviceId)
```

```
XStatus XSpi_SelfTest (XSpi *InstancePtr)
```

```
void XSpi_GetStats (XSpi *InstancePtr, XSpi_Stats *StatsPtr)
```

```
void XSpi_ClearStats (XSpi *InstancePtr)
```

```
XStatus XSpi_SetOptions (XSpi *InstancePtr, Xuint32 Options)
```

```
Xuint32 XSpi_GetOptions (XSpi *InstancePtr)
```

Define Documentation

#define XSP_CLK_ACTIVE_LOW_OPTION

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

#define XSP_CLK_PHASE_1_OPTION

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

#define XSP_LOOPBACK_OPTION

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

#define XSP_MANUAL_SSELECT_OPTION

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

#define XSP_MASTER_OPTION

The Master option configures the SPI device as a master. By default, the device is a slave.

The Active Low Clock option configures the device's clock polarity. Setting this option means the clock is active low and the SCK signal idles high. By default, the clock is active high and SCK idles low.

The Clock Phase option configures the SPI device for one of two transfer formats. A clock phase of 0, the default, means data is valid on the first SCK edge (rising or falling) after the slave select (SS) signal has been asserted. A clock phase of 1 means data is valid on the second SCK edge (rising or falling) after SS has been asserted.

The Loopback option configures the SPI device for loopback mode. Data is looped back from the transmitter to the receiver.

The Manual Slave Select option, which is default, causes the device not to automatically drive the slave select. The driver selects the device at the start of a transfer and deselects it at the end of a transfer. If this option is off, then the device automatically toggles the slave select signal between bytes in a transfer.

Typedef Documentation

```
typedef void(* XSpi_StatusHandler)(void *CallbackRef, Xuint32 StatusEvent, unsigned int ByteCount)
```

The handler data type allows the user to define a callback function to handle the asynchronous processing of the SPI driver. The application using this driver is expected to define a handler of this type to support interrupt driven mode. The handler executes in an interrupt context such that minimal processing should be performed.

Parameters:

- CallbackRef* A callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. Its type is unimportant to the driver component, so it is a void pointer.
 - StatusEvent* Indicates one or more status events that occurred. See the [XSpi_SetStatusHandler\(\)](#) for details on the status events that can be passed in the callback.
 - ByteCount* Indicates how many bytes of data were successfully transferred. This may be less than the number of bytes requested if the status event indicates an error.
-

Function Documentation

```
void XSpi_ClearStats( XSpi * InstancePtr)
```

Clears the statistics for the SPI device.

Parameters:

InstancePtr is a pointer to the [XSpi](#) instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 XSpi_GetOptions( XSpi * InstancePtr)
```

This function gets the options for the SPI device. The options control how the device behaves relative to the SPI bus.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. One or more bit values may be contained in the mask. See the bit definitions named XSP_*_OPTIONS in the file **xspi.h**.

Note:

None.

Xuint32 XSpi_GetSlaveSelect(XSpi * InstancePtr)

Gets the current slave select bit mask for the SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

The value returned is a 32-bit mask with a 1 in the bit position of the slave currently selected. The value may be zero if no slaves are selected.

Note:

None.

```
void XSpi_GetStats( XSpi * InstancePtr,  
                  XSpi_Stats * StatsPtr  
                  )
```

Gets a copy of the statistics for an SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

StatsPtr is a pointer to a **XSpi_Stats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

```
XStatus XSpi_Initialize( XSpi * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initializes a specific **XSpi** instance such that the driver is ready to use.

The state of the device after initialization is:

- Device is disabled
- Slave mode
- Active high clock polarity
- Clock phase 0

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XSpi** instance. Passing in a device id associates the generic **XSpi** instance to a specific device, as chosen by the caller or application developer.

Returns:

The return value is **XST_SUCCESS** if successful. On error, a code indicating the specific error is returned.

Possible error codes are:

- **XST_DEVICE_IS_STARTED** if the device is started. It must be stopped to re-initialize.
- **XST_DEVICE_NOT_FOUND** if the device was not found in the configuration such that initialization could not be accomplished.

Note:

None.

```
void XSpi_InterruptHandler( void * InstancePtr)
```

The interrupt handler for SPI interrupts. This function must be connected by the user to an interrupt source. This function does not save and restore the processor context such that the user must provide this processing.

The interrupts that are handled are:

- Mode Fault Error. This interrupt is generated if this device is selected as a slave when it is configured as a master. The driver aborts any data transfer that is in progress by resetting FIFOs (if present) and resetting its buffer pointers. The upper layer software is informed of the error.
- Data Transmit Register (FIFO) Empty. This interrupt is generated when the transmit register or FIFO is empty. The driver uses this interrupt during a transmission to continually send/receive data until there is no more data to send/receive.
- Data Transmit Register (FIFO) Underrun. This interrupt is generated when the SPI device, when configured as a slave, attempts to read an empty DTR/FIFO. An empty DTR/FIFO usually means that software is not giving the device data in a timely manner. No action is taken by the driver other than to inform the upper layer software of the error.

- Data Receive Register (FIFO) Overrun. This interrupt is generated when the SPI device attempts to write a received byte to an already full DRR/FIFO. A full DRR/FIFO usually means software is not emptying the data in a timely manner. No action is taken by the driver other than to inform the upper layer software of the error.
- Slave Mode Fault Error. This interrupt is generated if a slave device is selected as a slave while it is disabled. No action is taken by the driver other than to inform the upper layer software of the error.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

The slave select register is being set to deselect the slave when a transfer is complete. This is being done regardless of whether it is a slave or a master since the hardware does not drive the slave select as a slave.

XSpi_Config* XSpi_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found. See **xspi.h** for the definition of **XSpi_Config**.

Note:

None.

void XSpi_Reset(XSpi * InstancePtr)

Resets the SPI device. Reset must only be called after the driver has been initialized. The configuration of the device after reset is the same as its configuration after initialization. Refer to the XSpi_Initialize function for more details. This is a hard reset of the device. Any data transfer that is in progress is aborted.

The upper layer software is responsible for re-configuring (if necessary) and restarting the SPI device after the reset.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

None.

XStatus XSpi_SelfTest(**XSpi** * *InstancePtr*)

Runs a self-test on the driver/device. The self-test is destructive in that a reset of the device is performed in order to check the reset values of the registers and to get the device into a known state. A simple loopback test is also performed to verify that transmit and receive are working properly. The device is changed to master mode for the loopback test, since only a master can initiate a data transfer.

Upon successful return from the self-test, the device is reset.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

- XST_SUCCESS if successful, or one of the following error codes otherwise.
 - o XST_REGISTER_ERROR indicates a register did not read or write correctly
 - o XST_LOOPBACK_ERROR if a loopback error occurred.

Note:

None.

XStatus XSpi_SetOptions(**XSpi** * *InstancePtr*,
 Xuint32 *Options*
))

This function sets the options for the SPI device driver. The options control how the device behaves relative to the SPI bus. The device must be idle rather than busy transferring data before setting these device options.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. One or more bit values may be contained in the mask. See the bit definitions named XSP_*_OPTIONS in the file **xspi.h**.

Returns:

XST_SUCCESS if options are successfully set. Otherwise, returns:

- XST_DEVICE_BUSY if the device is currently transferring data. The transfer must complete or be aborted before setting options.
- XST_SPI_SLAVE_ONLY if the caller attempted to configure a slave-only device as a master.

Note:

This function makes use of internal resources that are shared between the **XSpi_Stop()** and **XSpi_SetOptions()** functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

```
XStatus XSpi_SetSlaveSelect( XSpi * InstancePtr,  
                             Xuint32 SlaveMask  
                             )
```

Selects or deselect the slave with which the master communicates. Each slave that can be selected is represented in the slave select register by a bit. The argument passed to this function is the bit mask with a 1 in the bit position of the slave being selected. Only one slave can be selected.

The user is not allowed to deselect the slave while a transfer is in progress. If no transfer is in progress, the user can select a new slave, which implicitly deselects the current slave. In order to explicitly deselect the current slave, a zero can be passed in as the argument to the function.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

SlaveMask is a 32-bit mask with a 1 in the bit position of the slave being selected. Only one slave can be selected. The SlaveMask can be zero if the slave is being deselected.

Returns:

XST_SUCCESS if the slave is selected or deselected successfully. Otherwise, returns:

- XST_DEVICE_BUSY if a transfer is in progress, slave cannot be changed
- XST_SPI_TOO_MANY_SLAVES if more than one slave is being selected.

Note:

This function only sets the slave which will be selected when a transfer occurs. The slave is not selected when the SPI is idle. The slave select has no affect when the device is configured as a slave.

```

void XSpi_SetStatusHandler( XSpi *           InstancePtr,
                           void *          CallbackRef,
                           XSpi_StatusHandler FuncPtr
                           )

```

Sets the status callback function, the status handler, which the driver calls when it encounters conditions that should be reported to the higher layer software. The handler executes in an interrupt context, so it must minimize the amount of processing performed such as transferring data to a thread context. One of the following status events is passed to the status handler.

XST_SPI_MODE_FAULT	A mode fault error occurred, meaning another master tried to select this device as a slave when this device was configured to be a master. Any transfer in progress is aborted.
XST_SPI_TRANSFER_DONE	The requested data transfer is done
XST_SPI_TRANSMIT_UNDERRUN	As a slave device, the master clocked data but there were none available in the transmit register/FIFO. This typically means the slave application did not issue a transfer request fast enough, or the processor/driver could not fill the transmit register/FIFO fast enough.
XST_SPI_RECEIVE_OVERRUN	The SPI device lost data. Data was received but the receive data register/FIFO was full. This indicates that the device is receiving data
XST_SPI_SLAVE_MODE_FAULT	faster than the processor/driver can consume it. A slave SPI device was selected as a slave while it was disabled. This indicates the master is already transferring data (which is being dropped until the slave application issues a transfer).

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context, so it should do its work quickly and queue potentially time-consuming work to a task-level thread.

XStatus XSpi_Start(XSpi * InstancePtr)

This function enables interrupts for the SPI device. It is up to the user to connect the SPI interrupt handler to the interrupt controller before this Start function is called. The GetIntrHandler function is used for that purpose. If the device is configured with FIFOs, the FIFOs are reset at this time.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if the device is successfully started, or XST_DEVICE_IS_STARTED if the device was already started.

Note:

None.

XStatus XSpi_Stop(XSpi * InstancePtr)

This function stops the SPI device by disabling interrupts and disabling the device itself. Interrupts are disabled only within the device itself. If desired, the caller is responsible for disabling interrupts in the interrupt controller and disconnecting the interrupt handler from the interrupt controller.

If the device is in progress of transferring data on the SPI bus, this function returns a status indicating the device is busy. The user will be notified via the status handler when the transfer is complete, and at that time can again try to stop the device. As a master, we do not allow the device to be stopped while a transfer is in progress because the slave may be left in a bad state. As a slave, we do not allow the device to be stopped while a transfer is in progress because the master is not done with its transfer yet.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if the device is successfully started, or XST_DEVICE_BUSY if a transfer is in progress and cannot be stopped.

Note:

This function makes use of internal resources that are shared between the **XSpi_Stop()** and **XSpi_SetOptions()** functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

```
XStatus XSpi_Transfer( XSpi * InstancePtr,  
                     Xuint8 * SendBufPtr,  
                     Xuint8 * RecvBufPtr,  
                     unsigned int ByteCount  
                     )
```

Transfers the specified data on the SPI bus. If the SPI device is configured to be a master, this function initiates bus communication and sends/receives the data to/from the selected SPI slave. If the SPI device is configured to be a slave, this function prepares the data to be sent/received when selected by a master. For every byte sent, a byte is received.

The caller has the option of providing two different buffers for send and receive, or one buffer for both send and receive, or no buffer for receive. The receive buffer must be at least as big as the send buffer to prevent unwanted memory writes. This implies that the byte count passed in as an argument must be the smaller of the two buffers if they differ in size. Here are some sample usages:

```
XSpi_Transfer(InstancePtr, SendBuf, RecvBuf, ByteCount)
```

The caller wishes to send and receive, and provides two different buffers for send and receive.

```
XSpi_Transfer(InstancePtr, SendBuf, NULL, ByteCount)
```

The caller wishes only to send and does not care about the received data. The driver ignores the received data in this case.

```
XSpi_Transfer(InstancePtr, SendBuf, SendBuf, ByteCount)
```

The caller wishes to send and receive, but provides the same buffer for doing both. The driver sends the data and overwrites the send buffer with received data as it transfers the data.

```
XSpi_Transfer(InstancePtr, RecvBuf, RecvBuf, ByteCount)
```

The caller wishes to only receive and does not care about sending data. In this case, the caller must still provide a send buffer, but it can be the same as the receive buffer if the caller does not care what it sends. The device must send N bytes of data if it wishes to receive N bytes of data.

Although this function takes a buffer as an argument, the driver can only transfer a limited number of bytes at time. It transfers only one byte at a time if there are no FIFOs, or it can transfer the number of bytes up to the size of the FIFO. A call to this function only starts the transfer, then subsequent transfer of the data is performed by the interrupt service routine until the entire buffer has been transferred. The status callback function is called when the entire buffer has been sent/received.

This function is non-blocking. As a master, the SetSlaveSelect function must be called prior to this function.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

SendBufPtr is a pointer to a buffer of data which is to be sent. This buffer must not be NULL.

RecvBufPtr is a pointer to a buffer which will be filled with received data. This argument can be NULL if the caller does not wish to receive data.

ByteCount contains the number of bytes to send/receive. The number of bytes received always equals the number of bytes sent.

Returns:

XST_SUCCESS if the buffers are successfully handed off to the driver for transfer. Otherwise, returns:

- XST_DEVICE_IS_STOPPED if the device must be started before transferring data.
- XST_DEVICE_BUSY indicates that a data transfer is already in progress. This is determined by the driver.
- XST_SPI_NO_SLAVE indicates the device is configured as a master and a slave has not yet been selected.

Note:

This function is not thread-safe. The higher layer software must ensure that no two threads are transferring data on the SPI bus at the same time.

Generated on 29 May 2003 for Xilinx Device Drivers

XSpi Struct Reference

```
#include <xspi.h>
```

Detailed Description

The XSpi driver instance data. The user is required to allocate a variable of this type for every SPI device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [spi/v1_00_b/src/xspi.h](#)
-

spi/v1_00_b/src/xspi.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xspi.h,v 1.4 2002/05/02 20:31:10 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file spi/v1_00_b/src/xspi.h
00026 *
00027 * This component contains the implementation of the XSpi component. It is the
00028 * driver for an SPI master or slave device. User documentation for the driver
00029 * functions is contained in this file in the form of comment blocks at the
00030 * front of each function.
00031 *
00032 * SPI is a 4-wire serial interface. It is a full-duplex, synchronous bus that
00033 * facilitates communication between one master and one slave. The device is
00034 * always full-duplex, which means that for every byte sent, one is received,
00035 * and
00036 * vice-versa. The master controls the clock, so it can regulate when it wants
00037 * to send or receive data. The slave is under control of the master, it must
00038 * respond quickly since it has no control of the clock and must send/receive
00039 * data as fast or as slow as the master does.
00040 *
00041 * The application software between master and slave must implement a higher
00042 * layer protocol so that slaves know what to transmit to the master and when.
```

00042 *
00043 * **Multiple Masters**
00044 *
00045 * More than one master can exist, but arbitration is the responsibility of the
00046 * higher layer software. The device driver does not perform any type of
00047 * arbitration.
00048 *
00049 * **Multiple Slaves**
00050 *
00051 * Multiple slaves are supported by adding additional slave select (SS) signals
00052 * to each device, one for each slave on the bus. The driver ensures that only
00053 * one slave can be selected at any one time.
00054 *
00055 * **FIFOs**
00056 *
00057 * The SPI hardware is parameterized such that it can be built with or without
00058 * FIFOs. When using FIFOs, both send and receive must have FIFOs. The driver
00059 * will not function correctly if one direction has a FIFO but the other
00060 * direction does not. The frequency of the interrupts which occur is
00061 * proportional to the data rate such that high data rates without the FIFOs
00062 * could cause the software to consume large amounts of processing time. The
00063 * driver is designed to work with or without the FIFOs.
00064 *
00065 * **Interrupts**
00066 *
00067 * The user must connect the interrupt handler of the driver,
00068 * XSpi_InterruptHandler to an interrupt system such that it will be called when
00069 * an interrupt occurs. This function does not save and restore the processor
00070 * context such that the user must provide this processing.
00071 *
00072 * The driver handles the following interrupts:
00073 * - Data Transmit Register/FIFO Empty
00074 * - Data Transmit Register/FIFO Underrun
00075 * - Data Receive Register/FIFO Overrun
00076 * - Mode Fault Error
00077 * - Slave Mode Fault Error
00078 *
00079 * The Data Transmit Register/FIFO Empty interrupt indicates that the SPI device
00080 * has transmitted all the data available to transmit, and now its data register
00081 * (or FIFO) is empty. The driver uses this interrupt to indicate progress while
00082 * sending data. The driver may have more data to send, in which case the data
00083 * transmit register (or FIFO) is filled for subsequent transmission. When this
00084 * interrupt arrives and all the data has been sent, the driver invokes the
status
00085 * callback with a value of XST_SPI_TRANSFER_DONE to inform the upper layer
00086 * software that all data has been sent.
00087 *
00088 * The Data Transmit Register/FIFO Underrun interrupt indicates that, as slave,
00089 * the SPI device was required to transmit but there was no data available to
00090 * transmit in the transmit register (or FIFO). This may not be an error if the
00091 * master is not expecting data, but in the case where the master is expecting
00092 * data this serves as a notification of such a condition. The driver reports
00093 * this condition to the upper layer software through the status handler.

00094 *
00095 * The Data Receive Register/FIFO Overrun interrupt indicates that the SPI
device
00096 * received data and subsequently dropped the data because the data receive
00097 * register (or FIFO) was full. The interrupt applies to both master and slave
00098 * operation. The driver reports this condition to the upper layer software
00099 * through the status handler. This likely indicates a problem with the higher
00100 * layer protocol, or a problem with the slave performance.
00101 *
00102 * The Mode Fault Error interrupt indicates that while configured as a master,
00103 * the device was selected as a slave by another master. This can be used by the
00104 * application for arbitration in a multimaster environment or to indicate a
00105 * problem with arbitration. When this interrupt occurs, the driver invokes the
00106 * status callback with a status value of XST_SPI_MODE_FAULT. It is up to the
00107 * application to resolve the conflict.
00108 *
00109 * The Slave Mode Fault Error interrupt indicates that a slave device was
00110 * selected as a slave by a master, but the slave device was disabled. This can
00111 * be used during system debugging or by the slave application to learn when the
00112 * slave application has not prepared for a master operation in a timely
fashion.
00113 * This likely indicates a problem with the higher layer protocol, or a problem
00114 * with the slave performance.
00115 *
00116 * Note that during the FPGA implementation process, the interrupt registers of
00117 * the IPIF can be parameterized away. This driver is currently dependent on
00118 * those interrupt registers and will not function without them.
00119 *
00120 * **Polled Operation**
00121 *
00122 * Currently there is no support for polled operation.
00123 *
00124 * **Device Busy**
00125 *
00126 * Some operations are disallowed when the device is busy. The driver tracks
00127 * whether a device is busy. The device is considered busy when a data transfer
00128 * request is outstanding, and is considered not busy only when that transfer
00129 * completes (or is aborted with a mode fault error). This applies to both
00130 * master and slave devices.
00131 *
00132 * **Device Configuration**
00133 *
00134 * The device can be configured in various ways during the FPGA implementation
00135 * process. Configuration parameters are stored in the xspi_g.c file. A table
00136 * is defined where each entry contains configuration information for an SPI
00137 * device. This information includes such things as the base address of the
00138 * memory-mapped device, the base address of the IPIF module within the device,
00139 * the number of slave select bits in the device, and whether the device has
00140 * FIFOs and is configured as slave-only.
00141 *
00142 * **RTOS Independence**
00143 *

```

00144 * This driver is intended to be RTOS and processor independent.  It works
00145 * with physical addresses only.  Any needs for dynamic memory management,
00146 * threads or thread mutual exclusion, virtual memory, or cache control must
00147 * be satisfied by the layer above this driver.
00148 *
00149 * <pre>
00150 * MODIFICATION HISTORY:
00151 *
00152 * Ver    Who   Date      Changes
00153 * -----
00154 * 1.00a rpm  10/11/01 First release
00155 * 1.00b jhl  03/14/02 Repartitioned driver for smaller files.
00156 * </pre>
00157 *
00158 *****/
00159
00160 #ifndef XSPI_H /* prevent circular inclusions */
00161 #define XSPI_H /* by using protection macros */
00162
00163 /***** Include Files *****/
00164
00165 #include "xbasic_types.h"
00166 #include "xstatus.h"
00167
00168 /***** Constant Definitions *****/
00169
00170 /** @name Configuration options
00171  *
00172  * The following options may be specified or retrieved for the device and
00173  * enable/disable additional features of the SPI.  Each of the options
00174  * are bit fields, so more than one may be specified.
00175  *
00176  * @{
00177  */
00178 /**
00179  * <pre>
00180  * The Master option configures the SPI device as a master.  By default, the
00181  * device is a slave.
00182  *
00183  * The Active Low Clock option configures the device's clock polarity.  Setting
00184  * this option means the clock is active low and the SCK signal idles high.  By
00185  * default, the clock is active high and SCK idles low.
00186  *
00187  * The Clock Phase option configures the SPI device for one of two transfer
00188  * formats.  A clock phase of 0, the default, means data is valid on the first
00189  * SCK edge (rising or falling) after the slave select (SS) signal has been
00190  * asserted.  A clock phase of 1 means data is valid on the second SCK edge
00191  * (rising or falling) after SS has been asserted.
00192  *
00193  * The Loopback option configures the SPI device for loopback mode.  Data is
00194  * looped back from the transmitter to the receiver.
00195  *

```



```

00196 * The Manual Slave Select option, which is default, causes the device not
00197 * to automatically drive the slave select. The driver selects the device
00198 * at the start of a transfer and deselects it at the end of a transfer.
00199 * If this option is off, then the device automatically toggles the slave
00200 * select signal between bytes in a transfer.
00201 * </pre>
00202 */
00203 #define XSP_MASTER_OPTION                0x1
00204 #define XSP_CLK_ACTIVE_LOW_OPTION       0x2
00205 #define XSP_CLK_PHASE_1_OPTION          0x4
00206 #define XSP_LOOPBACK_OPTION              0x8
00207 #define XSP_MANUAL_SSELECT_OPTION        0x10
00208 /*@}*/
00209
00210 /***** Type Definitions *****/
00211
00212 /**
00213 * The handler data type allows the user to define a callback function to
00214 * handle the asynchronous processing of the SPI driver. The application using
00215 * this driver is expected to define a handler of this type to support
interrupt
00216 * driven mode. The handler executes in an interrupt context such that minimal
00217 * processing should be performed.
00218 *
00219 * @param CallbackRef A callback reference passed in by the upper layer when
00220 * setting the callback functions, and passed back to the
00221 * upper layer when the callback is invoked. Its type is
00222 * unimportant to the driver component, so it is a void
00223 * pointer.
00224 * @param StatusEvent Indicates one or more status events that occurred. See
00225 * the XSpi_SetStatusHandler() for details on the status
00226 * events that can be passed in the callback.
00227 * @param ByteCount Indicates how many bytes of data were successfully
00228 * transferred. This may be less than the number of
bytes
00229 * requested if the status event indicates an error.
00230 */
00231 typedef void (*XSpi_StatusHandler)(void *CallbackRef, Xuint32 StatusEvent,
00232                                     unsigned int ByteCount);
00233
00234 /**
00235 * XSpi statistics
00236 */
00237 typedef struct
00238 {
00239     Xuint32 ModeFaults;           /**< Number of mode fault errors */
00240     Xuint32 XmitUnderruns;        /**< Number of transmit underruns */
00241     Xuint32 RecvOverruns;        /**< Number of receive overruns */
00242     Xuint32 SlaveModeFaults;     /**< Number of selects as a slave while
disabled */
00243     Xuint32 BytesTransferred;    /**< Number of bytes transferred */

```

```

00244     Xuint32 NumInterrupts;          /**< Number of transmit/receive interrupts */
00245 } XSpi_Stats;
00246
00247 /**
00248  * This typedef contains configuration information for the device.
00249  */
00250 typedef struct
00251 {
00252     Xuint16 DeviceId;                /**< Unique ID of device */
00253     Xuint32 BaseAddress;            /**< Base address of the device */
00254
00255     /* Device capabilities */
00256     Xboolean HasFifos;              /**< Does device have FIFOs? */
00257     Xboolean SlaveOnly;             /**< Is the device slave only? */
00258     Xuint8 NumSlaveBits;           /**< Number of slave select bits on the device */
00259 } XSpi_Config;
00260
00261 /**
00262  * The XSpi driver instance data. The user is required to allocate a
00263  * variable of this type for every SPI device in the system. A pointer
00264  * to a variable of this type is then passed to the driver API functions.
00265  */
00266 typedef struct
00267 {
00268     XSpi_Stats Stats;                /* Statistics */
00269     Xuint32 BaseAddr;                /* Base address of device (IPIF) */
00270     Xuint32 IsReady;                 /* Device is initialized and ready */
00271     Xuint32 IsStarted;               /* Device has been started */
00272     Xboolean HasFifos;               /* Device is configured with FIFOs or not */
00273     Xboolean SlaveOnly;              /* Device is configured to be slave only */
00274     Xuint8 NumSlaveBits;             /* Number of slave selects for this device */
00275     Xuint32 SlaveSelectMask;         /* Mask that matches the number of SS bits */
00276     Xuint32 SlaveSelectReg;         /* Slave select register */
00277
00278     Xuint8 *SendBufferPtr;           /* Buffer to send (state) */
00279     Xuint8 *RecvBufferPtr;          /* Buffer to receive (state) */
00280     unsigned int RequestedBytes;     /* Number of bytes to transfer (state) */
00281     unsigned int RemainingBytes;     /* Number of bytes left to transfer (state) */
00282     /*
00283     *
00284     */
00282     Xboolean IsBusy;                 /* A transfer is in progress (state) */
00283
00284     XSpi_StatusHandler StatusHandler;
00285     void *StatusRef;                 /* Callback reference for status handler */
00286
00287 } XSpi;
00288
00289 /***** Macros (Inline Functions) Definitions *****/
00290
00291
00292 /***** Function Prototypes *****/

```

```

00293 /*
00294  * required functions, in xspi.c
00295  */
00296 XStatus XSpi_Initialize(XSpi *InstancePtr, Xuint16 DeviceId);
00297
00298 XStatus XSpi_Start(XSpi *InstancePtr);
00299 XStatus XSpi_Stop(XSpi *InstancePtr);
00300
00301 void XSpi_Reset(XSpi *InstancePtr);
00302
00303 XStatus XSpi_SetSlaveSelect(XSpi *InstancePtr, Xuint32 SlaveMask);
00304 Xuint32 XSpi_GetSlaveSelect(XSpi *InstancePtr);
00305
00306 XStatus XSpi_Transfer(XSpi *InstancePtr, Xuint8 *SendBufPtr, Xuint8
*RecvBufPtr,
00307                     unsigned int ByteCount);
00308
00309 void XSpi_SetStatusHandler(XSpi *InstancePtr, void *CallBackRef,
00310                          XSpi_StatusHandler FuncPtr);
00311 void XSpi_InterruptHandler(void *InstancePtr);
00312 XSpi_Config *XSpi_LookupConfig(Xuint16 DeviceId);
00313
00314
00315 /*
00316  * functions for selftest, in xspi_selftest.c
00317  */
00318 XStatus XSpi_SelfTest(XSpi *InstancePtr);
00319
00320 /*
00321  * functions for statistics, in xspi_stats.c
00322  */
00323 void XSpi_GetStats(XSpi *InstancePtr, XSpi_Stats *StatsPtr);
00324 void XSpi_ClearStats(XSpi *InstancePtr);
00325
00326 /*
00327  * functions for options, in xspi_options.c
00328  */
00329 XStatus XSpi_SetOptions(XSpi *InstancePtr, Xuint32 Options);
00330 Xuint32 XSpi_GetOptions(XSpi *InstancePtr);
00331
00332 #endif /* end of protection macro */

```

XSpi_Stats Struct Reference

```
#include <xspi.h>
```

Detailed Description

XSpi statistics

Data Fields

Xuint32 ModeFaults

Xuint32 XmitUnderruns

Xuint32 RecvOverruns

Xuint32 SlaveModeFaults

Xuint32 BytesTransferred

Xuint32 NumInterrupts

Field Documentation

Xuint32 XSpi_Stats::BytesTransferred

Number of bytes transferred

Xuint32 XSpi_Stats::ModeFaults

Number of mode fault errors

Xuint32 XSpi_Stats::NumInterrupts

Number of transmit/receive interrupts

Xuint32 XSpi_Stats::RecvOverruns

Number of receive overruns

Xuint32 XSpi_Stats::SlaveModeFaults

Number of selects as a slave while disabled

Xuint32 XSpi_Stats::XmitUnderruns

Number of transmit underruns

The documentation for this struct was generated from the following file:

- [spi/v1_00_b/src/xspi.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XSpi_Config Struct Reference

```
#include <xspi.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 BaseAddress

Xboolean HasFifos

Xboolean SlaveOnly

Xuint8 NumSlaveBits

Field Documentation

Xuint32 XSpi_Config::BaseAddress

Base address of the device

Xuint16 XSpi_Config::DeviceId

Unique ID of device

Xboolean XSpi_Config::HasFifos

Does device have FIFOs?

Xuint8 XSpi_Config::NumSlaveBits

Number of slave select bits on the device

Xboolean XSpi_Config::SlaveOnly

Is the device slave only?

The documentation for this struct was generated from the following file:

- [spi/v1_00_b/src/xspi.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

spi/v1_00_b/src/xspi.c File Reference

Detailed Description

Contains required functions of the **XSpi** driver component. See [xspi.h](#) for a detailed description of the device and driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	10/11/01	First release
1.00b	jhl	03/14/02	Repartitioned driver for smaller files.
1.00b	rpm	04/25/02	Collapsed IPIF and reg base addresses into one
1.00b	rmm	05/14/03	Fixed diab compiler warnings relating to asserts

```
#include "xparameters.h"
#include "xspi.h"
#include "xspi_i.h"
#include "xipif_v1_23_b.h"
#include "xio.h"
```

Functions

XStatus XSpi_Initialize (**XSpi** *InstancePtr, **Xuint16** DeviceId)

XStatus XSpi_Start (**XSpi** *InstancePtr)

XStatus XSpi_Stop (**XSpi** *InstancePtr)

void **XSpi_Reset** (**XSpi** *InstancePtr)

XStatus XSpi_Transfer (**XSpi** *InstancePtr, **Xuint8** *SendBufPtr, **Xuint8** *RecvBufPtr, unsigned int
ByteCount)

XStatus XSpi_SetSlaveSelect (**XSpi** *InstancePtr, **Xuint32** SlaveMask)

Xuint32 XSpi_GetSlaveSelect (**XSpi** *InstancePtr)

void **XSpi_SetStatusHandler** (**XSpi** *InstancePtr, void *CallBackRef, **XSpi_StatusHandler** FuncPtr)

void **XSpi_InterruptHandler** (void *InstancePtr)

void **XSpi_Abort** (**XSpi** *InstancePtr)

XSpi_Config * **XSpi_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
void XSpi_Abort( XSpi * InstancePtr)
```

Aborts a transfer in progress by setting the stop bit in the control register, then resetting the FIFOs if present. The byte counts are cleared and the busy flag is set to false.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

This function does a read/modify/write of the control register. The user of this function needs to take care of critical sections.

```
Xuint32 XSpi_GetSlaveSelect( XSpi * InstancePtr)
```

Gets the current slave select bit mask for the SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

The value returned is a 32-bit mask with a 1 in the bit position of the slave currently selected. The value may be zero if no slaves are selected.

Note:

None.

```
XStatus XSpi_Initialize( XSpi * InstancePtr,  
                        Xuint16 DeviceId  
                        )
```

Initializes a specific **XSpi** instance such that the driver is ready to use.

The state of the device after initialization is:

- Device is disabled
- Slave mode
- Active high clock polarity
- Clock phase 0

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XSpi** instance. Passing in a device id associates the generic **XSpi** instance to a specific device, as chosen by the caller or application developer.

Returns:

The return value is XST_SUCCESS if successful. On error, a code indicating the specific error is returned.

Possible error codes are:

- XST_DEVICE_IS_STARTED if the device is started. It must be stopped to re-initialize.
- XST_DEVICE_NOT_FOUND if the device was not found in the configuration such that initialization could not be accomplished.

Note:

None.

void XSpi_InterruptHandler(void * *InstancePtr*)

The interrupt handler for SPI interrupts. This function must be connected by the user to an interrupt source. This function does not save and restore the processor context such that the user must provide this processing.

The interrupts that are handled are:

- Mode Fault Error. This interrupt is generated if this device is selected as a slave when it is configured as a master. The driver aborts any data transfer that is in progress by resetting FIFOs (if present) and resetting its buffer pointers. The upper layer software is informed of the error.
- Data Transmit Register (FIFO) Empty. This interrupt is generated when the transmit register or FIFO is empty. The driver uses this interrupt during a transmission to continually send/receive data until there is no more data to send/receive.
- Data Transmit Register (FIFO) Underrun. This interrupt is generated when the SPI device, when configured as a slave, attempts to read an empty DTR/FIFO. An empty DTR/FIFO usually means that software is not giving the device data in a timely manner. No action is taken by the driver other than to inform the upper layer software of the error.
- Data Receive Register (FIFO) Overrun. This interrupt is generated when the SPI device attempts to write a received byte to an already full DRR/FIFO. A full DRR/FIFO usually means software is not emptying the data

in a timely manner. No action is taken by the driver other than to inform the upper layer software of the error.

- Slave Mode Fault Error. This interrupt is generated if a slave device is selected as a slave while it is disabled. No action is taken by the driver other than to inform the upper layer software of the error.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

The slave select register is being set to deselect the slave when a transfer is complete. This is being done regardless of whether it is a slave or a master since the hardware does not drive the slave select as a slave.

XSpi_Config* XSpi_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found. See **xspi.h** for the definition of **XSpi_Config**.

Note:

None.

void XSpi_Reset(XSpi * InstancePtr)

Resets the SPI device. Reset must only be called after the driver has been initialized. The configuration of the device after reset is the same as its configuration after initialization. Refer to the XSpi_Initialize function for more details. This is a hard reset of the device. Any data transfer that is in progress is aborted.

The upper layer software is responsible for re-configuring (if necessary) and restarting the SPI device after the reset.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus XSpi_SetSlaveSelect( XSpi * InstancePtr,  
                             Xuint32 SlaveMask  
                             )
```

Selects or deselect the slave with which the master communicates. Each slave that can be selected is represented in the slave select register by a bit. The argument passed to this function is the bit mask with a 1 in the bit position of the slave being selected. Only one slave can be selected.

The user is not allowed to deselect the slave while a transfer is in progress. If no transfer is in progress, the user can select a new slave, which implicitly deselects the current slave. In order to explicitly deselect the current slave, a zero can be passed in as the argument to the function.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

SlaveMask is a 32-bit mask with a 1 in the bit position of the slave being selected. Only one slave can be selected. The SlaveMask can be zero if the slave is being deselected.

Returns:

XST_SUCCESS if the slave is selected or deselected successfully. Otherwise, returns:

- XST_DEVICE_BUSY if a transfer is in progress, slave cannot be changed
- XST_SPI_TOO_MANY_SLAVES if more than one slave is being selected.

Note:

This function only sets the slave which will be selected when a transfer occurs. The slave is not selected when the SPI is idle. The slave select has no affect when the device is configured as a slave.

```
void XSpi_SetStatusHandler( XSpi * InstancePtr,  
                            void * CallbackRef,  
                            XSpi_StatusHandler FuncPtr  
                            )
```

Sets the status callback function, the status handler, which the driver calls when it encounters conditions that should be reported to the higher layer software. The handler executes in an interrupt context, so it must minimize the amount of processing performed such as transferring data to a thread context. One of the following status events is passed to the status handler.

XST_SPI_MODE_FAULT	A mode fault error occurred, meaning another master tried to select this device as a slave when this device was configured to be a master. Any transfer in progress is aborted.
XST_SPI_TRANSFER_DONE	The requested data transfer is done
XST_SPI_TRANSMIT_UNDERRUN	As a slave device, the master clocked data but there were none available in the transmit

register/FIFO. This typically means the slave application did not issue a transfer request fast enough, or the processor/driver could not fill the transmit register/FIFO fast enough.

XST_SPI_RECEIVE_OVERRUN

The SPI device lost data. Data was received but the receive data register/FIFO was full. This indicates that the device is receiving data

XST_SPI_SLAVE_MODE_FAULT

faster than the processor/driver can consume it. A slave SPI device was selected as a slave while it was disabled. This indicates the master is already transferring data (which is being dropped until the slave application issues a transfer).

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context, so it should do its work quickly and queue potentially time-consuming work to a task-level thread.

XStatus XSpi_Start(XSpi * InstancePtr)

This function enables interrupts for the SPI device. It is up to the user to connect the SPI interrupt handler to the interrupt controller before this Start function is called. The GetIntrHandler function is used for that purpose. If the device is configured with FIFOs, the FIFOs are reset at this time.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if the device is successfully started, or XST_DEVICE_IS_STARTED if the device was already started.

Note:

None.

XStatus XSpi_Stop(XSpi * InstancePtr)

This function stops the SPI device by disabling interrupts and disabling the device itself. Interrupts are disabled only within the device itself. If desired, the caller is responsible for disabling interrupts in the interrupt controller and disconnecting the interrupt handler from the interrupt controller.

If the device is in progress of transferring data on the SPI bus, this function returns a status indicating the device is busy. The user will be notified via the status handler when the transfer is complete, and at that time can again try to stop the device. As a master, we do not allow the device to be stopped while a transfer is in progress because the slave may be left in a bad state. As a slave, we do not allow the device to be stopped while a transfer is in progress because the master is not done with its transfer yet.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if the device is successfully started, or XST_DEVICE_BUSY if a transfer is in progress and cannot be stopped.

Note:

This function makes use of internal resources that are shared between the **XSpi_Stop()** and **XSpi_SetOptions()** functions. So if one task might be setting device options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

```
XStatus XSpi_Transfer( XSpi *      InstancePtr,
                      Xuint8 *    SendBufPtr,
                      Xuint8 *    RecvBufPtr,
                      unsigned int ByteCount
                      )
```

Transfers the specified data on the SPI bus. If the SPI device is configured to be a master, this function initiates bus communication and sends/receives the data to/from the selected SPI slave. If the SPI device is configured to be a slave, this function prepares the data to be sent/received when selected by a master. For every byte sent, a byte is received.

The caller has the option of providing two different buffers for send and receive, or one buffer for both send and receive, or no buffer for receive. The receive buffer must be at least as big as the send buffer to prevent unwanted memory writes. This implies that the byte count passed in as an argument must be the smaller of the two buffers if they differ in size. Here are some sample usages:

```
XSpi_Transfer(InstancePtr, SendBuf, RecvBuf, ByteCount)
```

The caller wishes to send and receive, and provides two different buffers for send and receive.

```
XSpi_Transfer(InstancePtr, SendBuf, NULL, ByteCount)
```

The caller wishes only to send and does not care about the received data. The driver ignores the received data in this case.

XSpi_Transfer(InstancePtr, SendBuf, SendBuf, ByteCount)

The caller wishes to send and receive, but provides the same buffer for doing both. The driver sends the data and overwrites the send buffer with received data as it transfers the data.

XSpi_Transfer(InstancePtr, RecvBuf, RecvBuf, ByteCount)

The caller wishes to only receive and does not care about sending data. In this case, the caller must still provide a send buffer, but it can be the same as the receive buffer if the caller does not care what it sends. The device must send N bytes of data if it wishes to receive N bytes of data.

Although this function takes a buffer as an argument, the driver can only transfer a limited number of bytes at time. It transfers only one byte at a time if there are no FIFOs, or it can transfer the number of bytes up to the size of the FIFO. A call to this function only starts the transfer, then subsequent transfer of the data is performed by the interrupt service routine until the entire buffer has been transferred. The status callback function is called when the entire buffer has been sent/received.

This function is non-blocking. As a master, the SetSlaveSelect function must be called prior to this function.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

SendBufPtr is a pointer to a buffer of data which is to be sent. This buffer must not be NULL.

RecvBufPtr is a pointer to a buffer which will be filled with received data. This argument can be NULL if the caller does not wish to receive data.

ByteCount contains the number of bytes to send/receive. The number of bytes received always equals the number of bytes sent.

Returns:

XST_SUCCESS if the buffers are successfully handed off to the driver for transfer. Otherwise, returns:

- o XST_DEVICE_IS_STOPPED if the device must be started before transferring data.
- o XST_DEVICE_BUSY indicates that a data transfer is already in progress. This is determined by the driver.
- o XST_SPI_NO_SLAVE indicates the device is configured as a master and a slave has not yet been selected.

Note:

This function is not thread-safe. The higher layer software must ensure that no two threads are transferring data on the SPI bus at the same time.

spi/v1_00_b/src/xspi_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xspi_i.h,v 1.4 2002/05/02 20:31:10 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file spi/v1_00_b/src/xspi_i.h
00026 *
00027 * This header file contains internal identifiers. It is intended for internal
00028 * use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date      Changes
00034 * -----
00035 * 1.00a  rpm    10/11/01  First release
00036 * 1.00b  jhl    03/14/02  Repartitioned driver for smaller files.
00037 * 1.00b  rpm    04/24/02  Moved register definitions to xspi_l.h
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XSPI_I_H /* prevent circular inclusions */
```



```

00043 #define XSPI_I_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048 #include "xspi_1.h"
00049
00050 /***** Constant Definitions *****/
00051
00052 /*
00053  * IPIF SPI device interrupt mask. This mask is for the Device Interrupt
00054  * Register within the IPIF.
00055  */
00056 #define XSP_IPIF_SPI_MASK    0x4UL
00057
00058 #define XSP_IPIF_DEVICE_INTR_COUNT    3    /* Number of interrupt sources */
00059 #define XSP_IPIF_IP_INTR_COUNT    6    /* Number of SPI interrupts
00060                                     * note that there are 7 interrupts in
00061                                     * the h/w but s/w does not use the
00062                                     * half empty which only exists when
00063                                     * there are FIFOs, this allows the
IPIF
00064                                     * self test to pass with or without
00065                                     * FIFOs */
00066 /*
00067  * IPIF SPI IP interrupt masks. These masks are for the IP Interrupt Register
00068  * within the IPIF.
00069  */
00070 #define XSP_INTR_MODE_FAULT_MASK    0x1UL    /* Mode fault error */
00071 #define XSP_INTR_SLAVE_MODE_FAULT_MASK    0x2UL    /* Selected as slave while
00072                                     * disabled */
00073 #define XSP_INTR_TX_EMPTY_MASK    0x4UL    /* DTR/TxFIFO is empty */
00074 #define XSP_INTR_TX_UNDERRUN_MASK    0x8UL    /* DTR/TxFIFO was underrun */
00075 #define XSP_INTR_RX_FULL_MASK    0x10UL    /* DRR/RxFIFO is full */
00076 #define XSP_INTR_RX_OVERRUN_MASK    0x20UL    /* DRR/RxFIFO was overrun */
00077 #define XSP_INTR_TX_HALF_EMPTY_MASK    0x40UL    /* TxFIFO is half empty */
00078
00079 /*
00080  * The interrupts we want at startup. We add the TX_EMPTY interrupt in later
00081  * when we're getting ready to transfer data. The others we don't care
00082  * about for now.
00083  */
00084 #define XSP_INTR_DFT_MASK    (XSP_INTR_MODE_FAULT_MASK |    \
00085                               XSP_INTR_TX_UNDERRUN_MASK |    \
00086                               XSP_INTR_RX_OVERRUN_MASK |    \
00087                               XSP_INTR_SLAVE_MODE_FAULT_MASK)
00088
00089 /***** Type Definitions *****/
00090
00091 /***** Macros (Inline Functions) Definitions *****/
00092
00093

```

```

00094 /*****
00095 /*
00096 *
00097 * Clear the statistics of the driver instance.
00098 *
00099 * @param InstancePtr is a pointer to the XSpi instance to be worked on.
00100 *
00101 * @return None.
00102 *
00103 * @note
00104 *
00105 * Signature: void XSpi_mClearStats(XSpi *InstancePtr)
00106 *
00107 *****/
00108 #define XSpi_mClearStats(InstancePtr) \
00109 { \
00110     InstancePtr->Stats.ModeFaults = 0; \
00111     InstancePtr->Stats.XmitUnderruns = 0; \
00112     InstancePtr->Stats.RecvOverruns = 0; \
00113     InstancePtr->Stats.SlaveModeFaults = 0; \
00114     InstancePtr->Stats.BytesTransferred = 0;\
00115     InstancePtr->Stats.NumInterrupts = 0; \
00116 }
00117
00118 /***** Function Prototypes *****/
00119
00120 void XSpi_Abort(XSpi *InstancePtr);
00121
00122 /***** Variable Definitions *****/
00123
00124 extern XSpi_Config XSpi_ConfigTable[];
00125
00126 #endif /* end of protection macro */
00127

```

spi/v1_00_b/src/xspi_i.h File Reference

Detailed Description

This header file contains internal identifiers. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	10/11/01	First release
1.00b	jhl	03/14/02	Repartitioned driver for smaller files.
1.00b	rpm	04/24/02	Moved register definitions to xspi_1.h

```
#include "xbasic_types.h"  
#include "xspi_1.h"
```

[Go to the source code of this file.](#)

Functions

```
void XSpi\_Abort (XSpi *InstancePtr)
```

Variables

```
XSpi\_Config XSpi\_ConfigTable []
```

Function Documentation

```
void XSpi_Abort( XSpi * InstancePtr)
```

Aborts a transfer in progress by setting the stop bit in the control register, then resetting the FIFOs if present. The byte counts are cleared and the busy flag is set to false.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

This function does a read/modify/write of the control register. The user of this function needs to take care of critical sections.

Variable Documentation

```
XSpi_Config XSpi_ConfigTable[]( )
```

This table contains configuration information for each SPI device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

spi/v1_00_b/src/xspi_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xspi.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00b	rpm	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XSpi_mSetControlReg(BaseAddress, Mask)  
#define XSpi_mGetControlReg(BaseAddress)  
#define XSpi_mGetStatusReg(BaseAddress)  
#define XSpi_mSetSlaveSelectReg(BaseAddress, Mask)  
#define XSpi_mGetSlaveSelectReg(BaseAddress)  
#define XSpi_mEnable(BaseAddress)  
#define XSpi_mDisable(BaseAddress)  
#define XSpi_mSendByte(BaseAddress, Data)
```

```
#define XSpi_mRecvByte(BaseAddress)
```

Define Documentation

```
#define XSpi_mDisable( BaseAddress )
```

Disable the device. Preserves the current contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XSpi_mEnable( BaseAddress )
```

Enable the device and uninhibit master transactions. Preserves the current contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XSpi_mGetControlReg( BaseAddress )
```

Get the contents of the control register. Use the XSP_CR_* constants defined above to interpret the bit-mask returned.

Parameters:

BaseAddress is the base address of the device

Returns:

A 16-bit value representing the contents of the control register.

Note:

None.

```
#define XSpi_mGetSlaveSelectReg( BaseAddress )
```

Get the contents of the slave select register. Each bit in the mask corresponds to a slave select line. Only one slave should be selected at any one time.

Parameters:

BaseAddress is the base address of the device

Returns:

The 32-bit value in the slave select register

Note:

None.

```
#define XSpi_mGetStatusReg( BaseAddress )
```

Get the contents of the status register. Use the XSP_SR_* constants defined above to interpret the bit-mask returned.

Parameters:

BaseAddress is the base address of the device

Returns:

An 8-bit value representing the contents of the status register.

Note:

None.

```
#define XSpi_mRecvByte( BaseAddress )
```

Receive one byte from the device's receive FIFO/register. It is assumed that the byte is already available.

Parameters:

BaseAddress is the base address of the device

Returns:

The byte retrieved from the receive FIFO/register.

Note:

None.

```
#define XSpi_mSendByte( BaseAddress,  
                        Data          )
```


Send one byte to the currently selected slave. The byte that is received from the slave is saved in the receive FIFO/register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XSpi_mSetControlReg( BaseAddress,  
                             Mask          )
```

Set the contents of the control register. Use the XSP_CR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device

Mask is the 16-bit value to write to the control register

Returns:

None.

Note:

None.

```
#define XSpi_mSetSlaveSelectReg( BaseAddress,  
                                 Mask          )
```

Set the contents of the slave select register. Each bit in the mask corresponds to a slave select line. Only one slave should be selected at any one time.

Parameters:

BaseAddress is the base address of the device

Mask is the 32-bit value to write to the slave select register

Returns:

None.

Note:

None.

spi/v1_00_b/src/xspi_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xspi_l.h,v 1.1 2002/05/02 20:27:52 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file spi/v1_00_b/src/xspi_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  High-level driver functions
00029 * are defined in xspi.h.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date        Changes
00035 * -----
00036 * 1.00b rpm  04/24/02  First release
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XSPI_L_H /* prevent circular inclusions */
00042 #define XSPI_L_H /* by using protection macros */
```

```

00043
00044 /***** Include Files *****/
00045
00046 #include "xbasic_types.h"
00047 #include "xio.h"
00048
00049 /***** Constant Definitions *****/
00050
00051 /*
00052  * Offset from the device base address (IPIF) to the IP registers.
00053  */
00054 #define XSP_REGISTER_OFFSET      0x60
00055
00056 /*
00057  * Register offsets for the SPI. Each register except the CR & SSR is 8 bits,
00058  * so add 3 to the word-offset to get the LSB (in a big-endian system).
00059  */
00060 #define XSP_CR_OFFSET      (XSP_REGISTER_OFFSET + 0x2)      /* 16-bit Control */
00061 #define XSP_SR_OFFSET      (XSP_REGISTER_OFFSET + 0x4 + 3)  /* Status */
00062 #define XSP_DTR_OFFSET     (XSP_REGISTER_OFFSET + 0x8 + 3)  /* Data transmit */
00063 #define XSP_DRR_OFFSET     (XSP_REGISTER_OFFSET + 0xC + 3)  /* Data receive */
00064 #define XSP_SSR_OFFSET     (XSP_REGISTER_OFFSET + 0x10)     /* 32-bit slave select
*/
00065 #define XSP_TFO_OFFSET     (XSP_REGISTER_OFFSET + 0x14 + 3) /* Transmit FIFO
occupancy */
00066 #define XSP_RFO_OFFSET     (XSP_REGISTER_OFFSET + 0x18 + 3) /* Receive FIFO
occupancy */
00067
00068 /*
00069  * SPI Control Register (CR) masks
00070  */
00071 #define XSP_CR_LOOPBACK_MASK      0x1  /* Local loopback mode */
00072 #define XSP_CR_ENABLE_MASK       0x2  /* System enable */
00073 #define XSP_CR_MASTER_MODE_MASK  0x4  /* Enable master mode */
00074 #define XSP_CR_CLK_POLARITY_MASK 0x8  /* Clock polarity high or low */
00075 #define XSP_CR_CLK_PHASE_MASK   0x10  /* Clock phase 0 or 1 */
00076 #define XSP_CR_TXFIFO_RESET_MASK 0x20  /* Reset transmit FIFO */
00077 #define XSP_CR_RXFIFO_RESET_MASK 0x40  /* Reset receive FIFO */
00078 #define XSP_CR_MANUAL_SS_MASK    0x80  /* Manual slave select assertion */
00079 #define XSP_CR_TRANS_INHIBIT_MASK 0x100 /* Master transaction inhibit */
00080
00081 /*
00082  * SPI Status Register (SR) masks
00083  */
00084 #define XSP_SR_RX_EMPTY_MASK     0x1  /* Receive register/FIFO is empty */
00085 #define XSP_SR_RX_FULL_MASK      0x2  /* Receive register/FIFO is full */
00086 #define XSP_SR_TX_EMPTY_MASK     0x4  /* Transmit register/FIFO is empty */
00087 #define XSP_SR_TX_FULL_MASK      0x8  /* Transmit register/FIFO is full */
00088 #define XSP_SR_MODE_FAULT_MASK   0x10 /* Mode fault error */
00089
00090 /*
00091  * SPI Transmit FIFO Occupancy (TFO) mask. The binary value plus one yields

```

```

00092 * the occupancy.
00093 */
00094 #define XSP_TFO_MASK          0x1F
00095
00096 /*
00097 * SPI Receive FIFO Occupancy (RFO) mask. The binary value plus one yields
00098 * the occupancy.
00099 */
00100 #define XSP_RFO_MASK          0x1F
00101
00102
00103 /***** Type Definitions *****/
00104
00105 /***** Macros (Inline Functions) Definitions *****/
00106
00107
00108 /*****
00109 *
00110 * Low-level driver macros. The list below provides signatures to help the
00111 * user use the macros.
00112 *
00113 * void XSpi_mSetControlReg(Xuint32 BaseAddress, Xuint16 Mask)
00114 * Xuint16 XSpi_mGetControlReg(Xuint32 BaseAddress)
00115 * Xuint8 XSpi_mGetStatusReg(Xuint32 BaseAddress)
00116 *
00117 * void XSpi_mSetSlaveSelectReg(Xuint32 BaseAddress, Xuint32 Mask)
00118 * Xuint32 XSpi_mGetSlaveSelectReg(Xuint32 BaseAddress)
00119 *
00120 * void XSpi_mEnable(Xuint32 BaseAddress)
00121 * void XSpi_mDisable(Xuint32 BaseAddress)
00122 *
00123 * void XSpi_mSendByte(Xuint32 BaseAddress, Xuint8 Data);
00124 * Xuint8 XSpi_mRecvByte(Xuint32 BaseAddress);
00125 *
00126 *****/
00127
00128 /*****
00129 /**
00130 *
00131 * Set the contents of the control register. Use the XSP_CR_* constants defined
00132 * above to create the bit-mask to be written to the register.
00133 *
00134 * @param BaseAddress is the base address of the device
00135 * @param Mask is the 16-bit value to write to the control register
00136 *
00137 * @return None.
00138 *
00139 * @note None.
00140 *
00141 *****/
00142 #define XSpi_mSetControlReg(BaseAddress, Mask) \
00143         XIo_Out16((BaseAddress) + XSP_CR_OFFSET, (Mask))

```

```
00144
00145
00146 /*****
00147 /**
00148 *
00149 * Get the contents of the control register. Use the XSP_CR_* constants defined
00150 * above to interpret the bit-mask returned.
00151 *
00152 * @param   BaseAddress is the base address of the device
00153 *
00154 * @return  A 16-bit value representing the contents of the control register.
00155 *
00156 * @note    None.
00157 *
00158 *****/
00159 #define XSpi_mGetControlReg(BaseAddress) \
00160         XIo_In16((BaseAddress) + XSP_CR_OFFSET)
00161
00162
00163 /*****
00164 /**
00165 *
00166 * Get the contents of the status register. Use the XSP_SR_* constants defined
00167 * above to interpret the bit-mask returned.
00168 *
00169 * @param   BaseAddress is the base address of the device
00170 *
00171 * @return  An 8-bit value representing the contents of the status register.
00172 *
00173 * @note    None.
00174 *
00175 *****/
00176 #define XSpi_mGetStatusReg(BaseAddress) \
00177         XIo_In8((BaseAddress) + XSP_SR_OFFSET)
00178
00179
00180 /*****
00181 /**
00182 *
00183 * Set the contents of the slave select register. Each bit in the mask
00184 * corresponds to a slave select line. Only one slave should be selected at
00185 * any one time.
00186 *
00187 * @param   BaseAddress is the base address of the device
00188 * @param   Mask is the 32-bit value to write to the slave select register
00189 *
00190 * @return  None.
00191 *
00192 * @note    None.
00193 *
00194 *****/
00195 #define XSpi_mSetSlaveSelectReg(BaseAddress, Mask) \
```

```

00196             XIo_Out32((BaseAddress) + XSP_SSR_OFFSET, (Mask))
00197
00198
00199 /*****
00200 /**
00201 *
00202 * Get the contents of the slave select register. Each bit in the mask
00203 * corresponds to a slave select line. Only one slave should be selected at
00204 * any one time.
00205 *
00206 * @param    BaseAddress is the base address of the device
00207 *
00208 * @return   The 32-bit value in the slave select register
00209 *
00210 * @note     None.
00211 *
00212 *****/
00213 #define XSpi_mGetSlaveSelectReg(BaseAddress) \
00214             XIo_In32((BaseAddress) + XSP_SSR_OFFSET)
00215
00216 /*****
00217 /**
00218 *
00219 * Enable the device and uninhibit master transactions. Preserves the current
00220 * contents of the control register.
00221 *
00222 * @param    BaseAddress is the base address of the device
00223 *
00224 * @return   None.
00225 *
00226 * @note     None.
00227 *
00228 *****/
00229 #define XSpi_mEnable(BaseAddress) \
00230 { \
00231     Xuint16 Control; \
00232     Control = XSpi_mGetControlReg((BaseAddress)); \
00233     Control |= XSP_CR_ENABLE_MASK; \
00234     Control &= ~XSP_CR_TRANS_INHIBIT_MASK; \
00235     XSpi_mSetControlReg((BaseAddress), Control); \
00236 }
00237
00238 /*****
00239 /**
00240 *
00241 * Disable the device. Preserves the current contents of the control register.
00242 *
00243 * @param    BaseAddress is the base address of the device
00244 *
00245 * @return   None.
00246 *
00247 * @note     None.

```

```

00248 *
00249 *****/
00250 #define XSpi_mDisable(BaseAddress) \
00251     XSpi_mSetControlReg((BaseAddress), \
00252         XSpi_mGetControlReg((BaseAddress)) & ~XSP_CR_ENABLE_MASK)
00253
00254
00255 /**
00256 **
00257 *
00258 * Send one byte to the currently selected slave. The byte that is received
00259 * from the slave is saved in the receive FIFO/register.
00260 *
00261 * @param   BaseAddress is the base address of the device
00262 *
00263 * @return  None.
00264 *
00265 * @note    None.
00266 *
00267 *****/
00268 #define XSpi_mSendByte(BaseAddress, Data) \
00269     XIo_Out8((BaseAddress) + XSP_DTR_OFFSET, (Data))
00270
00271
00272 /**
00273 **
00274 *
00275 * Receive one byte from the device's receive FIFO/register. It is assumed
00276 * that the byte is already available.
00277 *
00278 * @param   BaseAddress is the base address of the device
00279 *
00280 * @return  The byte retrieved from the receive FIFO/register.
00281 *
00282 * @note    None.
00283 *
00284 *****/
00285 #define XSpi_mRecvByte(BaseAddress) \
00286     XIo_In8((BaseAddress) + XSP_DRR_OFFSET)
00287
00288 /** Function Prototypes *****/
00289
00290 /** Variable Definitions *****/
00291
00292 #endif          /* end of protection macro */
00293

```


spi/v1_00_b/src/xspi_selftest.c File Reference

Detailed Description

This component contains the implementation of selftest functions for the **XSpi** driver component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	2/27/02	First release
1.00b	rpm	04/25/02	Collapsed IPIF and reg base addresses into one

```
#include "xspi.h"  
#include "xspi_i.h"  
#include "xio.h"  
#include "xipif_v1_23_b.h"
```

Functions

XStatus XSpi_SelfTest (XSpi *InstancePtr)

Function Documentation

XStatus XSpi_SelfTest(XSpi * InstancePtr)

Runs a self-test on the driver/device. The self-test is destructive in that a reset of the device is performed in order to check the reset values of the registers and to get the device into a known state. A simple loopback test is also performed to verify that transmit and receive are working properly. The device is changed to master mode for the loopback test, since only a master can initiate a data transfer.

Upon successful return from the self-test, the device is reset.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

XST_SUCCESS if successful, or one of the following error codes otherwise.

- XST_REGISTER_ERROR indicates a register did not read or write correctly
- XST_LOOPBACK_ERROR if a loopback error occurred.

Note:

None.

spi/v1_00_b/src/xspi_stats.c File Reference

Detailed Description

This component contains the implementation of statistics functions for the **XSpi** driver component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	03/14/02	First release
1.00b	rpm	04/25/02	Changed macro naming convention

```
#include "xspi.h"  
#include "xspi_i.h"
```

Functions

```
void XSpi_GetStats (XSpi *InstancePtr, XSpi_Stats *StatsPtr)  
void XSpi_ClearStats (XSpi *InstancePtr)
```

Function Documentation

```
void XSpi_ClearStats( XSpi * InstancePtr)
```

Clears the statistics for the SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

None.

Note:

None.

```
void XSpi_GetStats( XSpi * InstancePtr,  
                   XSpi_Stats * StatsPtr  
                   )
```

Gets a copy of the statistics for an SPI device.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

StatsPtr is a pointer to a **XSpi_Stats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

spi/v1_00_b/src/xspi_options.c File Reference

Detailed Description

Contains functions for the configuration of the **XSpi** driver component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	2/27/02	First release
1.00b	rpm	04/25/02	Collapsed IPIF and reg base addresses into one

```
#include "xspi.h"  
#include "xspi_i.h"  
#include "xio.h"
```

Data Structures

```
struct OptionsMap
```

Functions

XStatus **XSpi_SetOptions** (**XSpi** *InstancePtr, **Xuint32** Options)

Xuint32 **XSpi_GetOptions** (**XSpi** *InstancePtr)

Function Documentation

Xuint32 XSpi_GetOptions(**XSpi** * *InstancePtr*)

This function gets the options for the SPI device. The options control how the device behaves relative to the SPI bus.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Returns:

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. One or more bit values may be contained in the mask. See the bit definitions named XSP_*_OPTIONS in the file **xspi.h**.

Note:

None.

XStatus XSpi_SetOptions(**XSpi** * *InstancePtr*,
 Xuint32 *Options*
)

This function sets the options for the SPI device driver. The options control how the device behaves relative to the SPI bus. The device must be idle rather than busy transferring data before setting these device options.

Parameters:

InstancePtr is a pointer to the **XSpi** instance to be worked on.

Options contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. One or more bit values may be contained in the mask. See the bit definitions named XSP_*_OPTIONS in the file **xspi.h**.

Returns:

XST_SUCCESS if options are successfully set. Otherwise, returns:

- XST_DEVICE_BUSY if the device is currently transferring data. The transfer must complete or be aborted before setting options.
- XST_SPI_SLAVE_ONLY if the caller attempted to configure a slave-only device as a

master.

Note:

This function makes use of internal resources that are shared between the **XSpi_Stop()** and **XSpi_SetOptions()** functions. So if one task might be setting device options options while another is trying to stop the device, the user is required to provide protection of this shared data (typically using a semaphore).

Generated on 29 May 2003 for Xilinx Device Drivers

spi/v1_00_b/src/xspi_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of SPI devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	rpm	10/11/01	First release
1.00b	jhl	03/14/02	Repartitioned driver for smaller files.
1.00b	rpm	04/24/02	Condensed config typedef - got rid of versions and multiple base addresses.

```
#include "xspi.h"  
#include "xparameters.h"
```

Variables

[XSpi_Config](#) [XSpi_ConfigTable](#) [XPAR_XSPI_NUM_INSTANCES]

Variable Documentation

[XSpi_Config](#) [XSpi_ConfigTable](#)[XPAR_XSPI_NUM_INSTANCES]

This table contains configuration information for each SPI device in the system.

tmrctr/v1_00_b/src/xtmrctr.h File Reference

Detailed Description

The Xilinx timer/counter component. This component supports the Xilinx timer/counter. More detailed description of the driver operation can be found in the [xtmrctr.c](#) file.

The Xilinx timer/counter supports the following features:

- Polled mode.
- Interrupt driven mode
- enabling and disabling specific timers
- PWM operation

The driver does not currently support the PWM operation of the device.

The timer counter operates in 2 primary modes, compare and capture. In either mode, the timer counter may count up or down, with up being the default.

Compare mode is typically used for creating a single time period or multiple repeating time periods in the auto reload mode, such as a periodic interrupt. When started, the timer counter loads an initial value, referred to as the compare value, into the timer counter and starts counting down or up. The timer counter expires when it rolls over/under depending upon the mode of counting. An external compare output signal may be configured such that a pulse is generated with this signal when it hits the compare value.

Capture mode is typically used for measuring the time period between external events. This mode uses an external capture input signal to cause the value of the timer counter to be captured. When started, the timer counter loads an initial value, referred to as the compare value,

The timer can be configured to either cause an interrupt when the count reaches the compare value in compare mode or latch the current count value in the capture register when an external input is asserted in capture mode. The external capture input can be enabled/disabled using the `XTmrCtr_SetOptions` function. While in compare mode, it is also possible to drive an external output when the compare value is reached in the count register. The external compare output can be enabled/disabled using the `XTmrCtr_SetOptions` function.

Interrupts

It is the responsibility of the application to connect the interrupt handler of the timer/counter to the interrupt source.

The interrupt handler function, `XTmrCtr_InterruptHandler`, is visible such that the user can connect it to the interrupt source. Note that this interrupt handler does not provide interrupt context save and restore processing, the user must perform this processing.

The driver services interrupts and passes timeouts to the upper layer software through callback functions. The upper layer software must register its callback functions during initialization. The driver requires callback functions for timers.

Note:

The default settings for the timers are:

- Interrupt generation disabled
- Count up mode
- Compare mode
- Hold counter (will not reload the timer)
- External compare output disabled
- External capture input disabled
- Pulse width modulation disabled
- Timer disabled, waits for Start function to be called

A timer counter device may contain multiple timer counters. The symbol `XTC_DEVICE_TIMER_COUNT` defines the number of timer counters in the device. The device currently contains 2 timer counters.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xtmrctr_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XTmrCtr  
struct XTmrCtr_Config
```

```
struct XTmrCtrStats
```

Configuration options

These options are used in **XTmrCtr_SetOptions()** and **XTmrCtr_GetOptions()**

```
#define XTC_ENABLE_ALL_OPTION  
#define XTC_DOWN_COUNT_OPTION  
#define XTC_CAPTURE_MODE_OPTION  
#define XTC_INT_MODE_OPTION  
#define XTC_AUTO_RELOAD_OPTION  
#define XTC_EXT_COMPARE_OPTION
```

Typedefs

```
typedef void(* XTmrCtr_Handler )(void *CallBackRef, Xuint8 TmrCtrNumber)
```

Functions

```
XStatus XTmrCtr_Initialize (XTmrCtr *InstancePtr, Xuint16 DeviceId)  
void XTmrCtr_Start (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
void XTmrCtr_Stop (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
Xuint32 XTmrCtr_GetValue (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
void XTmrCtr_SetResetValue (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber, Xuint32 ResetValue)  
Xuint32 XTmrCtr_GetCaptureValue (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
Xboolean XTmrCtr_IsExpired (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
void XTmrCtr_Reset (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
void XTmrCtr_SetOptions (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber, Xuint32 Options)  
Xuint32 XTmrCtr_GetOptions (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
void XTmrCtr_GetStats (XTmrCtr *InstancePtr, XTmrCtrStats *StatsPtr)  
void XTmrCtr_ClearStats (XTmrCtr *InstancePtr)  
XStatus XTmrCtr_SelfTest (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
void XTmrCtr_SetHandler (XTmrCtr *InstancePtr, XTmrCtr_Handler FuncPtr, void *CallBackRef)  
void XTmrCtr_InterruptHandler (void *InstancePtr)
```

Define Documentation

```
#define XTC_AUTO_RELOAD_OPTION
```

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define XTC_CAPTURE_MODE_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define XTC_DOWN_COUNT_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define XTC_ENABLE_ALL_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define XTC_EXT_COMPARE_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

#define XTC_INT_MODE_OPTION

Used to configure the timer counter device.

XTC_ENABLE_ALL_OPTION	Enables all timer counters at once.
XTC_DOWN_COUNT_OPTION	Configures the timer counter to count down from start value, the default is to count up.
XTC_CAPTURE_MODE_OPTION	Configures the timer to capture the timer counter value when the external capture line is asserted. The default mode is compare mode.
XTC_INT_MODE_OPTION	Enables the timer counter interrupt output.
XTC_AUTO_RELOAD_OPTION	In compare mode, configures the timer counter to reload from the compare value. The default mode causes the timer counter to hold when the compare value is hit. In capture mode, configures the timer counter to not hold the previous capture value if a new event occurs. The default mode cause the timer counter to hold the capture value until recognized.
XTC_EXT_COMPARE_OPTION	Enables the external compare output signal.

```
typedef void(* XTmrCtr_Handler)(void *CallbackRef, Xuint8 TmrCtrNumber)
```

Signature for the callback function.

Parameters:

CallbackRef is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. Its type is unimportant to the driver, so it is a void pointer.

TmrCtrNumber is the number of the timer/counter within the device. The device typically contains at least two timer/counters. The timer number is a zero based number with a range of 0 to (XTC_DEVICE_TIMER_COUNT - 1).

Function Documentation

```
void XTmrCtr_ClearStats( XTmrCtr * InstancePtr)
```

Clear the **XTmrCtrStats** structure for this driver.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

Returns:

None.

Note:

None.

```
Xuint32 XTmrCtr_GetCaptureValue( XTmrCtr * InstancePtr,  
                                Xuint8   TmrCtrNumber  
                                )
```

Returns the timer counter value that was captured the last time the external capture input was asserted.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The current capture value for the indicated timer counter.

Note:

None.

```
Xuint32 XTmrCtr_GetOptions( XTmrCtr * InstancePtr,  
                           Xuint8   TmrCtrNumber  
                           )
```

Get the options for the specified timer counter.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The currently set options. An option which is set to a '1' is enabled and set to a '0' is disabled. The options are bit masks such that multiple options may be set or cleared. The options are described in [xtmrctr.h](#).

Note:

None.

```
void XTmrCtr_GetStats( XTmrCtr * InstancePtr,  
                      XTmrCtrStats * StatsPtr  
                      )
```

Get a copy of the **XTmrCtrStats** structure, which contains the current statistics for this driver.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

StatsPtr is a pointer to a **XTmrCtrStats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

```
Xuint32 XTmrCtr_GetValue( XTmrCtr * InstancePtr,  
                          Xuint8   TmrCtrNumber  
                          )
```

Get the current value of the specified timer counter. The timer counter may be either incrementing or decrementing based upon the current mode of operation.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The current value for the timer counter.

Note:

None.

```
XStatus XTmrCtr_Initialize( XTmrCtr * InstancePtr,  
                          Xuint16 DeviceId  
                          )
```

Initializes a specific timer/counter instance/driver. Initialize fields of the **XTmrCtr** structure, then reset the timer/counter

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XTmrCtr** component. Passing in a device id associates the generic **XTmrCtr** component to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if the device doesn't exist

Note:

None.

```
void XTmrCtr_InterruptHandler( void * InstancePtr)
```

Interrupt Service Routine (ISR) for the driver. This function only performs processing for the device and does not save and restore the interrupt context.

Parameters:

InstancePtr contains a pointer to the timer/counter instance for the interrupt.

Returns:

None.

Note:

None.

```
Xboolean XTmrCtr_IsExpired( XTmrCtr * InstancePtr,  
                             Xuint8      TmrCtrNumber  
                             )
```

Checks if the specified timer counter of the device has expired. In capture mode, expired is defined as a capture occurred. In compare mode, expired is defined as the timer counter rolled over/under for up/down counting.

When interrupts are enabled, the expiration causes an interrupt. This function is typically used to poll a timer counter to determine when it has expired.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

XTRUE if the timer has expired, and XFALSE otherwise.

Note:

None.

```
void XTmrCtr_Reset( XTmrCtr * InstancePtr,  
                    Xuint8      TmrCtrNumber  
                    )
```

Resets the specified timer counter of the device. A reset causes the timer counter to set its value to the reset value.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
XStatus XTmrCtr_SelfTest( XTmrCtr * InstancePtr,  
                          Xuint8      TmrCtrNumber  
                          )
```

Runs a self-test on the driver/device. This test verifies that the specified timer counter of the device can be enabled and increments.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

XST_SUCCESS if self-test was successful, or XST_FAILURE if the timer is not incrementing.

Note:

This is a destructive test using the provided timer. The current settings of the timer are returned to the initialized values and all settings at the time this function is called are overwritten.

```
void XTmrCtr_SetHandler( XTmrCtr *      InstancePtr,  
                         XTmrCtr_Handler FuncPtr,  
                         void *          CallbackRef  
                         )
```

Sets the timer callback function, which the driver calls when the specified timer times out.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context so the function that is called should either be short or pass the more extensive processing off to another task to allow the interrupt to return and normal processing to continue.

```
void XTmrCtr_SetOptions( XTmrCtr * InstancePtr,  
                        Xuint8    TmrCtrNumber,  
                        Xuint32   Options  
                        )
```

Enables the specified options for the specified timer counter. This function sets the options without regard to the current options of the driver. To prevent a loss of the current options, the user should call **XTmrCtr_GetOptions()** prior to this function and modify the retrieved options to pass into this function to prevent loss of the current options.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Options contains the desired options to be set or cleared. Setting the option to '1' enables the option, clearing the to '0' disables the option. The options are bit masks such that multiple options may be set or cleared. The options are described in **xtmrctr.h**.

Returns:

None.

Note:

None.

```
void XTmrCtr_SetResetValue( XTmrCtr * InstancePtr,  
                            Xuint8    TmrCtrNumber,  
                            Xuint32   ResetValue  
                            )
```

Set the reset value for the specified timer counter. This is the value that is loaded into the timer counter when it is reset. This value is also loaded when the timer counter is started.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

ResetValue contains the value to be used to reset the timer counter.

Returns:

None.

Note:

None.

```
void XTmrCtr_Start( XTmrCtr * InstancePtr,  
                   Xuint8    TmrCtrNumber  
                   )
```

Starts the specified timer counter of the device such that it starts running. The timer counter is reset before it is started and the reset value is loaded into the timer counter.

If interrupt mode is specified in the options, it is necessary for the caller to connect the interrupt handler of the timer/counter to the interrupt source, typically an interrupt controller, and enable the interrupt within the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
void XTmrCtr_Stop( XTmrCtr * InstancePtr,  
                  Xuint8   TmrCtrNumber  
                  )
```

Stops the timer counter by disabling it.

It is the callers' responsibility to disconnect the interrupt handler of the timer_counter from the interrupt source, typically an interrupt controller, and disable the interrupt within the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

tmrctr/v1_00_b/src/xtmrctr.c File Reference

Detailed Description

Contains required functions for the [XTmrCtr](#) driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xparameters.h"  
#include "xtmrctr.h"  
#include "xtmrctr_i.h"
```

Functions

```
XStatus XTmrCtr_Initialize (XTmrCtr *InstancePtr, Xuint16 DeviceId)  
void XTmrCtr_Start (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
void XTmrCtr_Stop (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)  
Xuint32 XTmrCtr_GetValue (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
```



```
void XTmrCtr_SetResetValue (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber, Xuint32  
ResetValue)
```

```
Xuint32 XTmrCtr_GetCaptureValue (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
```

```
void XTmrCtr_Reset (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
```

```
Xboolean XTmrCtr_IsExpired (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
```

Function Documentation

```
Xuint32 XTmrCtr_GetCaptureValue( XTmrCtr * InstancePtr,  
Xuint8 TmrCtrNumber  
)
```

Returns the timer counter value that was captured the last time the external capture input was asserted.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The current capture value for the indicated timer counter.

Note:

None.

```
Xuint32 XTmrCtr_GetValue( XTmrCtr * InstancePtr,  
Xuint8 TmrCtrNumber  
)
```

Get the current value of the specified timer counter. The timer counter may be either incrementing or decrementing based upon the current mode of operation.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The current value for the timer counter.

Note:

None.

```
XStatus XTmrCtr_Initialize( XTmrCtr * InstancePtr,  
                           Xuint16 DeviceId  
                           )
```

Initializes a specific timer/counter instance/driver. Initialize fields of the **XTmrCtr** structure, then reset the timer/counter

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XTmrCtr** component. Passing in a device id associates the generic **XTmrCtr** component to a specific device, as chosen by the caller or application developer.

Returns:

- o XST_SUCCESS if initialization was successful
- o XST_DEVICE_IS_STARTED if the device has already been started
- o XST_DEVICE_NOT_FOUND if the device doesn't exist

Note:

None.

```
Xboolean XTmrCtr_IsExpired( XTmrCtr * InstancePtr,  
                             Xuint8 TmrCtrNumber  
                             )
```

Checks if the specified timer counter of the device has expired. In capture mode, expired is defined as a capture occurred. In compare mode, expired is defined as the timer counter rolled over/under for up/down counting.

When interrupts are enabled, the expiration causes an interrupt. This function is typically used to poll a timer counter to determine when it has expired.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

XTRUE if the timer has expired, and XFALSE otherwise.

Note:

None.

```
void XTmrCtr_Reset( XTmrCtr * InstancePtr,  
                   Xuint8 TmrCtrNumber  
                   )
```

Resets the specified timer counter of the device. A reset causes the timer counter to set it's value to the reset value.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
void XTmrCtr_SetResetValue( XTmrCtr * InstancePtr,  
                             Xuint8   TmrCtrNumber,  
                             Xuint32  ResetValue  
                             )
```

Set the reset value for the specified timer counter. This is the value that is loaded into the timer counter when it is reset. This value is also loaded when the timer counter is started.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

ResetValue contains the value to be used to reset the timer counter.

Returns:

None.

Note:

None.

```
void XTmrCtr_Start( XTmrCtr * InstancePtr,  
                   Xuint8   TmrCtrNumber  
                   )
```

Starts the specified timer counter of the device such that it starts running. The timer counter is reset before it is started and the reset value is loaded into the timer counter.

If interrupt mode is specified in the options, it is necessary for the caller to connect the interrupt handler of the timer/counter to the interrupt source, typically an interrupt controller, and enable the interrupt within the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
void XTmrCtr_Stop( XTmrCtr * InstancePtr,  
                  Xuint8   TmrCtrNumber  
                  )
```

Stops the timer counter by disabling it.

It is the callers' responsibility to disconnect the interrupt handler of the timer_counter from the interrupt source, typically an interrupt controller, and disable the interrupt within the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

XTmrCtr Struct Reference

```
#include <xtmrctr.h>
```

Detailed Description

The XTmrCtr driver instance data. The user is required to allocate a variable of this type for every timer/counter device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- `tmrctr/v1_00_b/src/xtmrctr.h`

tmrctr/v1_00_b/src/xtmrctr.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xtmrctr.h,v 1.10 2002/05/03 17:13:57 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file tmrctr/v1_00_b/src/xtmrctr.h
00026 *
00027 * The Xilinx timer/counter component. This component supports the Xilinx
00028 * timer/counter. More detailed description of the driver operation can
00029 * be found in the xtmrctr.c file.
00030 *
00031 * The Xilinx timer/counter supports the following features:
00032 * - Polled mode.
00033 * - Interrupt driven mode
00034 * - enabling and disabling specific timers
00035 * - PWM operation
00036 *
00037 * The driver does not currently support the PWM operation of the device.
00038 *
00039 * The timer counter operates in 2 primary modes, compare and capture. In
00040 * either mode, the timer counter may count up or down, with up being the
00041 * default.
00042 *
```

00043 * Compare mode is typically used for creating a single time period or multiple
00044 * repeating time periods in the auto reload mode, such as a periodic interrupt.
00045 * When started, the timer counter loads an initial value, referred to as the
00046 * compare value, into the timer counter and starts counting down or up. The
00047 * timer counter expires when it rolls over/under depending upon the mode of
00048 * counting. An external compare output signal may be configured such that a
00049 * pulse is generated with this signal when it hits the compare value.
00050 *
00051 * Capture mode is typically used for measuring the time period between
00052 * external events. This mode uses an external capture input signal to cause
00053 * the value of the timer counter to be captured. When started, the timer
00054 * counter loads an initial value, referred to as the compare value,
00055 *
00056 * The timer can be configured to either cause an interrupt when the count
00057 * reaches the compare value in compare mode or latch the current count
00058 * value in the capture register when an external input is asserted
00059 * in capture mode. The external capture input can be enabled/disabled using the
00060 * XTmrCtr_SetOptions function. While in compare mode, it is also possible to
00061 * drive an external output when the compare value is reached in the count
00062 * register The external compare output can be enabled/disabled using the
00063 * XTmrCtr_SetOptions function.
00064 *
00065 * **Interrupts**
00066 *
00067 * It is the responsibility of the application to connect the interrupt
00068 * handler of the timer/counter to the interrupt source. The interrupt
00069 * handler function, XTmrCtr_InterruptHandler, is visible such that the user
00070 * can connect it to the interrupt source. Note that this interrupt handler
00071 * does not provide interrupt context save and restore processing, the user
00072 * must perform this processing.
00073 *
00074 * The driver services interrupts and passes timeouts to the upper layer
00075 * software through callback functions. The upper layer software must register
00076 * its callback functions during initialization. The driver requires callback
00077 * functions for timers.
00078 *
00079 * @note
00080 * The default settings for the timers are:
00081 * - Interrupt generation disabled
00082 * - Count up mode
00083 * - Compare mode
00084 * - Hold counter (will not reload the timer)
00085 * - External compare output disabled
00086 * - External capture input disabled
00087 * - Pulse width modulation disabled
00088 * - Timer disabled, waits for Start function to be called
00089 *

00090 * A timer counter device may contain multiple timer counters. The symbol
00091 * XTC_DEVICE_TIMER_COUNT defines the number of timer counters in the device.
00092 * The device currently contains 2 timer counters.
00093 *

00094 * This driver is intended to be RTOS and processor independent. It works with
00095 * physical addresses only. Any needs for dynamic memory management, threads


```

00096 * or thread mutual exclusion, virtual memory, or cache control must be
00097 * satisfied by the layer above this driver.
00098 *
00099 * <pre>
00100 * MODIFICATION HISTORY:
00101 *
00102 * Ver    Who    Date      Changes
00103 * -----
00104 * 1.00a ecm   08/16/01 First release
00105 * 1.00b jhl   02/21/02 Repartitioned the driver for smaller files
00106 * </pre>
00107 *
00108 *****/
00109
00110 #ifndef XTMRCTR_H /* prevent circular inclusions */
00111 #define XTMRCTR_H /* by using protection macros */
00112
00113 /***** Include Files *****/
00114
00115 #include "xbasic_types.h"
00116 #include "xstatus.h"
00117 #include "xtmrctr_l.h"
00118
00119 /***** Constant Definitions *****/
00120
00121 /**
00122  * @name Configuration options
00123  * These options are used in XTmrCtr_SetOptions() and XTmrCtr_GetOptions()
00124  * @{
00125  */
00126 /**
00127  * Used to configure the timer counter device.
00128  * <pre>
00129  * XTC_ENABLE_ALL_OPTION      Enables all timer counters at once.
00130  * XTC_DOWN_COUNT_OPTION      Configures the timer counter to count down from
00131  *                             start value, the default is to count up.
00132  * XTC_CAPTURE_MODE_OPTION    Configures the timer to capture the timer counter
00133  *                             value when the external capture line is asserted.
00134  *                             The default mode is compare mode.
00135  * XTC_INT_MODE_OPTION        Enables the timer counter interrupt output.
00136  * XTC_AUTO_RELOAD_OPTION      In compare mode, configures the timer counter to
00137  *                             reload from the compare value. The default mode
00138  *                             causes the timer counter to hold when the compare
00139  *                             value is hit.
00140  *                             In capture mode, configures the timer counter to
00141  *                             not hold the previous capture value if a new event
00142  *                             occurs. The default mode cause the timer counter
00143  *                             to hold the capture value until recognized.
00144  * XTC_EXT_COMPARE_OPTION      Enables the external compare output signal.
00145  * </pre>
00146  */
00147 #define XTC_ENABLE_ALL_OPTION      0x00000040UL

```

```

00148 #define XTC_DOWN_COUNT_OPTION      0x00000020UL
00149 #define XTC_CAPTURE_MODE_OPTION    0x00000010UL
00150 #define XTC_INT_MODE_OPTION         0x00000008UL
00151 #define XTC_AUTO_RELOAD_OPTION      0x00000004UL
00152 #define XTC_EXT_COMPARE_OPTION      0x00000002UL
00153 /*@}*/
00154
00155 /***** Type Definitions *****/
00156
00157 /**
00158  * This typedef contains configuration information for the device.
00159  */
00160 typedef struct
00161 {
00162     Xuint16 DeviceId;           /**< Unique ID  of device */
00163     Xuint32 BaseAddress;       /**< Register base address */
00164 } XTmrCtr_Config;
00165
00166 /**
00167  * Signature for the callback function.
00168  *
00169  * @param CallbackRef is a callback reference passed in by the upper layer
00170  *         when setting the callback functions, and passed back to the upper
00171  *         layer when the callback is invoked. Its type is unimportant to the
00172  *         driver, so it is a void pointer.
00173  * @param TmrCtrNumber is the number of the timer/counter within the device.
00174  *         The device typically contains at least two timer/counters. The
00175  *         timer number is a zero based number with a range of 0 to
00176  *         (XTC_DEVICE_TIMER_COUNT - 1).
00177  */
00178 typedef void (*XTmrCtr_Handler)(void *CallbackRef, Xuint8 TmrCtrNumber);
00179
00180 /**
00181  * Timer/Counter statistics
00182  */
00183 typedef struct
00184 {
00185     Xuint32 Interrupts;         /**< The number of interrupts that have occurred
00186  */
00187 } XTmrCtrStats;
00188
00189 /**
00190  * The XTmrCtr driver instance data. The user is required to allocate a
00191  * variable of this type for every timer/counter device in the system. A
00192  * pointer to a variable of this type is then passed to the driver API
00193  * functions.
00194  */
00195 typedef struct
00196 {
00197     XTmrCtrStats Stats;         /* Component Statistics */
00198     Xuint32 BaseAddress;        /* Base address of registers */

```

```

00198     Xuint32 IsReady;                /* Device is initialized and ready */
00199
00200     XTmrCtr_Handler Handler;        /* Callback function */
00201     void *CallBackRef;              /* Callback reference for handler */
00202 } XTmrCtr;
00203
00204
00205 /***** Macros (Inline Functions) Definitions *****/
00206
00207
00208 /***** Function Prototypes *****/
00209
00210 /*
00211  * Required functions, in file xtmrctr.c
00212  */
00213 XStatus XTmrCtr_Initialize(XTmrCtr *InstancePtr, Xuint16 DeviceId);
00214 void XTmrCtr_Start(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber);
00215 void XTmrCtr_Stop(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber);
00216 Xuint32 XTmrCtr_GetValue(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber);
00217 void XTmrCtr_SetResetValue(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber,
00218                             Xuint32 ResetValue);
00219 Xuint32 XTmrCtr_GetCaptureValue(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber);
00220 Xboolean XTmrCtr_IsExpired(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber);
00221 void XTmrCtr_Reset(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber);
00222
00223 XTmrCtr_Config *XTmrCtr_LookupConfig(Xuint16 DeviceId);
00224
00225 /*
00226  * Functions for options, in file xtmrctr_options.c
00227  */
00228 void XTmrCtr_SetOptions(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber,
00229                         Xuint32 Options);
00230 Xuint32 XTmrCtr_GetOptions(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber);
00231
00232 /*
00233  * Functions for statistics, in file xtmrctr_stats.c
00234  */
00235 void XTmrCtr_GetStats(XTmrCtr *InstancePtr, XTmrCtrStats *StatsPtr);
00236 void XTmrCtr_ClearStats(XTmrCtr *InstancePtr);
00237
00238 /*
00239  * Functions for self-test, in file xtmrctr_selftest.c
00240  */
00241 XStatus XTmrCtr_SelfTest(XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber);
00242
00243 /*
00244  * Functions for interrupts, in file xtmrctr_intr.c
00245  */
00246 void XTmrCtr_SetHandler(XTmrCtr *InstancePtr, XTmrCtr_Handler FuncPtr,
00247                         void *CallBackRef);

```

```
00248 void XTmrCtr_InterruptHandler(void *InstancePtr);
00249
00250 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

tmrctr/v1_00_b/src/xtmrctr_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xtmrctr.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XTC_DEVICE_TIMER_COUNT  
#define XTimerCtr_mReadReg(BaseAddress, TmrCtrNumber, RegOffset)  
#define XTmrCtr_mWriteReg(BaseAddress, TmrCtrNumber, RegOffset, ValueToWrite)  
#define XTmrCtr_mSetControlStatusReg(BaseAddress, TmrCtrNumber, RegisterValue)  
#define XTmrCtr_mGetControlStatusReg(BaseAddress, TmrCtrNumber)
```

```
#define XTmrCtr_mGetTimerCounterReg(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mSetLoadReg(BaseAddress, TmrCtrNumber, RegisterValue)
#define XTmrCtr_mGetLoadReg(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mEnable(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mDisable(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mEnableIntr(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mDisableIntr(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mLoadTimerCounterReg(BaseAddress, TmrCtrNumber)
#define XTmrCtr_mHasEventOccurred(BaseAddress, TmrCtrNumber)
```

Define Documentation

```
#define XTC_DEVICE_TIMER_COUNT
```

Defines the number of timer counters within a single hardware device. This number is not currently parameterized in the hardware but may be in the future.

```
#define XTimerCtr_mReadReg( BaseAddress,
                           TmrCtrNumber,
                           RegOffset      )
```

Read one of the timer counter registers.

Parameters:

BaseAddress contains the base address of the timer counter device.

TmrCtrNumber contains the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

RegOffset contains the offset from the 1st register of the timer counter to select the specific register of the timer counter.

Returns:

The value read from the register, a 32 bit value.

Note:

None.

```
#define XTmrCtr_mDisable( BaseAddress,
                          TmrCtrNumber )
```

Disable a timer counter such that it stops running.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
#define XTmrCtr_mDisableIntr( BaseAddress,  
                             TmrCtrNumber )
```

Disable the interrupt for a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
#define XTmrCtr_mEnable( BaseAddress,  
                        TmrCtrNumber )
```

Enable a timer counter such that it starts running.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
#define XTmrCtr_mEnableIntr( BaseAddress,  
                             TmrCtrNumber )
```

Enable the interrupt for a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
#define XTmrCtr_mGetControlStatusReg( BaseAddress,  
                                       TmrCtrNumber )
```


Get the Control Status Register of a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The value read from the register, a 32 bit value.

Note:

None.

```
#define XTmrCtr_mGetLoadReg( BaseAddress,  
                             TmrCtrNumber )
```

Get the Load Register of a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The value read from the register, a 32 bit value.

Note:

None.

```
#define XTmrCtr_mGetTimerCounterReg( BaseAddress,  
                                     TmrCtrNumber )
```

Get the Timer Counter Register of a timer counter.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The value read from the register, a 32 bit value.

Note:

None.

```
#define XTmrCtr_mHasEventOccurred( BaseAddress,  
                                   TmrCtrNumber )
```

Determine if a timer counter event has occurred. Events are defined to be when a capture has occurred or the counter has roller over.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Note:

None.

```
#define XTmrCtr_mLoadTimerCounterReg( BaseAddress,  
                                       TmrCtrNumber )
```

Cause the timer counter to load it's Timer Counter Register with the value in the Load Register.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

None.

Note:

None.

```
#define XTmrCtr_mSetControlStatusReg( BaseAddress,  
                                     TmrCtrNumber,  
                                     RegisterValue  )
```

Set the Control Status Register of a timer counter to the specified value.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

RegisterValue is the 32 bit value to be written to the register.

Returns:

None.

Note:

None.

```
#define XTmrCtr_mSetLoadReg( BaseAddress,  
                             TmrCtrNumber,  
                             RegisterValue  )
```

Set the Load Register of a timer counter to the specified value.

Parameters:

BaseAddress is the base address of the device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

RegisterValue is the 32 bit value to be written to the register.

Returns:

None.

Note:

None.

```
#define XTmrCtr_mWriteReg( BaseAddress,  
                          TmrCtrNumber,  
                          RegOffset,  
                          ValueToWrite )
```

Write a specified value to a register of a timer counter.

Parameters:

BaseAddress is the base address of the timer counter device.

TmrCtrNumber is the specific timer counter within the device, a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).

RegOffset contain the offset from the 1st register of the timer counter to select the specific register of the timer counter.

ValueToWrite is the 32 bit value to be written to the register.

Returns:

None

tmrctr/v1_00_b/src/xtmrctr_1.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xtmrctr_1.h,v 1.2 2002/05/03 17:13:57 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file tmrctr/v1_00_b/src/xtmrctr_1.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  The user should refer to the
00029 * hardware device specification for more details of the device operation.
00030 * High-level driver functions are defined in xtmrctr.h.
00031 *
00032 * <pre>
00033 * MODIFICATION HISTORY:
00034 *
00035 * Ver   Who   Date     Changes
00036 * ----  ---  -
00037 * 1.00b jhl   04/24/02 First release
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XTMRCTR_L_H /* prevent circular inclusions */
```

```

00043 #define XTMRCTR_L_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048 #include "xio.h"
00049
00050 /***** Constant Definitions *****/
00051
00052 /**
00053  * Defines the number of timer counters within a single hardware device. This
00054  * number is not currently parameterized in the hardware but may be in the
00055  * future.
00056  */
00057 #define XTC_DEVICE_TIMER_COUNT      2
00058
00059 /* Each timer counter consumes 16 bytes of address space */
00060
00061 #define XTC_TIMER_COUNTER_OFFSET 16
00062
00063 /* Register offsets within a timer counter, there are multiple timer counters
00064  * within a single device
00065  */
00066
00067 #define XTC_TCSR_OFFSET      0      /* control/status register */
00068 #define XTC_TLR_OFFSET      4      /* load register */
00069 #define XTC_TCR_OFFSET      8      /* timer counter register */
00070
00071 /* Control Status Register bit masks */
00072
00073 #define XTC_CSR_ENABLE_ALL_MASK      0x00000400      /* ENALL bit */
00074 #define XTC_CSR_ENABLE_PWM_MASK     0x00000200      /* PWMA bit */
00075 #define XTC_CSR_INT_OCCURED_MASK    0x00000100      /* TINT bit */
00076 #define XTC_CSR_ENABLE_TMR_MASK     0x00000080      /* ENT bit */
00077 #define XTC_CSR_ENABLE_INT_MASK     0x00000040      /* ENIT bit */
00078 #define XTC_CSR_LOAD_MASK           0x00000020      /* LOAD bit */
00079 #define XTC_CSR_AUTO_RELOAD_MASK    0x00000010      /* ARHT bit */
00080 #define XTC_CSR_EXT_CAPTURE_MASK    0x00000008      /* CAPT bit */
00081 #define XTC_CSR_EXT_GENERATE_MASK   0x00000004      /* GENT bit */
00082 #define XTC_CSR_DOWN_COUNT_MASK     0x00000002      /* UDT bit */
00083 #define XTC_CSR_CAPTURE_MODE_MASK   0x00000001      /* MDT bit */
00084
00085 /***** Type Definitions *****/
00086
00087 extern Xuint8 XTmrCtr_Offsets[];
00088
00089 /***** Macros (Inline Functions) Definitions *****/
00090
00091
00092 /*****
00093  *
00094  * Low-level driver macros. The list below provides signatures to help the

```

```

00095 * user use the macros.
00096 *
00097 * Xuint32 XTmrCtr_mReadReg(Xuint32 BaseAddress, Xuint8 TimerNumber,
00098 *                          unsigned RegOffset)
00099 * void XTmrCtr_mWriteReg(Xuint32 BaseAddress, Xuint8 TimerNumber,
00100 *                          unsigned RegOffset, Xuint32 ValueToWrite)
00101 *
00102 * void XTmrCtr_mSetControlStatusReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber,
00103 *                                  Xuint32 RegisterValue)
00104 * Xuint32 XTmrCtr_mGetControlStatusReg(Xuint32 BaseAddress, Xuint8
TmrCtrNumber)
00105 *
00106 * Xuint32 XTmrCtr_mGetTimerCounterReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)
00107 *
00108 * void XTmrCtr_mSetLoadReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber,
00109 *                           Xuint32 RegisterValue)
00110 * Xuint32 XTmrCtr_mGetLoadReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)
00111 *
00112 * void XTmrCtr_mEnable(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)
00113 * void XTmrCtr_mDisable(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)
00114 *
00115 * void XTmrCtr_mEnableIntr(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)
00116 * void XTmrCtr_mDisableIntr(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)
00117 *
00118 * void XTmrCtr_mLoadTimerCounterReg(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)
00119 * Xboolean XTmrCtr_mHasEventOccurred(Xuint32 BaseAddress, Xuint8 TmrCtrNumber)
00120 *
00121 *****/
00122
00123 /***/
00124 /**
00125 * Read one of the timer counter registers.
00126 *
00127 * @param BaseAddress contains the base address of the timer counter device.
00128 * @param TmrCtrNumber contains the specific timer counter within the device,
00129 * a zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00130 * @param RegOffset contains the offset from the 1st register of the timer
00131 * counter to select the specific register of the timer counter.
00132 *
00133 * @return The value read from the register, a 32 bit value.
00134 *
00135 * @note None.
00136 *
00137 *****/
00138 #define XTmrCtr_mReadReg(BaseAddress, TmrCtrNumber, RegOffset) \
00139     XIo_In32((BaseAddress) + XTmrCtr_Offsets[(TmrCtrNumber)] + (RegOffset))
00140
00141 *****/
00142 /**
00143 * Write a specified value to a register of a timer counter.
00144 *
00145 * @param BaseAddress is the base address of the timer counter device.

```

```

00146 * @param TmrCtrNumber is the specific timer counter within the device, a
00147 * zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00148 * @param RegOffset contain the offset from the 1st register of the timer
00149 * counter to select the specific register of the timer counter.
00150 * @param ValueToWrite is the 32 bit value to be written to the register.
00151 *
00152 * @return None
00153 *
00154 *****/
00155 #define XTmrCtr_mWriteReg(BaseAddress, TmrCtrNumber, RegOffset, ValueToWrite) \
00156     XIo_Out32(((BaseAddress) + XTmrCtr_Offsets[(TmrCtrNumber)] + \
00157         (RegOffset)), (ValueToWrite))
00158
00159 *****/
00160 /**
00161 *
00162 * Set the Control Status Register of a timer counter to the specified value.
00163 *
00164 * @param BaseAddress is the base address of the device.
00165 * @param TmrCtrNumber is the specific timer counter within the device, a
00166 * zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00167 * @param RegisterValue is the 32 bit value to be written to the register.
00168 *
00169 * @return None.
00170 *
00171 * @note None.
00172 *
00173 *****/
00174 #define XTmrCtr_mSetControlStatusReg(BaseAddress, TmrCtrNumber, RegisterValue) \
00175     XTmrCtr_mWriteReg((BaseAddress), (TmrCtrNumber), XTC_TCSR_OFFSET, \
00176         (RegisterValue))
00177
00178 *****/
00179 /**
00180 *
00181 * Get the Control Status Register of a timer counter.
00182 *
00183 * @param BaseAddress is the base address of the device.
00184 * @param TmrCtrNumber is the specific timer counter within the device, a
00185 * zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00186 *
00187 * @return The value read from the register, a 32 bit value.
00188 *
00189 * @note None.
00190 *
00191 *****/
00192 #define XTmrCtr_mGetControlStatusReg(BaseAddress, TmrCtrNumber) \
00193     XTimerCtr_mReadReg((BaseAddress), (TmrCtrNumber), XTC_TCSR_OFFSET)
00194
00195 *****/
00196 /**

```



```

00197 *
00198 * Get the Timer Counter Register of a timer counter.
00199 *
00200 * @param BaseAddress is the base address of the device.
00201 * @param TmrCtrNumber is the specific timer counter within the device, a
00202 * zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00203 *
00204 * @return The value read from the register, a 32 bit value.
00205 *
00206 * @note None.
00207 *
00208 *****/
00209 #define XTmrCtr_mGetTimerCounterReg(BaseAddress, TmrCtrNumber) \
00210     XTimerCtr_mReadReg((BaseAddress), (TmrCtrNumber), XTC_TCR_OFFSET)
00211
00212 *****/
00213 /**
00214 *
00215 * Set the Load Register of a timer counter to the specified value.
00216 *
00217 * @param BaseAddress is the base address of the device.
00218 * @param TmrCtrNumber is the specific timer counter within the device, a
00219 * zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00220 * @param RegisterValue is the 32 bit value to be written to the register.
00221 *
00222 * @return None.
00223 *
00224 * @note None.
00225 *
00226 *****/
00227 #define XTmrCtr_mSetLoadReg(BaseAddress, TmrCtrNumber, RegisterValue) \
00228     XTmrCtr_mWriteReg((BaseAddress), (TmrCtrNumber), XTC_TLR_OFFSET, \
00229         (RegisterValue))
00230
00231 *****/
00232 /**
00233 *
00234 * Get the Load Register of a timer counter.
00235 *
00236 * @param BaseAddress is the base address of the device.
00237 * @param TmrCtrNumber is the specific timer counter within the device, a
00238 * zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00239 *
00240 * @return The value read from the register, a 32 bit value.
00241 *
00242 * @note None.
00243 *
00244 *****/
00245 #define XTmrCtr_mGetLoadReg(BaseAddress, TmrCtrNumber) \
00246     XTimerCtr_mReadReg((BaseAddress), (TmrCtrNumber), XTC_TLR_OFFSET)
00247
00248 *****/

```

```

00249 /**
00250 *
00251 * Enable a timer counter such that it starts running.
00252 *
00253 * @param BaseAddress is the base address of the device.
00254 * @param TmrCtrNumber is the specific timer counter within the device, a
00255 * zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00256 *
00257 * @return None.
00258 *
00259 * @note None.
00260 *
00261 *****/
00262 #define XTmrCtr_mEnable(BaseAddress, TmrCtrNumber) \
00263     XTmrCtr_mWriteReg((BaseAddress), (TmrCtrNumber), XTC_TCSR_OFFSET, \
00264     (XTimerCtr_mReadReg((BaseAddress), (TmrCtrNumber), \
00265     XTC_TCSR_OFFSET) | XTC_CSR_ENABLE_TMR_MASK))
00266
00267 *****/
00268 /**
00269 *
00270 * Disable a timer counter such that it stops running.
00271 *
00272 * @param BaseAddress is the base address of the device.
00273 * @param TmrCtrNumber is the specific timer counter within the device, a
00274 * zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00275 *
00276 * @return None.
00277 *
00278 * @note None.
00279 *
00280 *****/
00281 #define XTmrCtr_mDisable(BaseAddress, TmrCtrNumber) \
00282     XTmrCtr_mWriteReg((BaseAddress), (TmrCtrNumber), XTC_TCSR_OFFSET, \
00283     (XTimerCtr_mReadReg((BaseAddress), (TmrCtrNumber), \
00284     XTC_TCSR_OFFSET) & ~XTC_CSR_ENABLE_TMR_MASK))
00285
00286 *****/
00287 /**
00288 *
00289 * Enable the interrupt for a timer counter.
00290 *
00291 * @param BaseAddress is the base address of the device.
00292 * @param TmrCtrNumber is the specific timer counter within the device, a
00293 * zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00294 *
00295 * @return None.
00296 *
00297 * @note None.
00298 *
00299 *****/
00300 #define XTmrCtr_mEnableIntr(BaseAddress, TmrCtrNumber) \

```

```

00301     XTmrCtr_mWriteReg((BaseAddress), (TmrCtrNumber), XTC_TCSR_OFFSET, \
00302                     (XTimerCtr_mReadReg((BaseAddress), (TmrCtrNumber), \
00303                     XTC_TCSR_OFFSET) | XTC_CSR_ENABLE_INT_MASK))
00304
00305 /*****
00306 /**
00307 *
00308 * Disable the interrupt for a timer counter.
00309 *
00310 * @param   BaseAddress is the base address of the device.
00311 * @param   TmrCtrNumber is the specific timer counter within the device, a
00312 *           zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00313 *
00314 * @return  None.
00315 *
00316 * @note   None.
00317 *
00318 *****/
00319 #define XTmrCtr_mDisableIntr(BaseAddress, TmrCtrNumber) \
00320     XTmrCtr_mWriteReg((BaseAddress), (TmrCtrNumber), XTC_TCSR_OFFSET, \
00321                     (XTimerCtr_mReadReg((BaseAddress), (TmrCtrNumber), \
00322                     XTC_TCSR_OFFSET) & ~XTC_CSR_ENABLE_INT_MASK))
00323
00324
00325 /*****
00326 /**
00327 *
00328 * Cause the timer counter to load it's Timer Counter Register with the value
00329 * in the Load Register.
00330 *
00331 * @param   BaseAddress is the base address of the device.
00332 * @param   TmrCtrNumber is the specific timer counter within the device, a
00333 *           zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00334 *
00335 * @return  None.
00336 *
00337 * @note   None.
00338 *
00339 *****/
00340 #define XTmrCtr_mLoadTimerCounterReg(BaseAddress, TmrCtrNumber) \
00341     XTmrCtr_mWriteReg((BaseAddress), (TmrCtrNumber), XTC_TCSR_OFFSET, \
00342                     (XTimerCtr_mReadReg((BaseAddress), (TmrCtrNumber), \
00343                     XTC_TCSR_OFFSET) | XTC_CSR_LOAD_MASK))
00344
00345 /*****
00346 /**
00347 *
00348 * Determine if a timer counter event has occurred. Events are defined to be
00349 * when a capture has occurred or the counter has roller over.
00350 *
00351 * @param   BaseAddress is the base address of the device.
00352 * @param   TmrCtrNumber is the specific timer counter within the device, a

```

```
00353 *          zero based number, 0 - (XTC_DEVICE_TIMER_COUNT - 1).
00354 *
00355 * @note      None.
00356 *
00357 *****/
00358 #define XTmrCtr_mHasEventOccurred(BaseAddress, TmrCtrNumber)          \
00359     (XTimerCtr_mReadReg((BaseAddress), (TmrCtrNumber),                \
00360                          XTC_TCSR_OFFSET) & XTC_CSR_INT_OCCURED_MASK)
00361
00362 /***** Function Prototypes *****/
00363
00364 /***** Variable Definitions *****/
00365
00366 #endif          /* end of protection macro */
00367
```

Generated on 29 May 2003 for Xilinx Device Drivers

XTmrCtr_Config Struct Reference

```
#include <xtmrctr.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

Field Documentation

[Xuint32 XTmrCtr_Config::BaseAddress](#)

Register base address

[Xuint16 XTmrCtr_Config::DeviceId](#)

Unique ID of device

The documentation for this struct was generated from the following file:

- [tmrctr/v1_00_b/src/xtmrctr.h](#)
-

XTmrCtrStats Struct Reference

```
#include <xtmrctr.h>
```

Detailed Description

Timer/Counter statistics

Data Fields

[Xuint32 Interrupts](#)

Field Documentation

[Xuint32 XTmrCtrStats::Interrupts](#)

The number of interrupts that have occurred

The documentation for this struct was generated from the following file:

- `tmrctr/v1_00_b/src/xtmrctr.h`
-

tmrctr/v1_00_b/src/xtmrctr_options.c File Reference

Detailed Description

Contains configuration options functions for the **XTmrCtr** component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"  
#include "xtmrctr.h"  
#include "xtmrctr_i.h"
```

Data Structures

```
struct Mapping
```

Functions

```
void XTmrCtr_SetOptions (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber, Xuint32 Options)  
Xuint32 XTmrCtr_GetOptions (XTmrCtr *InstancePtr, Xuint8 TmrCtrNumber)
```

Function Documentation

```
Xuint32 XTmrCtr_GetOptions( XTmrCtr * InstancePtr,  
                           Xuint8   TmrCtrNumber  
                           )
```

Get the options for the specified timer counter.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

The currently set options. An option which is set to a '1' is enabled and set to a '0' is disabled. The options are bit masks such that multiple options may be set or cleared. The options are described in [xtmrctr.h](#).

Note:

None.

```
void XTmrCtr_SetOptions( XTmrCtr * InstancePtr,  
                        Xuint8   TmrCtrNumber,  
                        Xuint32  Options  
                        )
```

Enables the specified options for the specified timer counter. This function sets the options without regard to the current options of the driver. To prevent a loss of the current options, the user should call **XTmrCtr_GetOptions()** prior to this function and modify the retrieved options to pass into this function to prevent loss of the current options.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Options

contains the desired options to be set or cleared. Setting the option to '1' enables the option, clearing the to '0' disables the option. The options are bit masks such that multiple options may be set or cleared. The options are described in [xtmrctr.h](#).

Returns:

None.

Note:

None.

tmrctr/v1_00_b/src/xtmrctr_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xtmrctr_i.h,v 1.6 2002/05/02 20:38:16 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file tmrctr/v1_00_b/src/xtmrctr_i.h
00026 *
00027 * This file contains data which is shared between files internal to the
00028 * XTmrCtr component. It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver   Who   Date       Changes
00034 * ----  ---  -
00035 * 1.00b jhl   02/06/02 First release
00036 * </pre>
00037 *
00038 *****/
00039
00040 #ifndef XTMRCTR_I_H /* prevent circular inclusions */
00041 #define XTMRCTR_I_H /* by using protection macros */
00042
```

```
00043 /***** Include Files *****/
00044
00045 #include "xbasic_types.h"
00046
00047 /***** Constant Definitions *****/
00048
00049
00050 /***** Function Prototypes *****/
00051
00052
00053 /***** Variable Definitions *****/
00054
00055 extern XTmrCtr_Config XTmrCtr_ConfigTable[];
00056
00057 extern Xuint8 XTmrCtr_Offsets[];
00058
```

Generated on 29 May 2003 for Xilinx Device Drivers

tmrctr/v1_00_b/src/xtmrctr_i.h File Reference

Detailed Description

This file contains data which is shared between files internal to the [XTmrCtr](#) component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"
```

[Go to the source code of this file.](#)

Variables

[XTmrCtr_Config](#) [XTmrCtr_ConfigTable](#) []

Variable Documentation

[XTmrCtr_Config](#) [XTmrCtr_ConfigTable](#)[()]

The timer/counter configuration table, sized by the number of instances defined in [xparameters.h](#).

Generated on 29 May 2003 for Xilinx Device Drivers

tmrctr/v1_00_b/src/xtmrctr_stats.c File Reference

Detailed Description

Contains function to get and clear statistics for the **XTmrCtr** component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"  
#include "xtmrctr.h"
```

Functions

```
void XTmrCtr_GetStats (XTmrCtr *InstancePtr, XTmrCtrStats *StatsPtr)  
void XTmrCtr_ClearStats (XTmrCtr *InstancePtr)
```

Function Documentation

```
void XTmrCtr_ClearStats( XTmrCtr * InstancePtr)
```

Clear the **XTmrCtrStats** structure for this driver.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

Returns:

None.

Note:

None.

```
void XTmrCtr_GetStats( XTmrCtr * InstancePtr,  
                      XTmrCtrStats * StatsPtr  
                      )
```

Get a copy of the **XTmrCtrStats** structure, which contains the current statistics for this driver.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

StatsPtr is a pointer to a **XTmrCtrStats** structure which will get a copy of current statistics.

Returns:

None.

Note:

None.

tmrctr/v1_00_b/src/xtmrctr_selftest.c File Reference

Detailed Description

Contains diagnostic/self-test functions for the [XTmrCtr](#) component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xtmrctr.h"  
#include "xtmrctr_i.h"
```

Functions

[XStatus](#) [XTmrCtr_SelfTest](#) ([XTmrCtr](#) *InstancePtr, [Xuint8](#) TmrCtrNumber)

Function Documentation

```
XStatus XTmrCtr_SelfTest( XTmrCtr * InstancePtr,  
                        Xuint8   TmrCtrNumber  
                        )
```

Runs a self-test on the driver/device. This test verifies that the specified timer counter of the device can be enabled and increments.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

TmrCtrNumber is the timer counter of the device to operate on. Each device may contain multiple timer counters. The timer number is a zero based number with a range of 0 - (XTC_DEVICE_TIMER_COUNT - 1).

Returns:

XST_SUCCESS if self-test was successful, or XST_FAILURE if the timer is not incrementing.

Note:

This is a destructive test using the provided timer. The current settings of the timer are returned to the initialized values and all settings at the time this function is called are overwritten.

tmrctr/v1_00_b/src/xtmrctr_intr.c File Reference

Detailed Description

Contains interrupt-related functions for the **XTmrCtr** component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"  
#include "xtmrctr.h"  
#include "xtmrctr_1.h"
```

Functions

```
void XTmrCtr_SetHandler (XTmrCtr *InstancePtr, XTmrCtr_Handler FuncPtr, void *CallBackRef)  
void XTmrCtr_InterruptHandler (void *InstancePtr)
```

Function Documentation

```
void XTmrCtr_InterruptHandler( void * InstancePtr)
```

Interrupt Service Routine (ISR) for the driver. This function only performs processing for the device and does not save and restore the interrupt context.

Parameters:

InstancePtr contains a pointer to the timer/counter instance for the interrupt.

Returns:

None.

Note:

None.

```
void XTmrCtr_SetHandler( XTmrCtr *      InstancePtr,  
                        XTmrCtr_Handler FuncPtr,  
                        void *        CallbackRef  
                        )
```

Sets the timer callback function, which the driver calls when the specified timer times out.

Parameters:

InstancePtr is a pointer to the **XTmrCtr** instance to be worked on.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

FuncPtr is the pointer to the callback function.

Returns:

None.

Note:

The handler is called within interrupt context so the function that is called should either be short or pass the more extensive processing off to another task to allow the interrupt to return and normal processing to continue.

tmrctr/v1_00_b/src/xtmrctr_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of timer/counter devices in the system. Each timer/counter device should have an entry in this table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/16/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xtmrctr.h"  
#include "xparameters.h"
```

Variables

[XTmrCtr_Config](#) [XTmrCtr_ConfigTable](#) [XPAR_XTMRCTR_NUM_INSTANCES]

Variable Documentation

[XTmrCtr_Config](#) [XTmrCtr_ConfigTable](#)[XPAR_XTMRCTR_NUM_INSTANCES]

The timer/counter configuration table, sized by the number of instances defined in [xparameters.h](#).

Generated on 29 May 2003 for Xilinx Device Drivers

touchscreen_ref/v1_00_a/src/xtouchscreen.h

File Reference

Detailed Description

This driver supports the following features:

- Polled mode
- Interrupt driven mode

Interrupts

In order to use interrupts, it is necessary for the user to connect the driver interrupt handler, [XTouchscreen_InterruptHandler\(\)](#), to the interrupt system of the application. This function does not save and restore the processor context such that the user must provide it. A handler must be set for the driver such that the handler is called when interrupt events occur. The handler is called from interrupt context and is designed to allow application specific processing to be performed.

Note:

None.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ch	08/15/02	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xtouchscreen_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XTouchscreen
struct XTouchscreen_Config
```

Functions

```
XStatus XTouchscreen_Initialize (XTouchscreen *InstancePtr, Xuint16 DeviceId)
XTouchscreen_Config * XTouchscreen_LookupConfig (Xuint16 DeviceId)
void XTouchscreen_GetPosition_2D (XTouchscreen *InstancePtr, Xuint32 *x, Xuint32
    *y)
void XTouchscreen_GetPosition_3D (XTouchscreen *InstancePtr, Xuint32 *x, Xuint32 *y,
    Xuint32 *z)
void XTouchscreen_SetHandler (XTouchscreen *InstancePtr, XTouchscreen_Handler
    FuncPtr, void *CallbackRef)
void XTouchscreen_InterruptHandler (XTouchscreen *InstancePtr)
```

Function Documentation

```
void XTouchscreen_GetPosition_2D( XTouchscreen * InstancePtr,
    Xuint32 * x,
    Xuint32 * y
    )
```

This function reads 2D (X & Y) coordinates from the touchscreen.

Parameters:

InstancePtr is a pointer to the XTouchscreen instance to be worked on.

x pointer to store the x coordinate. *y* pointer to store the y coordinate.

Returns:

None.

Note:

None.


```
void XTouchscreen_GetPosition_3D( XTouchscreen * InstancePtr,  
                                Xuint32 * x,  
                                Xuint32 * y,  
                                Xuint32 * z  
                                )
```

This function reads 3D (X, Y & Z) coordinates from the touchscreen. x and y are the actual positions, the pressure has to be calculated from z1 and z2

Parameters:

InstancePtr is a pointer to the XTouchscreen instance to be worked on.

x pointer to store the x coordinate. **y* pointer to store the y coordinate. **z* pointer to store the z coordinate (pressure)

Returns:

None.

Note:

None.

```
XStatus XTouchscreen_Initialize( XTouchscreen * InstancePtr,  
                                Xuint16 DeviceId  
                                )
```

Initializes a specific touchscreen instance such that it is ready to be used. The default operating mode of the driver is polled mode.

Parameters:

InstancePtr is a pointer to the XTouchscreen instance to be worked on.

DeviceId is the unique id of the device controlled by this XTouchscreen instance. Passing in a device id associates the generic XTouchscreen instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if the device ID could not be found in the configuration table

Note:

None.

```
void XTouchscreen_InterruptHandler( XTouchscreen * InstancePtr)
```

This function is the interrupt handler for the Touchscreen driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any PS/2 port occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the Touchscreen instance.

Returns:

None.

Note:

None.

XTouchscreen_Config* XTouchscreen_LookupConfig([Xuint16](#) *DeviceId*)

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found.

Note:

None.

```
void XTouchscreen_SetHandler( XTouchscreen *      InstancePtr,
                             XTouchscreen_Handler FuncPtr,
                             void *              CallbackRef
                             )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the XTouchscreen instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

touchscreen_ref/v1_00_a/src/xtouchscreen_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: touchscreen_l.h,v 1.3 2002/10/24 00:01:12 carsten Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file touchscreen_ref/v1_00_a/src/xtouchscreen_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  The user should refer to the
00029 * hardware device specification for more details of the device operation.
00030 * High-level driver functions are defined in touchscreen.h.
00031 *
00032 * <pre>
00033 * MODIFICATION HISTORY:
00034 *
00035 * Ver   Who   Date       Changes
00036 * ----  ---  -
00037 * 1.00a ch   08/08/02 First release
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XTOUCHSCREEN_L_H /* prevent circular inclusions */
```

```

00043 #define XTOUCHSCREEN_L_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048 #include "xio.h"
00049
00050 /***** Constant Definitions *****/
00051
00052 /* Register offsets */
00053 #define XTOUCHSCREEN_CTRL_REG_OFFSET 0
00054 #define XTOUCHSCREEN_INTR_REG_OFFSET 4
00055
00056 /* control register channel selectors */
00057 #define XTOUCHSCREEN_CTRL_CHSEL_X 0xd3
00058 #define XTOUCHSCREEN_CTRL_CHSEL_Y 0x93
00059 #define XTOUCHSCREEN_CTRL_CHSEL_Z1 0xb3
00060 #define XTOUCHSCREEN_CTRL_CHSEL_Z2 0xc3
00061
00062 /* interrupt register positions */
00063 #define XTOUCHSCREEN_INT_PEN_UP 0x01
00064 #define XTOUCHSCREEN_INT_PEN_DOWN 0x02
00065
00066 /* error reading touchscreen */
00067 #define XTOUCHSCREEN_SAMPLE_ERROR 0xffffffff
00068
00069 /***** Type Definitions *****/
00070
00071 /***** Macros (Inline Functions) Definitions *****/
00072
00073 /*****
00074 *
00075 * Low-level driver macros. The list below provides signatures to help the
00076 * user use the macros.
00077 *
00078 * Xuint32 XTouchscreen_mReadCtrlReg(Xuint32 BaseAddress)
00079 * void XTouchscreen_mWriteCtrlReg(Xuint32 BaseAddress, Xuint8 Value)
00080 *
00081 * Xuint32 XTouchscreen_mGetIntrStatus(Xuint32 BaseAddress)
00082 * void XTouchscreen_mClearIntr(Xuint32 BaseAddress, Xuint32 ClearMask)
00083 *
00084 *****/
00085
00086 /*****
00087 **
00088 * Read the control register.
00089 *
00090 * @param BaseAddress contains the base address of the device.
00091 *
00092 * @return The value read from the register.
00093 *
00094 * @note None.

```

```

00095 *
00096 *****/
00097 #define XTouchscreen_mReadCtrlReg(BaseAddress) \
00098     XIo_In32((BaseAddress) + XTOUCHSCREEN_CTRL_REG_OFFSET)
00099
00100 /**
00101 **
00102 * Write to the control register
00103 *
00104 * @param BaseAddress contains the base address of the device.
00105 *        Value to be written
00106 *
00107 * @return None.
00108 *
00109 * @note None.
00110 *
00111 *****/
00112 #define XTouchscreen_mWriteCtrlReg(BaseAddress, Value) \
00113     XIo_Out8((BaseAddress) + XTOUCHSCREEN_CTRL_REG_OFFSET, Value)
00114
00115 /**
00116 **
00117 * Read the interrupt status register.
00118 *
00119 * @param BaseAddress contains the base address of the device.
00120 *
00121 * @return The value read from the register.
00122 *
00123 * @note None.
00124 *
00125 *****/
00126 #define XTouchscreen_mGetIntrStatus(BaseAddress) \
00127     XIo_In32((BaseAddress) + XTOUCHSCREEN_INTR_REG_OFFSET)
00128
00129 /**
00130 **
00131 * Clear pending interrupts.
00132 *
00133 * @param BaseAddress contains the base address of the device.
00134 *        Bitmask for interrupts to be cleared. A "1" clears the interrupt.
00135 *
00136 * @return None.
00137 *
00138 * @note None.
00139 *
00140 *****/
00141 #define XTouchscreen_mClearIntr(BaseAddress, ClearMask) \
00142     XIo_Out32((BaseAddress) + XTOUCHSCREEN_INTR_REG_OFFSET, ClearMask)
00143
00144 /** Variable Definitions *****/
00145
00146 /** Function Prototypes *****/

```

```
00147
00148 Xuint32 XTouchscreen_GetValue(Xuint32 BaseAddress, Xuint8 Channel);
00149
00150 /*****
00151
00152 #endif
```

Generated on 29 May 2003 for Xilinx Device Drivers

touchscreen_ref/v1_00_a/src/xtouchscreen_l.h

File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xtouchscreen.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	08/08/02	First release

```
#include "xbasic_types.h"
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XTouchscreen_mReadCtrlReg(BaseAddress)
#define XTouchscreen_mWriteCtrlReg(BaseAddress, Value)
#define XTouchscreen_mGetIntrStatus(BaseAddress)
#define XTouchscreen_mClearIntr(BaseAddress, ClearMask)
```

Functions

[Xuint32](#) [XTouchscreen_GetValue](#) ([Xuint32](#) BaseAddress, [Xuint8](#) Channel)

Define Documentation

```
#define XTouchscreen_mClearIntr( BaseAddress,  
                                ClearMask  )
```

Clear pending interrupts.

Parameters:

BaseAddress contains the base address of the device. Bitmask for interrupts to be cleared. A "1" clears the interrupt.

Returns:

None.

Note:

None.

```
#define XTouchscreen_mGetIntrStatus( BaseAddress )
```

Read the interrupt status register.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XTouchscreen_mReadCtrlReg( BaseAddress )
```

Read the control register.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XTouchscreen_mWriteCtrlReg( BaseAddress,  
                                   Value      )
```

Write to the control register

Parameters:

BaseAddress contains the base address of the device. Value to be written

Returns:

None.

Note:

None.

Function Documentation

```
Xuint32 XTouchscreen_GetValue( Xuint32 BaseAddress,  
                               Xuint8  Channel  
                               )
```

This function reads a digitized value from the touchscreen controller.

Parameters:

BaseAddress contains the base address of the touchscreen controller.

Channel addresses the channel to be read.

Returns:

Digitized value of the addressed channel.

Note:

None.

touchscreen_ref/v1_00_a/src/xtouchscreen_l.c

File Reference

Detailed Description

This file contains low-level driver functions that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	08/08/02	First release

```
#include "xbasic_types.h"  
#include "xenv.h"  
#include "xtouchscreen_l.h"
```

Functions

Xuint32 [XTouchscreen_GetValue](#) (**Xuint32** BaseAddress, **Xuint8** Channel)

Function Documentation

```
Xuint32 XTouchscreen_GetValue( Xuint32 BaseAddress,  
                               Xuint8 Channel  
                               )
```

This function reads a digitized value from the touchscreen controller.

Parameters:

BaseAddress contains the base address of the touchscreen controller.

Channel addresses the channel to be read.

Returns:

Digitized value of the addressed channel.

Note:

None.

touchscreen_ref/v1_00_a/src/xtouchscreen.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: touchscreen.h,v 1.4 2002/10/25 23:18:10 carsten Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file touchscreen_ref/v1_00_a/src/xtouchscreen.h
00026 *
00027 * This driver supports the following features:
00028 *
00029 * - Polled mode
00030 * - Interrupt driven mode
00031 *
00032 * <b>Interrupts</b>
00033 *
00034 * In order to use interrupts, it is necessary for the user to connect the
00035 * driver interrupt handler, XTouchscreen_InterruptHandler(), to the interrupt
00036 * system of the application. This function does not save and restore the
00037 * processor context such that the user must provide it. A handler must be set
00038 * for the driver such that the handler is called when interrupt events occur.
00039 * The handler is called from interrupt context and is designed to allow
00040 * application specific processing to be performed.
00041 *
00042 * @note
```

```

00043 *
00044 * None.
00045 *
00046 * <pre>
00047 * MODIFICATION HISTORY:
00048 *
00049 * Ver    Who    Date      Changes
00050 * -----
00051 * 1.00a ch   08/15/02 First release
00052 * </pre>
00053 *
00054 *****/
00055
00056 #ifndef XTOUCHSCREEN_H /* prevent circular inclusions */
00057 #define XTOUCHSCREEN_H /* by using protection macros */
00058
00059 /***** Include Files *****/
00060
00061 #include "xbasic_types.h"
00062 #include "xstatus.h"
00063 #include "xtouchscreen_1.h"
00064
00065 /***** Constant Definitions *****/
00066
00067 /*
00068 * These constants specify the handler events that are passed to
00069 * a handler from the driver. These constants are not bit masks such that
00070 * only one will be passed at a time to the handler
00071 */
00072 #define XTOUCHSCREEN_EVENT_PEN_UP      1
00073 #define XTOUCHSCREEN_EVENT_PEN_DOWN   2
00074
00075 /*
00076 * These constants are used to specify the current pen state.
00077 * They are only used internally.
00078 */
00079 #define XTOUCHSCREEN_STATE_PEN_UP      1
00080 #define XTOUCHSCREEN_STATE_PEN_DOWN   2
00081
00082 /***** Type Definitions *****/
00083
00084 /*
00085 * This typedef contains configuration information for the device
00086 */
00087 typedef struct
00088 {
00089     Xuint16 DeviceId;
00090     Xuint32 BaseAddress;
00091 } XTouchscreen_Config;
00092
00093 /*
00094 * This data type defines a handler which the application must define

```

```

00095 * when using interrupt mode. The handler will be called from the driver in an
00096 * interrupt context to handle application specific processing.
00097 *
00098 * @param CallbackRef is a callback reference passed in by the upper layer
00099 *         when setting the handler, and is passed back to the upper layer when
00100 *         the handler is called.
00101 * @param Event contains one of the event constants indicating why the handler
00102 *         is being called.
00103 * @param EventData contains the number of bytes sent or received at the time
00104 *         of the call.
00105 */
00106 typedef void (*XTouchscreen_Handler)(void *CallbackRef, Xuint32 Event,
00107                                     unsigned int EventData);
00108 /*
00109 * The Touchscreen driver instance data. The user is required to allocate a
00110 * variable of this type for every touchscreen device in the system.
00111 * A pointer to a variable of this type is then passed to the driver API
00112 * functions
00113 */
00114 typedef struct
00115 {
00116     Xuint32 BaseAddress;
00117     Xuint32 IsReady;
00118     Xuint32 CurrentState;
00119     XTouchscreen_Handler Handler;
00120     void *CallbackRef;
00121 } XTouchscreen;
00122
00123 /***** Macros (Inline Functions) Definitions *****/
00124
00125 /***** Function Prototypes *****/
00126
00127 /*
00128 * required functions is xtouchscreen.c
00129 */
00130 XStatus XTouchscreen_Initialize(XTouchscreen *InstancePtr, Xuint16 DeviceId);
00131 XTouchscreen_Config *XTouchscreen_LookupConfig(Xuint16 DeviceId);
00132 void XTouchscreen_GetPosition_2D(XTouchscreen *InstancePtr, Xuint32 *x,
00133                                 Xuint32 *y);
00134 void XTouchscreen_GetPosition_3D(XTouchscreen *InstancePtr, Xuint32 *x,
00135                                 Xuint32 *y, Xuint32 *z);
00136 /*
00137 * interrupt functions in xtouchscreen_intr.c
00138 */
00139 void XTouchscreen_SetHandler(XTouchscreen *InstancePtr,
00140                             XTouchscreen_Handler FuncPtr, void *CallbackRef);
00141 void XTouchscreen_InterruptHandler(XTouchscreen *InstancePtr);
00142
00143 #endif

```


touchscreen_ref/v1_00_a/src/xtouchscreen.c

File Reference

Detailed Description

This file contains the required functions for the touchscreen driver. Refer to the header file [xtouchscreen.h](#) for more detailed information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	08/15/02	First release
1.00a	rmm	05/15/03	Fixed diab compiler warnings relating to asserts

```
#include "xenv.h"  
#include "xstatus.h"  
#include "xparameters.h"  
#include "xtouchscreen.h"  
#include "xtouchscreen_i.h"  
#include "xtouchscreen_l.h"  
#include "xio.h"
```

Functions

```
XStatus XTouchscreen_Initialize (XTouchscreen *InstancePtr, Xuint16 DeviceId)  
XTouchscreen_Config * XTouchscreen_LookupConfig (Xuint16 DeviceId)  
void XTouchscreen_GetPosition_2D (XTouchscreen *InstancePtr, Xuint32 *x, Xuint32 *y)  
void XTouchscreen_GetPosition_3D (XTouchscreen *InstancePtr, Xuint32 *x, Xuint32 *y, Xuint32 *z)
```

Function Documentation

```
void XTouchscreen_GetPosition_2D( XTouchscreen * InstancePtr,  
                                Xuint32 * x,  
                                Xuint32 * y  
                                )
```

This function reads 2D (X & Y) coordinates from the touchscreen.

Parameters:

InstancePtr is a pointer to the XTouchscreen instance to be worked on.

x pointer to store the x coordinate. **y* pointer to store the y coordinate.

Returns:

None.

Note:

None.

```
void XTouchscreen_GetPosition_3D( XTouchscreen * InstancePtr,  
                                Xuint32 * x,  
                                Xuint32 * y,  
                                Xuint32 * z  
                                )
```

This function reads 3D (X, Y & Z) coordinates from the touchscreen. *x* and *y* are the actual positions, the pressure has to be calculated from *z1* and *z2*

Parameters:

InstancePtr is a pointer to the XTouchscreen instance to be worked on.

x pointer to store the x coordinate. **y* pointer to store the y coordinate. **z* pointer to store the z coordinate (pressure)

Returns:

None.

Note:

None.

```
XStatus XTouchscreen_Initialize( XTouchscreen * InstancePtr,  
                                Xuint16 DeviceId  
                                )
```

Initializes a specific touchscreen instance such that it is ready to be used. The default operating mode of the driver is polled mode.

Parameters:

InstancePtr is a pointer to the XTouchscreen instance to be worked on.

DeviceId is the unique id of the device controlled by this XTouchscreen instance. Passing in a device id associates the generic XTouchscreen instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if the device ID could not be found in the configuration table

Note:

None.

```
XTouchscreen_Config* XTouchscreen_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found.

Note:

None.

touchscreen_ref/v1_00_a/src/xtouchscreen_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: touchscreen_i.h,v 1.1 2002/08/15 22:17:01 carsten Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 **
00024 *
00025 * @file touchscreen_ref/v1_00_a/src/xtouchscreen_i.h
00026 *
00027 * This header file contains internal identifiers, which are those shared
00028 * between the files of the driver. It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date        Changes
00034 * -----
00035 * 1.00a ch    08/15/02 First release
00036 * </pre>
00037 *
00038 *****/
00039 #ifndef XTOUCHSCREEN_I_H /* prevent circular inclusions */
00040 #define XTOUCHSCREEN_I_H /* by using protection macros */
00041
00042 /***** Include Files *****/
```

```
00043
00044 #include "xtouchscreen.h"
00045
00046 /***** Constant Definitions *****/
00047
00048 /***** Type Definitions *****/
00049
00050 /***** Macros (Inline Functions) Definitions *****/
00051
00052 /***** Variable Definitions *****/
00053
00054 extern XTouchscreen_Config XTouchscreen_ConfigTable[];
00055
00056 /***** Function Prototypes *****/
00057
00058 #endif
```

Generated on 29 May 2003 for Xilinx Device Drivers

touchscreen_ref/v1_00_a/src/xtouchscreen_i.h

File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between the files of the driver. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	08/15/02	First release

```
#include "xtouchscreen.h"
```

[Go to the source code of this file.](#)

touchscreen_ref/v1_00_a/src/xtouchscreen_intr.c

File Reference

Detailed Description

This file contains the functions that are related to interrupt processing for the touchscreen driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ch	08/15/02	First release

```
#include "xio.h"  
#include "xtouchscreen.h"  
#include "xtouchscreen_1.h"
```

Functions

```
void XTouchscreen\_SetHandler (XTouchscreen *InstancePtr, XTouchscreen_Handler FuncPtr, void *CallBackRef)  
void XTouchscreen\_InterruptHandler (XTouchscreen *InstancePtr)
```

Function Documentation

```
void XTouchscreen_InterruptHandler( XTouchscreen * InstancePtr)
```

This function is the interrupt handler for the Touchscreen driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any PS/2 port occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the Touchscreen instance.

Returns:

None.

Note:

None.

```
void XTouchscreen_SetHandler( XTouchscreen *      InstancePtr,
                             XTouchscreen_Handler FuncPtr,
                             void *              CallbackRef
                             )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the XTouchscreen instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

uartlite/v1_00_b/src/xuartlite_stats.c File Reference

Detailed Description

This file contains the statistics functions for the UART Lite component ([XUartLite](#)).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xuartlite.h"  
#include "xuartlite_i.h"
```

Functions

```
void XUartLite_GetStats (XUartLite *InstancePtr, XUartLite_Stats *StatsPtr)  
void XUartLite_ClearStats (XUartLite *InstancePtr)
```

Function Documentation

```
void XUartLite_ClearStats( XUartLite * InstancePtr)
```

This function zeros the statistics for the given instance.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
void XUartLite_GetStats( XUartLite * InstancePtr,  
                        XUartLite_Stats * StatsPtr  
                        )
```

Returns a snapshot of the current statistics in the structure specified.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

StatsPtr is a pointer to a XUartLiteStats structure to where the statistics are to be copied.

Returns:

None.

Note:

None.

XUartLite Struct Reference

```
#include <xuartlite.h>
```

Detailed Description

The XUartLite driver instance data. The user is required to allocate a variable of this type for every UART Lite device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [uartlite/v1_00_b/src/xuartlite.h](#)

uartlite/v1_00_b/src/xuartlite.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xuartlite.h,v 1.10 2002/07/24 22:35:28 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file uartlite/v1_00_b/src/xuartlite.h
00026 *
00027 * This component contains the implementation of the XUartLite component which
00028 is
00029 * the driver for the Xilinx UART Lite device. This UART is a minimal hardware
00030 * implementation with minimal features. Most of the features, including baud
00031 * rate, parity, and number of data bits are only configurable when the hardware
00032 * device is built, rather than at run time by software.
00033 *
00034 * The device has 16 byte transmit and receive FIFOs and supports interrupts.
00035 * The device does not have any way to disable the receiver such that the
00036 * receive FIFO may contain unwanted data. The FIFOs are not flushed when the
00037 * driver is initialized, but a function is provided to allow the user to
00038 * reset the FIFOs if desired.
00039 *
00040 * The driver defaults to no interrupts at initialization such that interrupts
00041 * must be enabled if desired. An interrupt is generated when the transmit FIFO
00042 * transitions from having data to being empty or when any data is contained in
```

```

00042 * the receive FIFO.
00043 *
00044 * In order to use interrupts, it's necessary for the user to connect the driver
00045 * interrupt handler, XUartLite_InterruptHandler, to the interrupt system of the
00046 * application. This function does not save and restore the processor context
00047 * such that the user must provide it. Send and receive handlers may be set for
00048 * the driver such that the handlers are called when transmit and receive
00049 * interrupts occur. The handlers are called from interrupt context and are
00050 * designed to allow application specific processing to be performed.
00051 *
00052 * The functions, XUartLite_Send and XUartLite_Recv, are provided in the driver
00053 * to allow data to be sent and received. They are designed to be used in
00054 * polled or interrupt modes.
00055 *
00056 * The driver provides a status for each received byte indicating any parity
00057 * frame or overrun error. The driver provides statistics which allow visibility
00058 * into these errors.
00059 *
00060 * <b>RTOS Independence</b>
00061 *
00062 * This driver is intended to be RTOS and processor independent. It works
00063 * with physical addresses only. Any needs for dynamic memory management,
00064 * threads or thread mutual exclusion, virtual memory, or cache control must
00065 * be satisfied by the layer above this driver.
00066 *
00067 * @note
00068 *
00069 * The driver is partitioned such that a minimal implementation may be used.
00070 * More features require additional files to be linked in.
00071 *
00072 * <pre>
00073 * MODIFICATION HISTORY:
00074 *
00075 * Ver    Who    Date      Changes
00076 * -----
00077 * 1.00a  ecm    08/31/01  First release
00078 * 1.00b  jhl    02/21/02  Repartitioned the driver for smaller files
00079 * </pre>
00080 *
00081 * *****/
00082
00083 #ifndef XUARTLITE_H /* prevent circular inclusions */
00084 #define XUARTLITE_H /* by using protection macros */
00085
00086 /***** Include Files *****/
00087
00088 #include "xbasic_types.h"
00089 #include "xstatus.h"
00090
00091 /***** Constant Definitions *****/
00092
00093 /***** Type Definitions *****/

```

```

00094
00095 /**
00096  * Callback function. The first argument is a callback reference passed in by
00097  * the upper layer when setting the callback functions, and passed back to the
00098  * upper layer when the callback is invoked.
00099  * The second argument is the ByteCount which is the number of bytes that
00100  * actually moved from/to the buffer provided in the _Send/_Receive call.
00101  */
00102 typedef void (*XUartLite_Handler)(void *CallBackRef, unsigned int ByteCount);
00103
00104 /**
00105  * Statistics for the XUartLite driver
00106  */
00107 typedef struct
00108 {
00109     Xuint32 TransmitInterrupts;    /**< Number of transmit interrupts */
00110     Xuint32 ReceiveInterrupts;    /**< Number of receive interrupts */
00111     Xuint32 CharactersTransmitted; /**< Number of characters transmitted */
00112     Xuint32 CharactersReceived;   /**< Number of characters received */
00113     Xuint32 ReceiveOverrunErrors; /**< Number of receive overruns */
00114     Xuint32 ReceiveParityErrors;  /**< Number of receive parity errors */
00115     Xuint32 ReceiveFramingErrors; /**< Number of receive framing errors */
00116 } XUartLite_Stats;
00117
00118 /**
00119  * The following data type is used to manage the buffers that are handled
00120  * when sending and receiving data in the interrupt mode. It is intended
00121  * for internal use only.
00122  */
00123 typedef struct
00124 {
00125     Xuint8 *NextBytePtr;
00126     unsigned int RequestedBytes;
00127     unsigned int RemainingBytes;
00128 } XUartLite_Buffer;
00129
00130 /**
00131  * This typedef contains configuration information for the device.
00132  */
00133 typedef struct
00134 {
00135     Xuint16 DeviceId;    /**< Unique ID of device */
00136     Xuint32 RegBaseAddr; /**< Register base address */
00137     Xuint32 BaudRate;    /**< Fixed baud rate */
00138     Xuint8 UseParity;    /**< Parity generator enabled when XTRUE */
00139     Xuint8 ParityOdd;    /**< Parity generated is odd when XTRUE, even when
00140                          XFALSE */
00141     Xuint8 DataBits;    /**< Fixed data bits */
00142 } XUartLite_Config;
00143

```

```

00144 /**
00145  * The XUartLite driver instance data. The user is required to allocate a
00146  * variable of this type for every UART Lite device in the system. A pointer
00147  * to a variable of this type is then passed to the driver API functions.
00148  */
00149 typedef struct
00150 {
00151     XUartLite_Stats Stats;          /* Component Statistics */
00152     Xuint32 RegBaseAddress;        /* Base address of registers */
00153     Xuint32 IsReady;              /* Device is initialized and ready */
00154
00155
00156     XUartLite_Buffer SendBuffer;
00157     XUartLite_Buffer ReceiveBuffer;
00158
00159     XUartLite_Handler RecvHandler;
00160     void *RecvCallBackRef;        /* Callback reference for recv handler */
00161     XUartLite_Handler SendHandler;
00162     void *SendCallBackRef;       /* Callback reference for send handler */
00163 } XUartLite;
00164
00165
00166 /***** Macros (Inline Functions) Definitions *****/
00167
00168
00169 /***** Function Prototypes *****/
00170
00171 /*
00172  * Required functions, in file xuart.c
00173  */
00174 XStatus XUartLite_Initialize(XUartLite *InstancePtr, Xuint16 DeviceId);
00175
00176 void XUartLite_ResetFifos(XUartLite *InstancePtr);
00177
00178 unsigned int XUartLite_Send(XUartLite *InstancePtr, Xuint8 *DataBufferPtr,
00179                             unsigned int NumBytes);
00180 unsigned int XUartLite_Recv(XUartLite *InstancePtr, Xuint8 *DataBufferPtr,
00181                             unsigned int NumBytes);
00182
00183 Xboolean XUartLite_IsSending(XUartLite *InstancePtr);
00184 XUartLite_Config *XUartLite_LookupConfig(Xuint16 DeviceId);
00185
00186 /*
00187  * Functions for statistics, in file xuartlite_stats.c
00188  */
00189 void XUartLite_GetStats(XUartLite *InstancePtr, XUartLite_Stats *StatsPtr);
00190 void XUartLite_ClearStats(XUartLite *InstancePtr);
00191
00192 /*
00193  * Functions for self-test, in file xuartlite_selftest.c

```

```
00194  */
00195 XStatus XUartLite_SelfTest(XUartLite *InstancePtr);
00196
00197 /*
00198  * Functions for interrupts, in file xuartlite_intr.c
00199  */
00200 void XUartLite_EnableInterrupt(XUartLite *InstancePtr);
00201 void XUartLite_DisableInterrupt(XUartLite *InstancePtr);
00202
00203 void XUartLite_SetRecvHandler(XUartLite *InstancePtr, XUartLite_Handler
FuncPtr,
00204                               void *CallBackRef);
00205 void XUartLite_SetSendHandler(XUartLite *InstancePtr, XUartLite_Handler
FuncPtr,
00206                               void *CallBackRef);
00207
00208 void XUartLite_InterruptHandler(XUartLite *InstancePtr);
00209
00210 #endif                /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

uartlite/v1_00_b/src/xuartlite.h File Reference

Detailed Description

This component contains the implementation of the **XUartLite** component which is the driver for the Xilinx UART Lite device. This UART is a minimal hardware implementation with minimal features. Most of the features, including baud rate, parity, and number of data bits are only configurable when the hardware device is built, rather than at run time by software.

The device has 16 byte transmit and receive FIFOs and supports interrupts. The device does not have any way to disable the receiver such that the receive FIFO may contain unwanted data. The FIFOs are not flushed when the driver is initialized, but a function is provided to allow the user to reset the FIFOs if desired.

The driver defaults to no interrupts at initialization such that interrupts must be enabled if desired. An interrupt is generated when the transmit FIFO transitions from having data to being empty or when any data is contained in the receive FIFO.

In order to use interrupts, it's necessary for the user to connect the driver interrupt handler, XUartLite_InterruptHandler, to the interrupt system of the application. This function does not save and restore the processor context such that the user must provide it. Send and receive handlers may be set for the driver such that the handlers are called when transmit and receive interrupts occur. The handlers are called from interrupt context and are designed to allow application specific processing to be performed.

The functions, XUartLite_Send and XUartLite_Recv, are provided in the driver to allow data to be sent and received. They are designed to be used in polled or interrupt modes.

The driver provides a status for each received byte indicating any parity frame or overrun error. The driver provides statistics which allow visibility into these errors.

RTOS Independence

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

Note:

The driver is partitioned such that a minimal implementation may be used. More features require additional files to be linked in.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XUartLite  
struct XUartLite_Buffer  
struct XUartLite_Config  
struct XUartLite_Stats
```

Typedefs

```
typedef void(* XUartLite_Handler)(void *CallBackRef, unsigned int ByteCount)
```

Functions

XStatus XUartLite_Initialize (**XUartLite** *InstancePtr, **Xuint16** DeviceId)
void **XUartLite_ResetFifos** (**XUartLite** *InstancePtr)
unsigned int **XUartLite_Send** (**XUartLite** *InstancePtr, **Xuint8** *DataBufferPtr, unsigned int NumBytes)
unsigned int **XUartLite_Recv** (**XUartLite** *InstancePtr, **Xuint8** *DataBufferPtr, unsigned int NumBytes)
Xboolean XUartLite_IsSending (**XUartLite** *InstancePtr)
void **XUartLite_GetStats** (**XUartLite** *InstancePtr, **XUartLite_Stats** *StatsPtr)
void **XUartLite_ClearStats** (**XUartLite** *InstancePtr)
XStatus XUartLite_SelfTest (**XUartLite** *InstancePtr)
void **XUartLite_EnableInterrupt** (**XUartLite** *InstancePtr)
void **XUartLite_DisableInterrupt** (**XUartLite** *InstancePtr)
void **XUartLite_SetRecvHandler** (**XUartLite** *InstancePtr, **XUartLite_Handler** FuncPtr, void *CallbackRef)
void **XUartLite_SetSendHandler** (**XUartLite** *InstancePtr, **XUartLite_Handler** FuncPtr, void *CallbackRef)
void **XUartLite_InterruptHandler** (**XUartLite** *InstancePtr)

Typedef Documentation

typedef void(* XUartLite_Handler)(void *CallbackRef, unsigned int ByteCount)

Callback function. The first argument is a callback reference passed in by the upper layer when setting the callback functions, and passed back to the upper layer when the callback is invoked. The second argument is the ByteCount which is the number of bytes that actually moved from/to the buffer provided in the _Send/_Receive call.

Function Documentation

void XUartLite_ClearStats(XUartLite * InstancePtr)

This function zeros the statistics for the given instance.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

void XUartLite_DisableInterrupt(XUartLite * InstancePtr)

This function disables the UART interrupt. After calling this function, data may still be received by the UART but no interrupt will be generated since the hardware device has no way to disable the receiver.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

void XUartLite_EnableInterrupt(XUartLite * InstancePtr)

This function enables the UART interrupt such that an interrupt will occur when data is received or data has been transmitted. The device contains 16 byte receive and transmit FIFOs such that an interrupt is generated anytime there is data in the receive FIFO and when the transmit FIFO transitions from not empty to empty.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
void XUartLite_GetStats( XUartLite * InstancePtr,  
                        XUartLite_Stats * StatsPtr  
                        )
```

Returns a snapshot of the current statistics in the structure specified.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

StatsPtr is a pointer to a XUartLiteStats structure to where the statistics are to be copied.

Returns:

None.

Note:

None.

```
XStatus XUartLite_Initialize( XUartLite * InstancePtr,  
                              Xuint16 DeviceId  
                              )
```

Initialize a **XUartLite** instance. The receive and transmit FIFOs of the UART are not flushed, so the user may want to flush them. The hardware device does not have any way to disable the receiver such that any valid data may be present in the receive FIFO. This function disables the UART interrupt. The baudrate and format of the data are fixed in the hardware at hardware build time.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XUartLite** instance. Passing in a device id associates the generic **XUartLite** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the device is not found in the configuration table.

Note:

None.

```
void XUartLite_InterruptHandler( XUartLite * InstancePtr)
```

This function is the interrupt handler for the UART lite driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any UART lite occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the UART that the interrupt is for.

Returns:

None.

Note:

None.

```
Xboolean XUartLite_IsSending( XUartLite * InstancePtr)
```

This function determines if the specified UART is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

A value of XTRUE if the UART is sending data, otherwise XFALSE.

Note:

None.

```
unsigned int XUartLite_Recv( XUartLite * InstancePtr,  
                             Xuint8 *   DataBufferPtr,  
                             unsigned int NumBytes  
                             )
```

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer in the FIFO. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```
void XUartLite_ResetFifos( XUartLite * InstancePtr)
```

This function resets the FIFOs, both transmit and receive, of the UART such that they are emptied. Since the UART does not have any way to disable it from receiving data, it may be necessary for the application to reset the FIFOs to get rid of any unwanted data.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
XStatus XUartLite_SelfTest( XUartLite * InstancePtr)
```

Runs a self-test on the device hardware. Since there is no way to perform a loopback in the hardware, this test can only check the state of the status register to verify it is correct. This test assumes that the hardware device is still in its reset state, but has been initialized with the Initialize function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

- XST_SUCCESS if the self-test was successful.
- XST_FAILURE if the self-test failed, the status register value was not correct

Note:

None.


```
unsigned int XUartLite_Send( XUartLite * InstancePtr,  
                             Xuint8 *   DataBufferPtr,  
                             unsigned int NumBytes  
                             )
```

This functions sends the specified buffer of data using the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART. If the UART is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send as much data as the UART can buffer in the FIFO. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.

NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```
void XUartLite_SetRecvHandler( XUartLite * InstancePtr,  
                               XUartLite_Handler FuncPtr,  
                               void *       CallbackRef  
                               )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

```
void XUartLite_SetSendHandler( XUartLite *      InstancePtr,  
                               XUartLite_Handler FuncPtr,  
                               void *          CallbackRef  
                               )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

XUartLite_Buffer Struct Reference

```
#include <xuartlite.h>
```

Detailed Description

The following data type is used to manage the buffers that are handled when sending and receiving data in the interrupt mode. It is intended for internal use only.

The documentation for this struct was generated from the following file:

- [uartlite/v1_00_b/src/xuartlite.h](#)
-

XUartLite_Config Struct Reference

```
#include <xuartlite.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

Xuint16 DeviceId

Xuint32 RegBaseAddr

Xuint32 BaudRate

Xuint8 UseParity

Xuint8 ParityOdd

Xuint8 DataBits

Field Documentation

Xuint32 XUartLite_Config::BaudRate

Fixed baud rate

Xuint8 XUartLite_Config::DataBits

Fixed data bits

Xuint16 XUartLite_Config::DeviceId

Unique ID of device

Xuint8 XUartLite_Config::ParityOdd

Parity generated is odd when XTRUE, even when XFALSE

Xuint32 XUartLite_Config::RegBaseAddr

Register base address

Xuint8 XUartLite_Config::UseParity

Parity generator enabled when XTRUE

The documentation for this struct was generated from the following file:

- [uartlite/v1_00_b/src/xuartlite.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XUartLite_Stats Struct Reference

```
#include <xuartlite.h>
```

Detailed Description

Statistics for the [XUartLite](#) driver

Data Fields

[Xuint32 TransmitInterrupts](#)

[Xuint32 ReceiveInterrupts](#)

[Xuint32 CharactersTransmitted](#)

[Xuint32 CharactersReceived](#)

[Xuint32 ReceiveOverrunErrors](#)

[Xuint32 ReceiveParityErrors](#)

[Xuint32 ReceiveFramingErrors](#)

Field Documentation

[Xuint32 XUartLite_Stats::CharactersReceived](#)

Number of characters received

[Xuint32 XUartLite_Stats::CharactersTransmitted](#)

Number of characters transmitted

Xuint32 XUartLite_Stats::ReceiveFramingErrors

Number of receive framing errors

Xuint32 XUartLite_Stats::ReceiveInterrupts

Number of receive interrupts

Xuint32 XUartLite_Stats::ReceiveOverrunErrors

Number of receive overruns

Xuint32 XUartLite_Stats::ReceiveParityErrors

Number of receive parity errors

Xuint32 XUartLite_Stats::TransmitInterrupts

Number of transmit interrupts

The documentation for this struct was generated from the following file:

- [uartlite/v1_00_b/src/xuartlite.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

uartlite/v1_00_b/src/xuartlite.c File Reference

Detailed Description

Contains required functions for the **XUartLite** driver. See the [xuartlite.h](#) header file for more details on this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files
1.00b	rmm	05/13/03	Fixed diab compiler warnings relating to asserts

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xparameters.h"  
#include "xuartlite.h"  
#include "xuartlite_i.h"  
#include "xio.h"
```

Functions

XStatus XUartLite_Initialize (**XUartLite** *InstancePtr, **Xuint16** DeviceId)


```
unsigned int XUartLite_Send (XUartLite *InstancePtr, Xuint8 *DataBufferPtr, unsigned int
    NumBytes)
unsigned int XUartLite_Recv (XUartLite *InstancePtr, Xuint8 *DataBufferPtr, unsigned int
    NumBytes)
void XUartLite_ResetFifos (XUartLite *InstancePtr)
Xboolean XUartLite_IsSending (XUartLite *InstancePtr)
unsigned int XUartLite_SendBuffer (XUartLite *InstancePtr)
unsigned int XUartLite_ReceiveBuffer (XUartLite *InstancePtr)
```

Function Documentation

```
XStatus XUartLite_Initialize( XUartLite * InstancePtr,
                             Xuint16   DeviceId
                             )
```

Initialize a **XUartLite** instance. The receive and transmit FIFOs of the UART are not flushed, so the user may want to flush them. The hardware device does not have any way to disable the receiver such that any valid data may be present in the receive FIFO. This function disables the UART interrupt. The baudrate and format of the data are fixed in the hardware at hardware build time.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XUartLite** instance. Passing in a device id associates the generic **XUartLite** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if everything starts up as expected.
- XST_DEVICE_NOT_FOUND if the device is not found in the configuration table.

Note:

None.

```
Xboolean XUartLite_IsSending( XUartLite * InstancePtr)
```

This function determines if the specified UART is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

A value of XTRUE if the UART is sending data, otherwise XFALSE.

Note:

None.

unsigned int XUartLite_ReceiveBuffer(XUartLite * InstancePtr)

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartLite** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

```
unsigned int XUartLite_Recv( XUartLite * InstancePtr,  
                             Xuint8 *   DataBufferPtr,  
                             unsigned int NumBytes  
                             )
```

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer in the FIFO. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```
void XUartLite_ResetFifos( XUartLite * InstancePtr)
```

This function resets the FIFOs, both transmit and receive, of the UART such that they are emptied. Since the UART does not have any way to disable it from receiving data, it may be necessary for the application to reset the FIFOs to get rid of any unwanted data.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

```
unsigned int XUartLite_Send( XUartLite * InstancePtr,  
                             Xuint8 *   DataBufferPtr,  
                             unsigned int NumBytes  
                             )
```

This functions sends the specified buffer of data using the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART. If the UART is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send as much data as the UART can buffer in the FIFO. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.

NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

unsigned int XUartLite_SendBuffer(XUartLite * InstancePtr)

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartLite** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent (put into the UART transmitter and/or FIFO).

Note:

None.

uartlite/v1_00_b/src/xuartlite_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xuartlite_i.h,v 1.7 2002/05/02 20:34:12 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file uartlite/v1_00_b/src/xuartlite_i.h
00026 *
00027 * Contains data which is shared between the files of the XUartLite component.
00028 * It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date      Changes
00034 * -----
00035 * 1.00a  ecm    08/31/01  First release
00036 * 1.00b  jhl    02/21/02  Reparitioned the driver for smaller files
00037 * 1.00b  rpm    04/24/02  Moved register definitions to xuartlite_l.h and
00038 *                          updated macro naming convention
00039 * </pre>
00040 *
00041 *****/
00042
```

```

00043 #ifndef XUARTLITE_I_H /* prevent circular inclusions */
00044 #define XUARTLITE_I_H /* by using protection macros */
00045
00046 /***** Include Files *****/
00047
00048 #include "xuartlite.h"
00049 #include "xuartlite_l.h"
00050
00051 /***** Constant Definitions *****/
00052
00053 /***** Type Definitions *****/
00054
00055 /***** Macros (Inline Functions) Definitions *****/
00056
00057 /*****
00058 *
00059 * Update the statistics of the instance.
00060 *
00061 * @param InstancePtr is a pointer to the XUartLite instance to be worked on.
00062 * @param StatusRegister contains the contents of the UART status register
00063 * to update the statistics with.
00064 *
00065 * @return
00066 *
00067 * None.
00068 *
00069 * @note
00070 *
00071 * Signature: void XUartLite_mUpdateStats(XUartLite *InstancePtr,
00072 *                                         Xuint32 StatusRegister)
00073 *
00074 *****/
00075 #define XUartLite_mUpdateStats(InstancePtr, StatusRegister) \
00076 { \
00077     if ((StatusRegister) & XUL_SR_OVERRUN_ERROR) \
00078     { \
00079         (InstancePtr)->Stats.ReceiveOverrunErrors++; \
00080     } \
00081     if ((StatusRegister) & XUL_SR_PARITY_ERROR) \
00082     { \
00083         (InstancePtr)->Stats.ReceiveParityErrors++; \
00084     } \
00085     if ((StatusRegister) & XUL_SR_FRAMING_ERROR) \
00086     { \
00087         (InstancePtr)->Stats.ReceiveFramingErrors++; \
00088     } \
00089 }
00090
00091 /*****
00092 *
00093 * Clear the statistics for the instance.
00094 *

```

```

00095 * @param InstancePtr is a pointer to the XUartLite instance to be worked on.
00096 *
00097 * @return
00098 *
00099 * None.
00100 *
00101 * @note
00102 *
00103 * Signature: void XUartLite_mClearStats(XUartLite *InstancePtr)
00104 *
00105 *****/
00106 #define XUartLite_mClearStats(InstancePtr) \
00107 { \
00108     (InstancePtr)->Stats.TransmitInterrupts = 0UL; \
00109     (InstancePtr)->Stats.ReceiveInterrupts = 0UL; \
00110     (InstancePtr)->Stats.CharactersTransmitted = 0UL; \
00111     (InstancePtr)->Stats.CharactersReceived = 0UL; \
00112     (InstancePtr)->Stats.ReceiveOverrunErrors = 0UL; \
00113     (InstancePtr)->Stats.ReceiveFramingErrors = 0UL; \
00114     (InstancePtr)->Stats.ReceiveParityErrors = 0UL; \
00115 }
00116
00117 *****/
00118
00119 /* the configuration table */
00120 extern XUartLite_Config XUartLite_ConfigTable[];
00121
00122 *****/
00123
00124 unsigned int XUartLite_SendBuffer(XUartLite *InstancePtr);
00125 unsigned int XUartLite_ReceiveBuffer(XUartLite *InstancePtr);
00126
00127
00128 #endif /* end of protection macro */

```


uartlite/v1_00_b/src/xuartlite_i.h File Reference

Detailed Description

Contains data which is shared between the files of the **XUartLite** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files
1.00b	rpm	04/24/02	Moved register definitions to xuartlite_1.h and updated macro naming convention

```
#include "xuartlite.h"  
#include "xuartlite\_1.h"
```

[Go to the source code of this file.](#)

Functions

```
unsigned int XUartLite\_SendBuffer (XUartLite *InstancePtr)  
unsigned int XUartLite\_ReceiveBuffer (XUartLite *InstancePtr)
```

Variables

[XUartLite_Config XUartLite_ConfigTable](#) []

Function Documentation

unsigned int XUartLite_ReceiveBuffer(XUartLite * InstancePtr)

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartLite** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

unsigned int XUartLite_SendBuffer(XUartLite * InstancePtr)

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartLite** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent (put into the UART transmitter and/or FIFO).

Note:

None.

Variable Documentation

XUartLite_Config XUartLite_ConfigTable[]()

The configuration table for UART Lite devices

uartlite/v1_00_b/src/xuartlite_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xuartlite.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00b	rpm	04/25/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XUartLite_mSetControlReg(BaseAddress, Mask)  
#define XUartLite_mGetControlReg(BaseAddress)  
#define XUartLite_mGetStatusReg(BaseAddress)  
#define XUartLite_mIsReceiveEmpty(BaseAddress)  
#define XUartLite_mIsTransmitFull(BaseAddress)  
#define XUartLite_mIsIntrEnabled(BaseAddress)
```

```
#define XUartLite_mEnableIntr(BaseAddress)
#define XUartLite_mDisableIntr(BaseAddress)
```

Functions

```
void XUartLite_SendByte (Xuint32 BaseAddress, Xuint8 Data)
Xuint8 XUartLite_RecvByte (Xuint32 BaseAddress)
```

Define Documentation

```
#define XUartLite_mDisableIntr( BaseAddress )
```

Disable the device interrupt. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XUartLite_mEnableIntr( BaseAddress )
```

Enable the device interrupt. Preserve the contents of the control register.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

#define XUartLite_mGetControlReg(BaseAddress)

Get the contents of the control register. Use the XUL_CR_* constants defined above to interpret the bit-mask returned.

Parameters:

BaseAddress is the base address of the device

Returns:

A 32-bit value representing the contents of the control register.

Note:

None.

#define XUartLite_mGetStatusReg(BaseAddress)

Get the contents of the status register. Use the XUL_SR_* constants defined above to interpret the bit-mask returned.

Parameters:

BaseAddress is the base address of the device

Returns:

A 32-bit value representing the contents of the status register.

Note:

None.

#define XUartLite_mIsIntrEnabled(BaseAddress)

Check to see if the interrupt is enabled.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the interrupt is enabled, XFALSE otherwise.

Note:

None.

```
#define XUartLite_mIsReceiveEmpty( BaseAddress )
```

Check to see if the receiver has data.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the receiver is empty, XFALSE if there is data present.

Note:

None.

```
#define XUartLite_mIsTransmitFull( BaseAddress )
```

Check to see if the transmitter is full.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the transmitter is full, XFALSE otherwise.

Note:

None.

```
#define XUartLite_mSetControlReg( BaseAddress,  
                                Mask      )
```

Set the contents of the control register. Use the XUL_CR_* constants defined above to create the bit-mask to be written to the register.

Parameters:

BaseAddress is the base address of the device

Mask is the 32-bit value to write to the control register

Returns:

None.

Note:

None.

Function Documentation

```
Xuint8 XUartLite_RecvByte( Xuint32 BaseAddress)
```

This functions receives a single byte using the UART. It is blocking in that it waits for the receiver to become non-empty before it reads from the receive register.

Parameters:

BaseAddress is the base address of the device

Returns:

The byte of data received.

Note:

None.

```
void XUartLite_SendByte( Xuint32 BaseAddress,  
                        Xuint8  Data  
                        )
```


This function sends a single byte using the UART. It is blocking in that it waits for the transmitter to become non-full before it writes the byte to the transmit register.

Parameters:

BaseAddress is the base address of the device

Data is the byte of data to send

Returns:

None.

Note:

None.

uartlite/v1_00_b/src/xuartlite_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xuartlite_l.h,v 1.2 2002/07/24 22:35:28 robertm Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file uartlite/v1_00_b/src/xuartlite_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  High-level driver functions
00029 * are defined in xuartlite.h.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date        Changes
00035 * -----
00036 * 1.00b rpm  04/25/02  First release
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XUARTLITE_L_H /* prevent circular inclusions */
00042 #define XUARTLITE_L_H /* by using protection macros */
```

```

00043
00044 /***** Include Files *****/
00045
00046 #include "xbasic_types.h"
00047 #include "xio.h"
00048
00049 /***** Constant Definitions *****/
00050
00051 /* UART Lite register offsets */
00052
00053 #define XUL_RX_FIFO_OFFSET      0 /* receive FIFO, read only */
00054 #define XUL_TX_FIFO_OFFSET      4 /* transmit FIFO, write only */
00055 #define XUL_STATUS_REG_OFFSET   8 /* status register, read only */
00056 #define XUL_CONTROL_REG_OFFSET 12 /* control register, write only */
00057
00058 /* control register bit positions */
00059
00060 #define XUL_CR_ENABLE_INTR      0x10 /* enable interrupt */
00061 #define XUL_CR_FIFO_RX_RESET    0x02 /* reset receive FIFO */
00062 #define XUL_CR_FIFO_TX_RESET    0x01 /* reset transmit FIFO */
00063
00064 /* status register bit positions */
00065
00066 #define XUL_SR_PARITY_ERROR      0x80
00067 #define XUL_SR_FRAMING_ERROR     0x40
00068 #define XUL_SR_OVERRUN_ERROR    0x20
00069 #define XUL_SR_INTR_ENABLED     0x10 /* interrupt enabled */
00070 #define XUL_SR_TX_FIFO_FULL     0x08 /* transmit FIFO full */
00071 #define XUL_SR_TX_FIFO_EMPTY    0x04 /* transmit FIFO empty */
00072 #define XUL_SR_RX_FIFO_FULL     0x02 /* receive FIFO full */
00073 #define XUL_SR_RX_FIFO_VALID_DATA 0x01 /* data in receive FIFO */
00074
00075 /* the following constant specifies the size of the FIFOs, the size of the
00076 * FIFOs includes the transmitter and receiver such that it is the total number
00077 * of bytes that the UART can buffer
00078 */
00079 #define XUL_FIFO_SIZE            16
00080
00081 /* Stop bits are fixed at 1. Baud, parity, and data bits are fixed on a
00082 * per instance basis
00083 */
00084 #define XUL_STOP_BITS            1
00085
00086 /* Parity definitions
00087 */
00088 #define XUL_PARITY_NONE          0
00089 #define XUL_PARITY_ODD           1
00090 #define XUL_PARITY_EVEN         2
00091
00092 /***** Type Definitions *****/
00093
00094 /***** Macros (Inline Functions) Definitions *****/

```

```

00095
00096 /*****
00097 *
00098 * Low-level driver macros and functions. The list below provides signatures
00099 * to help the user use the macros.
00100 *
00101 * void XUartLite_mSetControlReg(Xuint32 BaseAddress, Xuint32 Mask)
00102 * Xuint32 XUartLite_mGetControlReg(Xuint32 BaseAddress)
00103 * Xuint32 XUartLite_mGetStatusReg(Xuint32 BaseAddress)
00104 *
00105 * Xboolean XUartLite_mIsReceiveEmpty(Xuint32 BaseAddress)
00106 * Xboolean XUartLite_mIsTransmitFull(Xuint32 BaseAddress)
00107 * Xboolean XUartLite_mIsIntrEnabled(Xuint32 BaseAddress)
00108 *
00109 * void XUartLite_mEnableIntr(Xuint32 BaseAddress)
00110 * void XUartLite_mDisableIntr(Xuint32 BaseAddress)
00111 *
00112 * void XUartLite_SendByte(Xuint32 BaseAddress, Xuint8 Data);
00113 * Xuint8 XUartLite_RecvByte(Xuint32 BaseAddress);
00114 *
00115 *****/
00116
00117 /*****/
00118 /**
00119 *
00120 * Set the contents of the control register. Use the XUL_CR_* constants defined
00121 * above to create the bit-mask to be written to the register.
00122 *
00123 * @param BaseAddress is the base address of the device
00124 * @param Mask is the 32-bit value to write to the control register
00125 *
00126 * @return None.
00127 *
00128 * @note None.
00129 *
00130 *****/
00131 #define XUartLite_mSetControlReg(BaseAddress, Mask) \
00132         XIo_Out32((BaseAddress) + XUL_CONTROL_REG_OFFSET, (Mask))
00133
00134
00135 /*****/
00136 /**
00137 *
00138 * Get the contents of the control register. Use the XUL_CR_* constants defined
00139 * above to interpret the bit-mask returned.
00140 *
00141 * @param BaseAddress is the base address of the device
00142 *
00143 * @return A 32-bit value representing the contents of the control register.
00144 *
00145 * @note None.
00146 *

```

```
00147 *****/
00148 #define XUartLite_mGetControlReg(BaseAddress) \
00149         XIo_In32((BaseAddress) + XUL_CONTROL_REG_OFFSET)
00150
00151
00152 /***/
00153 /**
00154 *
00155 * Get the contents of the status register. Use the XUL_SR_* constants defined
00156 * above to interpret the bit-mask returned.
00157 *
00158 * @param   BaseAddress is the base address of the device
00159 *
00160 * @return  A 32-bit value representing the contents of the status register.
00161 *
00162 * @note    None.
00163 *
00164 *****/
00165 #define XUartLite_mGetStatusReg(BaseAddress) \
00166         XIo_In32((BaseAddress) + XUL_STATUS_REG_OFFSET)
00167
00168
00169 /***/
00170 /**
00171 *
00172 * Check to see if the receiver has data.
00173 *
00174 * @param   BaseAddress is the base address of the device
00175 *
00176 * @return  XTRUE if the receiver is empty, XFALSE if there is data present.
00177 *
00178 * @note    None.
00179 *
00180 *****/
00181 #define XUartLite_mIsReceiveEmpty(BaseAddress) \
00182         (!(XUartLite_mGetStatusReg((BaseAddress)) &
XUL_SR_RX_FIFO_VALID_DATA))
00183
00184
00185 /***/
00186 /**
00187 *
00188 * Check to see if the transmitter is full.
00189 *
00190 * @param   BaseAddress is the base address of the device
00191 *
00192 * @return  XTRUE if the transmitter is full, XFALSE otherwise.
00193 *
00194 * @note    None.
00195 *
00196 *****/
00197 #define XUartLite_mIsTransmitFull(BaseAddress) \
```

```

00198             (XUartLite_mGetStatusReg((BaseAddress)) & XUL_SR_TX_FIFO_FULL)
00199
00200
00201 /*****
00202 /**
00203 *
00204 * Check to see if the interrupt is enabled.
00205 *
00206 * @param   BaseAddress is the base address of the device
00207 *
00208 * @return  XTRUE if the interrupt is enabled, XFALSE otherwise.
00209 *
00210 * @note    None.
00211 *
00212 *****/
00213 #define XUartLite_mIsIntrEnabled(BaseAddress) \
00214             (XUartLite_mGetStatusReg((BaseAddress)) & XUL_SR_INTR_ENABLED)
00215
00216
00217 /*****
00218 /**
00219 *
00220 * Enable the device interrupt. Preserve the contents of the control register.
00221 *
00222 * @param   BaseAddress is the base address of the device
00223 *
00224 * @return  None.
00225 *
00226 * @note    None.
00227 *
00228 *****/
00229 #define XUartLite_mEnableIntr(BaseAddress) \
00230             XUartLite_mSetControlReg((BaseAddress), \
00231             XUartLite_mGetControlReg((BaseAddress)) |
XUL_CR_ENABLE_INTR)
00232
00233
00234 /*****
00235 /**
00236 *
00237 * Disable the device interrupt. Preserve the contents of the control register.
00238 *
00239 * @param   BaseAddress is the base address of the device
00240 *
00241 * @return  None.
00242 *
00243 * @note    None.
00244 *
00245 *****/
00246 #define XUartLite_mDisableIntr(BaseAddress) \
00247             XUartLite_mSetControlReg((BaseAddress), \
00248             XUartLite_mGetControlReg((BaseAddress)) &

```

```
~XUL_CR_ENABLE_INTR)
00249
00250
00251 /***** Function Prototypes *****/
00252
00253 void XUartLite_SendByte(Xuint32 BaseAddress, Xuint8 Data);
00254 Xuint8 XUartLite_RecvByte(Xuint32 BaseAddress);
00255
00256
00257 #endif                /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

uartlite/v1_00_b/src/xuartlite_l.c File Reference

Detailed Description

This file contains low-level driver functions that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	rpm	04/25/02	First release

```
#include "xuartlite_l.h"
```

Functions

```
void XUartLite_SendByte (Xuint32 BaseAddress, Xuint8 Data)  
Xuint8 XUartLite_RecvByte (Xuint32 BaseAddress)
```

Function Documentation

```
Xuint8 XUartLite_RecvByte( Xuint32 BaseAddress)
```


This functions receives a single byte using the UART. It is blocking in that it waits for the receiver to become non-empty before it reads from the receive register.

Parameters:

BaseAddress is the base address of the device

Returns:

The byte of data received.

Note:

None.

```
void XUartLite_SendByte( Xuint32 BaseAddress,  
                        Xuint8 Data  
                        )
```

This functions sends a single byte using the UART. It is blocking in that it waits for the transmitter to become non-full before it writes the byte to the transmit register.

Parameters:

BaseAddress is the base address of the device

Data is the byte of data to send

Returns:

None.

Note:

None.

uartlite/v1_00_b/src/xuartlite_selftest.c

File Reference

Detailed Description

This file contains the self-test functions for the UART Lite component ([XUartLite](#)).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xuartlite.h"  
#include "xuartlite_i.h"  
#include "xio.h"
```

Functions

[XStatus XUartLite_SelfTest](#) ([XUartLite](#) *InstancePtr)

Function Documentation

XStatus XUartLite_SelfTest(XUartLite * InstancePtr)

Runs a self-test on the device hardware. Since there is no way to perform a loopback in the hardware, this test can only check the state of the status register to verify it is correct. This test assumes that the hardware device is still in its reset state, but has been initialized with the Initialize function.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

- XST_SUCCESS if the self-test was successful.
- XST_FAILURE if the self-test failed, the status register value was not correct

Note:

None.

uartlite/v1_00_b/src/xuartlite_intr.c File Reference

Detailed Description

This file contains interrupt-related functions for the UART Lite component ([XUartLite](#)).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xbasic_types.h"  
#include "xuartlite.h"  
#include "xuartlite_i.h"  
#include "xio.h"
```

Functions

```
void XUartLite_SetRecvHandler (XUartLite *InstancePtr, XUartLite_Handler FuncPtr, void  
*CallbackRef)  
void XUartLite_SetSendHandler (XUartLite *InstancePtr, XUartLite_Handler FuncPtr, void  
*CallbackRef)  
void XUartLite_InterruptHandler (XUartLite *InstancePtr)  
void XUartLite_DisableInterrupt (XUartLite *InstancePtr)
```

void **XUartLite_EnableInterrupt** (**XUartLite** *InstancePtr)

Function Documentation

void **XUartLite_DisableInterrupt**(**XUartLite** * *InstancePtr*)

This function disables the UART interrupt. After calling this function, data may still be received by the UART but no interrupt will be generated since the hardware device has no way to disable the receiver.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

void **XUartLite_EnableInterrupt**(**XUartLite** * *InstancePtr*)

This function enables the UART interrupt such that an interrupt will occur when data is received or data has been transmitted. The device contains 16 byte receive and transmit FIFOs such that an interrupt is generated anytime there is data in the receive FIFO and when the transmit FIFO transitions from not empty to empty.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

Returns:

None.

Note:

None.

void **XUartLite_InterruptHandler**(**XUartLite** * *InstancePtr*)

This function is the interrupt handler for the UART lite driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any UART lite occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the UART that the interrupt is for.

Returns:

None.

Note:

None.

```
void XUartLite_SetRecvHandler( XUartLite *      InstancePtr,  
                               XUartLite_Handler FuncPtr,  
                               void *          CallbackRef  
                               )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

```
void XUartLite_SetSendHandler( XUartLite *      InstancePtr,  
                               XUartLite_Handler FuncPtr,  
                               void *          CallbackRef  
                               )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartLite** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

uartlite/v1_00_b/src/xuartlite_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of UART Lite devices in the system. Each device in the system should have an entry in the table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/31/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files

```
#include "xuartlite.h"  
#include "xparameters.h"
```

Variables

[XUartLite_Config XUartLite_ConfigTable](#) [XPAR_XUARTLITE_NUM_INSTANCES]

Variable Documentation

[XUartLite_Config XUartLite_ConfigTable](#)[XPAR_XUARTLITE_NUM_INSTANCES]

The configuration table for UART Lite devices

Generated on 29 May 2003 for Xilinx Device Drivers

uartns550/v1_00_b/src/xuartns550_stats.c

File Reference

Detailed Description

This file contains the statistics functions for the 16450/16550 UART driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xuartns550.h"  
#include "xuartns550_i.h"
```

Functions

```
void XUartNs550_GetStats (XUartNs550 *InstancePtr, XUartNs550Stats *StatsPtr)  
void XUartNs550_ClearStats (XUartNs550 *InstancePtr)
```

Function Documentation

```
void XUartNs550_ClearStats( XUartNs550 * InstancePtr)
```

This function zeros the statistics for the given instance.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

None.

Note:

None.

```
void XUartNs550_GetStats( XUartNs550 * InstancePtr,  
                        XUartNs550Stats * StatsPtr  
                        )
```

This functions returns a snapshot of the current statistics in the area provided.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

StatsPtr is a pointer to a **XUartNs550Stats** structure to where the statistics are to be copied to.

Returns:

None.

Note:

None.

uartns550/v1_00_b/src/xuartns550.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xuartns550.h,v 1.4 2002/05/02 20:56:21 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file uartns550/v1_00_b/src/xuartns550.h
00026 *
00027 * This driver supports the following features in the Xilinx 16450/16550
00028 * compatible UART.
00029 *
00030 * - Dynamic data format (baud rate, data bits, stop bits, parity)
00031 * - Polled mode
00032 * - Interrupt driven mode
00033 * - Transmit and receive FIFOs (16 bytes each for the 16550)
00034 * - Access to the external modem control lines and the two discrete outputs
00035 *
00036 * The only difference between the 16450 and the 16550 is the addition of
00037 * transmit and receive FIFOs in the 16550.
00038 *
00039 * <b>Baud Rate</b>
00040 *
00041 * The UART has an internal baud rate generator that is clocked at a specified
00042 * input clock frequency. Not all baud rates can be generated from some clock
```

00043 * frequencies. The requested baud rate is checked using the provided clock for
00044 * the system, and checked against the acceptable error range. An error may be
00045 * returned from some functions indicating the baud rate was in error because
00046 * it could not be generated.
00047 *
00048 * **Interrupts**
00049 *
00050 * The device does not have any way to disable the receiver such that the
00051 * receive FIFO may contain unwanted data. The FIFOs are not flushed when the
00052 * driver is initialized, but a function is provided to allow the user to reset
00053 * the FIFOs if desired.
00054 *
00055 * The driver defaults to no interrupts at initialization such that interrupts
00056 * must be enabled if desired. An interrupt is generated for any of the
following
00057 * conditions.
00058 *
00059 * - Transmit FIFO is empty
00060 * - Data in the receive FIFO equal to the receive threshold
00061 * - Data in the receiver when FIFOs are disabled
00062 * - Any receive status error or break condition detected
00063 * - Data in the receive FIFO for 4 character times without receiver activity
00064 * - A change of a modem signal
00065 *
00066 * The application can control which interrupts are enabled using the SetOptions
00067 * function.
00068 *
00069 * In order to use interrupts, it is necessary for the user to connect the
driver
00070 * interrupt handler, XUartNs550_InterruptHandler(), to the interrupt system of
00071 * the application. This function does not save and restore the processor
context
00072 * such that the user must provide it. A handler must be set for the driver such
00073 * that the handler is called when interrupt events occur. The handler is called
00074 * from interrupt context and is designed to allow application specific
processing
00075 * to be performed.
00076 *
00077 * The functions, XUartNs550_Send() and XUartNs550_Recv(), are provided in the
00078 * driver to allow data to be sent and received. They are designed to be used in
00079 * polled or interrupt modes.
00080 *
00081 * @note
00082 *
00083 * The default configuration for the UART after initialization is:
00084 *
00085 * - 19,200 bps or XPAR_DEFAULT_BAUD_RATE if defined
00086 * - 8 data bits
00087 * - 1 stop bit
00088 * - no parity
00089 * - FIFO's are enabled with a receive threshold of 8 bytes
00090 *
00091 *

```

00092 * MODIFICATION HISTORY:
00093 *
00094 * Ver    Who    Date      Changes
00095 * -----
00096 * 1.00a ecm   08/16/01 First release
00097 * 1.00b jhl   03/11/02 Repartitioned the driver for smaller files.
00098 * </pre>
00099 *
00100 *****/
00101
00102 #ifndef XUARTNS550_H /* prevent circular inclusions */
00103 #define XUARTNS550_H /* by using protection macros */
00104
00105 /***** Include Files *****/
00106
00107 #include "xbasic_types.h"
00108 #include "xstatus.h"
00109 #include "xuartns550_1.h"
00110
00111 /***** Constant Definitions *****/
00112
00113 /* The following constants indicate the max and min baud rates and these
00114  * numbers are based only on the testing that has been done. The hardware
00115  * is capable of other baud rates.
00116  */
00117 #define XUN_NS16550_MAX_RATE      115200
00118 #define XUN_NS16550_MIN_RATE      300
00119
00120 /** @name Configuration options
00121  * @{
00122  */
00123 /**
00124  * These constants specify the options that may be set or retrieved
00125  * with the driver, each is a unique bit mask such that multiple options
00126  * may be specified. These constants indicate the function of the option
00127  * when in the active state.
00128  *
00129  * <pre>
00130  * XUN_OPTION_SET_BREAK          Set a break condition
00131  * XUN_OPTION_LOOPBACK          Enable local loopback
00132  * XUN_OPTION_DATA_INTR         Enable data interrupts
00133  * XUN_OPTION_MODEM_INTR        Enable modem interrupts
00134  * XUN_OPTION_FIFOS_ENABLE      Enable FIFOs
00135  * XUN_OPTION_RESET_TX_FIFO     Reset the transmit FIFO
00136  * XUN_OPTION_RESET_RX_FIFO     Reset the receive FIFO
00137  * XUN_OPTION_ASSERT_OUT2       Assert out2 signal
00138  * XUN_OPTION_ASSERT_OUT1       Assert out1 signal
00139  * XUN_OPTION_ASSERT_RTS        Assert RTS signal
00140  * XUN_OPTION_ASSERT_DTR        Assert DTR signal
00141  * </pre>
00142  */
00143 #define XUN_OPTION_SET_BREAK      0x0400

```

```

00144 #define XUN_OPTION_LOOPBACK          0x0200
00145 #define XUN_OPTION_DATA_INTR         0x0100
00146 #define XUN_OPTION_MODEM_INTR        0x0080
00147 #define XUN_OPTION_FIFOS_ENABLE      0x0040
00148 #define XUN_OPTION_RESET_TX_FIFO     0x0020
00149 #define XUN_OPTION_RESET_RX_FIFO     0x0010
00150 #define XUN_OPTION_ASSERT_OUT2       0x0008
00151 #define XUN_OPTION_ASSERT_OUT1       0x0004
00152 #define XUN_OPTION_ASSERT_RTS        0x0002
00153 #define XUN_OPTION_ASSERT_DTR        0x0001
00154 /*@}*/
00155
00156 /** @name Data format values
00157  * @{
00158  */
00159 /**
00160  * These constants specify the data format that may be set or retrieved
00161  * with the driver.  The data format includes the number of data bits, the
00162  * number of stop bits and parity.
00163  *
00164  * <pre>
00165  * XUN_FORMAT_8_BITS          8 data bits
00166  * XUN_FORMAT_7_BITS          7 data bits
00167  * XUN_FORMAT_6_BITS          6 data bits
00168  * XUN_FORMAT_5_BITS          5 data bits
00169  * XUN_FORMAT_EVEN_PARITY     Even parity
00170  * XUN_FORMAT_ODD_PARITY      Odd parity
00171  * XUN_FORMAT_NO_PARITY       No parity
00172  * XUN_FORMAT_2_STOP_BIT      2 stop bits
00173  * XUN_FORMAT_1_STOP_BIT      1 stop bit
00174  * </pre>
00175  */
00176 #define XUN_FORMAT_8_BITS              3
00177 #define XUN_FORMAT_7_BITS              2
00178 #define XUN_FORMAT_6_BITS              1
00179 #define XUN_FORMAT_5_BITS              0
00180
00181 #define XUN_FORMAT_EVEN_PARITY         2
00182 #define XUN_FORMAT_ODD_PARITY         1
00183 #define XUN_FORMAT_NO_PARITY          0
00184
00185 #define XUN_FORMAT_2_STOP_BIT          1
00186 #define XUN_FORMAT_1_STOP_BIT          0
00187 /*@}*/
00188
00189 /** @name FIFO trigger values
00190  * @{
00191  */
00192 /*
00193  * These constants specify receive FIFO trigger levels which specify

```

```

00194 * the number of bytes at which a receive data event (interrupt) will occur.
00195 *
00196 * <pre>
00197 * XUN_FIFO_TRIGGER_14          14 byte trigger level
00198 * XUN_FIFO_TRIGGER_08          8 byte trigger level
00199 * XUN_FIFO_TRIGGER_04          4 byte trigger level
00200 * XUN_FIFO_TRIGGER_01          1 byte trigger level
00201 * </pre>
00202 */
00203 #define XUN_FIFO_TRIGGER_14      0xC0
00204 #define XUN_FIFO_TRIGGER_08      0x80
00205 #define XUN_FIFO_TRIGGER_04      0x40
00206 #define XUN_FIFO_TRIGGER_01      0x00
00207 /*@}*/
00208
00209 /** @name Modem status values
00210 * @{
00211 */
00212 /**
00213 * These constants specify the modem status that may be retrieved
00214 * from the driver.
00215 *
00216 * <pre>
00217 * XUN_MODEM_DCD_DELTA_MASK      DCD signal changed state
00218 * XUN_MODEM_DSR_DELTA_MASK      DSR signal changed state
00219 * XUN_MODEM_CTS_DELTA_MASK      CTS signal changed state
00220 * XUN_MODEM_RINGING_MASK        Ring signal is active
00221 * XUN_MODEM_DSR_MASK            Current state of DSR signal
00222 * XUN_MODEM_CTS_MASK            Current state of CTS signal
00223 * XUN_MODEM_DCD_MASK            Current state of DCD signal
00224 * XUN_MODEM_RING_STOP_MASK      Ringing has stopped
00225 * </pre>
00226 */
00227 #define XUN_MODEM_DCD_DELTA_MASK  0x80
00228 #define XUN_MODEM_DSR_DELTA_MASK  0x02
00229 #define XUN_MODEM_CTS_DELTA_MASK  0x01
00230 #define XUN_MODEM_RINGING_MASK    0x40
00231 #define XUN_MODEM_DSR_MASK        0x20
00232 #define XUN_MODEM_CTS_MASK        0x10
00233 #define XUN_MODEM_DCD_MASK        0x08
00234 #define XUN_MODEM_RING_STOP_MASK  0x04
00235 /*@}*/
00236
00237 /** @name Callback events
00238 * @{
00239 */
00240 /**
00241 * These constants specify the handler events that are passed to
00242 * a handler from the driver. These constants are not bit masks such that
00243 * only one will be passed at a time to the handler.
00244 *
00245 * <pre>

```



```

00246 * XUN_EVENT_RECV_DATA          Data has been received
00247 * XUN_EVENT_RECV_TIMEOUT      A receive timeout occurred
00248 * XUN_EVENT_SENT_DATA          Data has been sent
00249 * XUN_EVENT_RECV_ERROR         A receive error was detected
00250 * XUN_EVENT_MODEM              A change in modem status
00251 * </pre>
00252 */
00253 #define XUN_EVENT_RECV_DATA      1
00254 #define XUN_EVENT_RECV_TIMEOUT  2
00255 #define XUN_EVENT_SENT_DATA      3
00256 #define XUN_EVENT_RECV_ERROR     4
00257 #define XUN_EVENT_MODEM          5
00258 /*@}*/
00259
00260 /** @name Error values
00261 * @{
00262 */
00263 /**
00264 * These constants specify the errors that may be retrieved from
00265 * the driver using the XUartNs550_GetLastErrors function. All of them are
00266 * bit masks, except no error, such that multiple errors may be specified.
00267 *
00268 * <pre>
00269 * XUN_ERROR_BREAK_MASK          Break detected
00270 * XUN_ERROR_FRAMING_MASK        Receive framing error
00271 * XUN_ERROR_PARITY_MASK         Receive parity error
00272 * XUN_ERROR_OVERRUN_MASK        Receive overrun error
00273 * XUN_ERROR_NONE                No error
00274 * </pre>
00275 */
00276 #define XUN_ERROR_BREAK_MASK      0x10
00277 #define XUN_ERROR_FRAMING_MASK    0x08
00278 #define XUN_ERROR_PARITY_MASK     0x04
00279 #define XUN_ERROR_OVERRUN_MASK    0x02
00280 #define XUN_ERROR_NONE            0x00
00281 /*@}*/
00282
00283 /***** Type Definitions *****/
00284
00285 /**
00286 * This typedef contains configuration information for the device.
00287 */
00288 typedef struct
00289 {
00290     Xuint16 DeviceId;          /**< Unique ID of device */
00291     Xuint32 BaseAddress;       /**< Base address of device (IPIF) */
00292     Xuint32 InputClockHz;     /**< Input clock frequency */
00293 } XUartNs550_Config;
00294
00295 /**
00296 * The following data type is used to manage the buffers that are handled

```

```

00297 * when sending and receiving data in the interrupt mode.
00298 */
00299 typedef struct
00300 {
00301     Xuint8 *NextBytePtr;
00302     unsigned int RequestedBytes;
00303     unsigned int RemainingBytes;
00304 } XUartNs550Buffer;
00305
00306 /**
00307 * This data type allows the data format of the device to be set
00308 * and retrieved.
00309 */
00310 typedef struct
00311 {
00312     Xuint32 BaudRate;           /**< In bps, ie 1200 */
00313     Xuint32 DataBits;          /**< Number of data bits */
00314     Xuint32 Parity;            /**< Parity */
00315     Xuint8 StopBits;           /**< Number of stop bits */
00316 } XUartNs550Format;
00317
00318 /**
00319 * This data type defines a handler which the application must define
00320 * when using interrupt mode. The handler will be called from the driver in an
00321 * interrupt context to handle application specific processing.
00322 *
00323 * @param CallbackRef is a callback reference passed in by the upper layer
00324 * when setting the handler, and is passed back to the upper layer when
00325 * the handler is called.
00326 * @param Event contains one of the event constants indicating why the handler
00327 * is being called.
00328 * @param EventData contains the number of bytes sent or received at the time
00329 * of
00330 * the call for send and receive events and contains the modem status
00331 * for
00332 * modem events.
00333 */
00334 typedef void (*XUartNs550_Handler)(void *CallbackRef, Xuint32 Event,
00335                                     unsigned int EventData);
00336
00337 /**
00338 * UART statistics
00339 */
00340 typedef struct
00341 {
00342     Xuint16 TransmitInterrupts;    /**< Number of transmit interrupts */
00343     Xuint16 ReceiveInterrupts;     /**< Number of receive interrupts */
00344     Xuint16 StatusInterrupts;      /**< Number of status interrupts */
00345     Xuint16 ModemInterrupts;       /**< Number of modem interrupts */
00346     Xuint16 CharactersTransmitted;  /**< Number of characters transmitted
00347 */

```

```

00345     Xuint16 CharactersReceived;           /**< Number of characters received */
00346     Xuint16 ReceiveOverrunErrors;         /**< Number of receive overruns */
00347     Xuint16 ReceiveParityErrors;         /**< Number of receive parity errors */
00348     Xuint16 ReceiveFramingErrors;        /**< Number of receive framing errors
*/
00349     Xuint16 ReceiveBreakDetected;        /**< Number of receive breaks */
00350 } XUartNs550Stats;
00351
00352 /**
00353  * The XUartNs550 driver instance data. The user is required to allocate a
00354  * variable of this type for every UART 16550/16450 device in the system.
00355  * A pointer to a variable of this type is then passed to the driver API
00356  * functions.
00357  */
00358 typedef struct
00359 {
00360     XUartNs550Stats Stats;                /* Component Statistics */
00361     Xuint32 BaseAddress;                  /* Base address of device (IPIF) */
00362     Xuint32 InputClockHz;                 /* Input clock frequency */
00363     Xuint32 IsReady;                      /* Device is initialized and ready */
00364     Xuint32 BaudRate;                     /* current baud rate of hw */
00365     Xuint8  LastErrors;                   /* the accumulated errors */
00366
00367     XUartNs550Buffer SendBuffer;
00368     XUartNs550Buffer ReceiveBuffer;
00369
00370     XUartNs550_Handler Handler;
00371     void *CallBackRef;                    /* Callback reference for control handler */
00372 } XUartNs550;
00373
00374
00375 /***** Macros (Inline Functions) Definitions *****/
00376
00377
00378 /***** Function Prototypes *****/
00379 /*
00380  * required functions in xuartns550.c
00381  */
00382 XStatus XUartNs550_Initialize(XUartNs550 *InstancePtr, Xuint16 DeviceId);
00383
00384 unsigned int XUartNs550_Send(XUartNs550 *InstancePtr, Xuint8 *BufferPtr,
00385                             unsigned int NumBytes);
00386 unsigned int XUartNs550_Recv(XUartNs550 *InstancePtr, Xuint8 *BufferPtr,
00387                             unsigned int NumBytes);
00388 XUartNs550_Config *XUartNs550_LookupConfig(Xuint16 DeviceId);
00389
00390 /*
00391  * options functions in xuartns550_options.c
00392  */
00393 XStatus XUartNs550_SetOptions(XUartNs550 *InstancePtr, Xuint16 Options);

```

```

00394 Xuint16 XUartNs550_GetOptions(XUartNs550 *InstancePtr);
00395
00396 XStatus XUartNs550_SetFifoThreshold(XUartNs550 *InstancePtr,
00397                                     Xuint8 TriggerLevel);
00398 Xuint8 XUartNs550_GetFifoThreshold(XUartNs550 *InstancePtr);
00399
00400 Xboolean XUartNs550_IsSending(XUartNs550 *InstancePtr);
00401
00402 Xuint8 XUartNs550_GetLastErrors(XUartNs550 *InstancePtr);
00403
00404 Xuint8 XUartNs550_GetModemStatus(XUartNs550 *InstancePtr);
00405
00406 /*
00407  * data format functions in xuartns550_format.c
00408  */
00409 XStatus XUartNs550_SetDataFormat(XUartNs550 *InstancePtr,
00410                                   XUartNs550Format *Format);
00411 void XUartNs550_GetDataFormat(XUartNs550 *InstancePtr,
00412                                 XUartNs550Format *Format);
00413 /*
00414  * interrupt functions in xuartns550_intr.c
00415  */
00416 void XUartNs550_SetHandler(XUartNs550 *InstancePtr, XUartNs550_Handler FuncPtr,
00417                             void *CallBackRef);
00418
00419 void XUartNs550_InterruptHandler(XUartNs550 *InstancePtr);
00420
00421 /*
00422  * statistics functions in xuartns550_stats.c
00423  */
00424 void XUartNs550_GetStats(XUartNs550 *InstancePtr, XUartNs550Stats *StatsPtr);
00425 void XUartNs550_ClearStats(XUartNs550 *InstancePtr);
00426
00427 /*
00428  * self-test functions in xuartns550_selftest.c
00429  */
00430 XStatus XUartNs550_SelfTest(XUartNs550 *InstancePtr);
00431
00432 #endif                                     /* end of protection macro */

```

uartns550/v1_00_b/src/xuartns550.h File Reference

Detailed Description

This driver supports the following features in the Xilinx 16450/16550 compatible UART.

- Dynamic data format (baud rate, data bits, stop bits, parity)
- Polled mode
- Interrupt driven mode
- Transmit and receive FIFOs (16 bytes each for the 16550)
- Access to the external modem control lines and the two discrete outputs

The only difference between the 16450 and the 16550 is the addition of transmit and receive FIFOs in the 16550.

Baud Rate

The UART has an internal baud rate generator that is clocked at a specified input clock frequency. Not all baud rates can be generated from some clock frequencies. The requested baud rate is checked using the provided clock for the system, and checked against the acceptable error range. An error may be returned from some functions indicating the baud rate was in error because it could not be generated.

Interrupts

The device does not have any way to disable the receiver such that the receive FIFO may contain unwanted data. The FIFOs are not flushed when the driver is initialized, but a function is provided to allow the user to reset the FIFOs if desired.

The driver defaults to no interrupts at initialization such that interrupts must be enabled if desired. An interrupt is generated for any of the following conditions.

- Transmit FIFO is empty
- Data in the receive FIFO equal to the receive threshold
- Data in the receiver when FIFOs are disabled
- Any receive status error or break condition detected
- Data in the receive FIFO for 4 character times without receiver activity
- A change of a modem signal

The application can control which interrupts are enabled using the SetOptions function.

In order to use interrupts, it is necessary for the user to connect the driver interrupt handler, [XUartNs550_InterruptHandler\(\)](#), to the interrupt system of the application. This function does not save and restore the processor context such that the user must provide it. A handler must be set for the driver such that the handler is called when interrupt events occur. The handler is called from interrupt context and is designed to allow application specific processing to be performed.

The functions, [XUartNs550_Send\(\)](#) and [XUartNs550_Recv\(\)](#), are provided in the driver to allow data to be sent and received. They are designed to be used in polled or interrupt modes.

Note:

The default configuration for the UART after initialization is:

- 19,200 bps or XPAR_DEFAULT_BAUD_RATE if defined
- 8 data bits
- 1 stop bit
- no parity
- FIFO's are enabled with a receive threshold of 8 bytes

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned the driver for smaller files.

```
#include "xbasic_types.h"
#include "xstatus.h"
#include "xuartns550_1.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XUartNs550
struct XUartNs550_Config
struct XUartNs550Buffer
struct XUartNs550Format
struct XUartNs550Stats
```

Configuration options

```
#define XUN_OPTION_SET_BREAK
#define XUN_OPTION_LOOPBACK
#define XUN_OPTION_DATA_INTR
#define XUN_OPTION_MODEM_INTR
#define XUN_OPTION_FIFOS_ENABLE
#define XUN_OPTION_RESET_TX_FIFO
#define XUN_OPTION_RESET_RX_FIFO
#define XUN_OPTION_ASSERT_OUT2
#define XUN_OPTION_ASSERT_OUT1
#define XUN_OPTION_ASSERT_RTS
#define XUN_OPTION_ASSERT_DTR
```

Data format values

```
#define XUN_FORMAT_8_BITS
#define XUN_FORMAT_7_BITS
#define XUN_FORMAT_6_BITS
#define XUN_FORMAT_5_BITS
#define XUN_FORMAT_EVEN_PARITY
#define XUN_FORMAT_ODD_PARITY
#define XUN_FORMAT_NO_PARITY
#define XUN_FORMAT_2_STOP_BIT
#define XUN_FORMAT_1_STOP_BIT
```

Modem status values

```
#define XUN_MODEM_DCD_DELTA_MASK
#define XUN_MODEM_DSR_DELTA_MASK
#define XUN_MODEM_CTS_DELTA_MASK
#define XUN_MODEM_RINGING_MASK
#define XUN_MODEM_DSR_MASK
#define XUN_MODEM_CTS_MASK
#define XUN_MODEM_DCD_MASK
#define XUN_MODEM_RING_STOP_MASK
```

Callback events

```
#define XUN_EVENT_RECV_DATA
#define XUN_EVENT_RECV_TIMEOUT
#define XUN_EVENT_SENT_DATA
#define XUN_EVENT_RECV_ERROR
#define XUN_EVENT_MODEM
```

Error values

```
#define XUN_ERROR_BREAK_MASK
#define XUN_ERROR_FRAMING_MASK
#define XUN_ERROR_PARITY_MASK
#define XUN_ERROR_OVERRUN_MASK
#define XUN_ERROR_NONE
```

Typedefs

```
typedef void(* XUartNs550_Handler )(void *CallBackRef, Xuint32 Event, unsigned int
    EventData)
```


Functions

XStatus XUartNs550_Initialize (**XUartNs550** *InstancePtr, **Xuint16** DeviceId)
unsigned int **XUartNs550_Send** (**XUartNs550** *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)
unsigned int **XUartNs550_Recv** (**XUartNs550** *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)
XUartNs550_Config * **XUartNs550_LookupConfig** (**Xuint16** DeviceId)
XStatus XUartNs550_SetOptions (**XUartNs550** *InstancePtr, **Xuint16** Options)
Xuint16 XUartNs550_GetOptions (**XUartNs550** *InstancePtr)
XStatus XUartNs550_SetFifoThreshold (**XUartNs550** *InstancePtr, **Xuint8** TriggerLevel)
Xuint8 XUartNs550_GetFifoThreshold (**XUartNs550** *InstancePtr)
Xboolean XUartNs550_IsSending (**XUartNs550** *InstancePtr)
Xuint8 XUartNs550_GetLastErrors (**XUartNs550** *InstancePtr)
Xuint8 XUartNs550_GetModemStatus (**XUartNs550** *InstancePtr)
XStatus XUartNs550_SetDataFormat (**XUartNs550** *InstancePtr, **XUartNs550Format** *Format)
void **XUartNs550_GetDataFormat** (**XUartNs550** *InstancePtr, **XUartNs550Format** *Format)
void **XUartNs550_SetHandler** (**XUartNs550** *InstancePtr, **XUartNs550_Handler** FuncPtr, void *CallbackRef)
void **XUartNs550_InterruptHandler** (**XUartNs550** *InstancePtr)
void **XUartNs550_GetStats** (**XUartNs550** *InstancePtr, **XUartNs550Stats** *StatsPtr)
void **XUartNs550_ClearStats** (**XUartNs550** *InstancePtr)
XStatus XUartNs550_SelfTest (**XUartNs550** *InstancePtr)

Define Documentation

```
#define XUN_ERROR_BREAK_MASK
```

These constants specify the errors that may be retrieved from the driver using the XUartNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define XUN_ERROR_FRAMING_MASK

These constants specify the errors that may be retrieved from the driver using the XUartNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define XUN_ERROR_NONE

These constants specify the errors that may be retrieved from the driver using the XUartNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define XUN_ERROR_OVERRUN_MASK

These constants specify the errors that may be retrieved from the driver using the XUartNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define XUN_ERROR_PARITY_MASK

These constants specify the errors that may be retrieved from the driver using the XUartNs550_GetLastErrors function. All of them are bit masks, except no error, such that multiple errors may be specified.

XUN_ERROR_BREAK_MASK	Break detected
XUN_ERROR_FRAMING_MASK	Receive framing error
XUN_ERROR_PARITY_MASK	Receive parity error
XUN_ERROR_OVERRUN_MASK	Receive overrun error
XUN_ERROR_NONE	No error

#define XUN_EVENT_MODEM

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define XUN_EVENT_RECV_DATA

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define XUN_EVENT_RECV_ERROR

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define XUN_EVENT_RECV_TIMEOUT

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define XUN_EVENT_SENT_DATA

These constants specify the handler events that are passed to a handler from the driver. These constants are not bit masks such that only one will be passed at a time to the handler.

XUN_EVENT_RECV_DATA	Data has been received
XUN_EVENT_RECV_TIMEOUT	A receive timeout occurred
XUN_EVENT_SENT_DATA	Data has been sent
XUN_EVENT_RECV_ERROR	A receive error was detected
XUN_EVENT_MODEM	A change in modem status

#define XUN_FORMAT_1_STOP_BIT

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define XUN_FORMAT_2_STOP_BIT

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define XUN_FORMAT_5_BITS

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define XUN_FORMAT_6_BITS

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define XUN_FORMAT_7_BITS

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define XUN_FORMAT_8_BITS

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define XUN_FORMAT_EVEN_PARITY

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define XUN_FORMAT_NO_PARITY

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define XUN_FORMAT_ODD_PARITY

These constants specify the data format that may be set or retrieved with the driver. The data format includes the number of data bits, the number of stop bits and parity.

XUN_FORMAT_8_BITS	8 data bits
XUN_FORMAT_7_BITS	7 data bits
XUN_FORMAT_6_BITS	6 data bits
XUN_FORMAT_5_BITS	5 data bits
XUN_FORMAT_EVEN_PARITY	Even parity
XUN_FORMAT_ODD_PARITY	Odd parity
XUN_FORMAT_NO_PARITY	No parity
XUN_FORMAT_2_STOP_BIT	2 stop bits
XUN_FORMAT_1_STOP_BIT	1 stop bit

#define XUN_MODEM_CTS_DELTA_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringling has stopped

#define XUN_MODEM_CTS_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define XUN_MODEM_DCD_DELTA_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define XUN_MODEM_DCD_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define XUN_MODEM_DSR_DELTA_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define XUN_MODEM_DSR_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define XUN_MODEM_RING_STOP_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define XUN_MODEM_RINGING_MASK

These constants specify the modem status that may be retrieved from the driver.

XUN_MODEM_DCD_DELTA_MASK	DCD signal changed state
XUN_MODEM_DSR_DELTA_MASK	DSR signal changed state
XUN_MODEM_CTS_DELTA_MASK	CTS signal changed state
XUN_MODEM_RINGING_MASK	Ring signal is active
XUN_MODEM_DSR_MASK	Current state of DSR signal
XUN_MODEM_CTS_MASK	Current state of CTS signal
XUN_MODEM_DCD_MASK	Current state of DCD signal
XUN_MODEM_RING_STOP_MASK	Ringing has stopped

#define XUN_OPTION_ASSERT_DTR

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal

XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_ASSERT_OUT1

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_ASSERT_OUT2

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_ASSERT_RTS

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_DATA_INTR

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_FIFOS_ENABLE

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_LOOPBACK

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_MODEM_INTR

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_RESET_RX_FIFO

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_RESET_TX_FIFO

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

#define XUN_OPTION_SET_BREAK

These constants specify the options that may be set or retrieved with the driver, each is a unique bit mask such that multiple options may be specified. These constants indicate the function of the option when in the active state.

XUN_OPTION_SET_BREAK	Set a break condition
XUN_OPTION_LOOPBACK	Enable local loopback
XUN_OPTION_DATA_INTR	Enable data interrupts
XUN_OPTION_MODEM_INTR	Enable modem interrupts
XUN_OPTION_FIFOS_ENABLE	Enable FIFOs
XUN_OPTION_RESET_TX_FIFO	Reset the transmit FIFO
XUN_OPTION_RESET_RX_FIFO	Reset the receive FIFO
XUN_OPTION_ASSERT_OUT2	Assert out2 signal
XUN_OPTION_ASSERT_OUT1	Assert out1 signal
XUN_OPTION_ASSERT_RTS	Assert RTS signal
XUN_OPTION_ASSERT_DTR	Assert DTR signal

Typedef Documentation

```
typedef void(* XUartNs550_Handler)(void *CallbackRef, Xuint32 Event, unsigned int EventData)
```

This data type defines a handler which the application must define when using interrupt mode. The handler will be called from the driver in an interrupt context to handle application specific processing.

Parameters:

- CallbackRef* is a callback reference passed in by the upper layer when setting the handler, and is passed back to the upper layer when the handler is called.
- Event* contains one of the event constants indicating why the handler is being called.
- EventData* contains the number of bytes sent or received at the time of the call for send and receive events and contains the modem status for modem events.

Function Documentation

```
void XUartNs550_ClearStats( XUartNs550 * InstancePtr)
```

This function zeros the statistics for the given instance.

Parameters:

- InstancePtr* is a pointer to the **XUartNs550** instance to be worked on.

Returns:

None.

Note:

None.

```
void XUartNs550_GetDataFormat( XUartNs550 * InstancePtr,  
                               XUartNs550Format * FormatPtr  
                               )
```

Gets the data format for the specified UART. The data format includes the baud rate, number of data bits, number of stop bits, and parity.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FormatPtr is a pointer to a format structure that will contain the data format after this call completes.

Returns:

None.

Note:

None.

Xuint8 XUartNs550_GetFifoThreshold(XUartNs550 * InstancePtr)

This function gets the receive FIFO trigger level. The receive trigger level indicates the number of bytes in the receive FIFO that cause a receive data event (interrupt) to be generated.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The current receive FIFO trigger level. Constants which define each trigger level are contained in the file **xuartns550.h** and named XUN_FIFO_TRIGGER_*.

Note:

None.

Xuint8 XUartNs550_GetLastErrors(XUartNs550 * InstancePtr)

This function returns the last errors that have occurred in the specified UART. It also clears the errors such that they cannot be retrieved again. The errors include parity error, receive overrun error, framing error, and break detection.

The last errors is an accumulation of the errors each time an error is discovered in the driver. A status is checked for each received byte and this status is accumulated in the last errors.

If this function is called after receiving a buffer of data, it will indicate any errors that occurred for the bytes of the buffer. It does not indicate which bytes contained errors.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The last errors that occurred. The errors are bit masks that are contained in the file **xuartns550.h** and named XUN_ERROR_*.

Note:

None.

Xuint8 XUartNs550_GetModemStatus(XUartNs550 * InstancePtr)

This function gets the modem status from the specified UART. The modem status indicates any changes of the modem signals. This function allows the modem status to be read in a polled mode. The modem status is updated whenever it is read such that reading it twice may not yield the same results.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The modem status which are bit masks that are contained in the file **xuartns550.h** and named XUN_MODEM_*.

Note:

The bit masks used for the modem status are the exact bits of the modem status register with no abstraction.

Xuint16 XUartNs550_GetOptions(XUartNs550 * InstancePtr)

Gets the options for the specified driver instance. The options are implemented as bit masks such that multiple options may be enabled or disabled simulataneously.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The current options for the UART. The options are bit masks that are contained in the file **xuartns550.h** and named XUN_OPTION_*.

Returns:

None.

```
void XUartNs550_GetStats( XUartNs550 * InstancePtr,  
                        XUartNs550Stats * StatsPtr  
                        )
```

This functions returns a snapshot of the current statistics in the area provided.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

StatsPtr is a pointer to a **XUartNs550Stats** structure to where the statistics are to be copied to.

Returns:

None.

Note:

None.

```
XStatus XUartNs550_Initialize( XUartNs550 * InstancePtr,  
                              Xuint16 DeviceId  
                              )
```

Initializes a specific **XUartNs550** instance such that it is ready to be used. The data format of the device is setup for 8 data bits, 1 stop bit, and no parity by default. The baud rate is set to a default value specified by XPAR_DEFAULT_BAUD_RATE if the symbol is defined, otherwise it is set to 19.2K baud. If the device has FIFOs (16550), they are enabled and the a receive FIFO threshold is set for 8 bytes. The default operating mode of the driver is polled mode.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XUartNs550** instance. Passing in a device id associates the generic **XUartNs550** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if the device ID could not be found in the configuration table
- XST_UART_BAUD_ERROR if the baud rate is not possible because the input clock frequency is not divisible with an acceptable amount of error

Note:

None.

```
void XUartNs550_InterruptHandler( XUartNs550 * InstancePtr)
```

This function is the interrupt handler for the 16450/16550 UART driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any 16450/16550 UART occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the UART that the interrupt is for.

Returns:

None.

Note:

None.

Xboolean XUartNs550_IsSending(XUartNs550 * InstancePtr)

This function determines if the specified UART is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

A value of XTRUE if the UART is sending data, otherwise XFALSE.

Note:

None.

XUartNs550_Config* XUartNs550_LookupConfig(Xuint16 DeviceId)

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found.

Note:

None.

```
unsigned int XUartNs550_Recv( XUartNs550 * InstancePtr,  
                             Xuint8 * BufferPtr,  
                             unsigned int NumBytes  
                             )
```

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the `SetOptions` function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

XStatus XUartNs550_SelfTest(XUartNs550 * InstancePtr)

This functions runs a self-test on the driver and hardware device. This self test performs a local loopback and verifies data can be sent and received.

The statistics are cleared at the end of the test. The time for this test to execute is proportional to the baud rate that has been set prior to calling this function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

- XST_SUCCESS if the test was successful
- XST_UART_TEST_FAIL if the test failed looping back the data

Note:

This function can hang if the hardware is not functioning properly.

```
unsigned int XUartNs550_Send( XUartNs550 * InstancePtr,  
                             Xuint8 * BufferPtr,  
                             unsigned int NumBytes  
                             )
```

This functions sends the specified buffer of data using the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART. If the UART is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.

NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

This function and the **XUartNs550_SetOptions()** function modify shared data such that there may be a need for mutual exclusion in a multithreaded environment and if **XUartNs550_SetOptions()** if called from a handler.

```
XStatus XUartNs550_SetDataFormat( XUartNs550 * InstancePtr,
                                   XUartNs550Format * FormatPtr
                                   )
```

Sets the data format for the specified UART. The data format includes the baud rate, number of data bits, number of stop bits, and parity. It is the caller's responsibility to ensure that the UART is not sending or receiving data when this function is called.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FormatPtr is a pointer to a format structure containing the data format to be set.

Returns:

- XST_SUCCESS if the data format was successfully set.
- XST_UART_BAUD_ERROR indicates the baud rate could not be set because of the amount of error with the baud rate and the input clock frequency.
- XST_INVALID_PARAM if one of the parameters was not valid.

Note:

The data types in the format type, data bits and parity, are 32 bit fields to prevent a compiler warning that is a bug with the GNU PowerPC compiler. The asserts in this function will cause a warning if these fields are bytes.

The baud rates tested include: 1200, 2400, 4800, 9600, 19200, 38400, 57600 and 115200.

```
XStatus XUartNs550_SetFifoThreshold( XUartNs550 * InstancePtr,
                                     Xuint8 TriggerLevel
                                     )
```

This functions sets the receive FIFO trigger level. The receive trigger level specifies the number of bytes in the receive FIFO that cause a receive data event (interrupt) to be generated. The FIFOs must be enabled to set the trigger level.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

TriggerLevel contains the trigger level to set. Constants which define each trigger level are contained in the file **xuartns550.h** and named XUN_FIFO_TRIGGER_*.

Returns:

- XST_SUCCESS if the trigger level was set

- XST_UART_CONFIG_ERROR if the trigger level could not be set, either the hardware does not support the FIFOs or FIFOs are not enabled

Note:

None.

```
void XUartNs550_SetHandler( XUartNs550 * InstancePtr,  
                           XUartNs550_Handler FuncPtr,  
                           void * CallbackRef  
                           )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the CallbackRef since the driver doesn't know what it is (nor should it)

```
XStatus XUartNs550_SetOptions( XUartNs550 * InstancePtr,  
                               Xuint16 Options  
                               )
```

Sets the options for the specified driver instance. The options are implemented as bit masks such that multiple options may be enabled or disabled simultaneously.

The `GetOptions` function may be called to retrieve the currently enabled options. The result is ORed in the desired new settings to be enabled and ANDed with the inverse to clear the settings to be disabled. The resulting value is then used as the options for the `SetOption` function call.

Parameters:

InstancePtr is a pointer to the `XUartNs550` instance to be worked on.

Options contains the options to be set which are bit masks contained in the file `xuartns550.h` and named `XUN_OPTION_*`.

Returns:

- `XST_SUCCESS` if the options were set successfully.
- `XST_UART_CONFIG_ERROR` if the options could not be set because the hardware does not support FIFOs

Note:

None.

uartns550/v1_00_b/src/xuartns550.c File Reference

Detailed Description

This file contains the required functions for the 16450/16550 UART driver. Refer to the header file [xuartns550.h](#) for more detailed information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.
1.00b	rmm	05/14/03	Fixed diab compiler warnings relating to asserts.

```
#include "xstatus.h"  
#include "xparameters.h"  
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Functions

XStatus XUartNs550_Initialize (**XUartNs550** *InstancePtr, **Xuint16** DeviceId)
unsigned int **XUartNs550_Send** (**XUartNs550** *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)

unsigned int **XUartNs550_Recv** (**XUartNs550** *InstancePtr, **Xuint8** *BufferPtr, unsigned int NumBytes)

unsigned int **XUartNs550_SendBuffer** (**XUartNs550** *InstancePtr)

unsigned int **XUartNs550_ReceiveBuffer** (**XUartNs550** *InstancePtr)

XUartNs550_Config * **XUartNs550_LookupConfig** (**Xuint16** DeviceId)

Function Documentation

```
XStatus XUartNs550_Initialize( XUartNs550 * InstancePtr,  
                               Xuint16      DeviceId  
                               )
```

Initializes a specific **XUartNs550** instance such that it is ready to be used. The data format of the device is setup for 8 data bits, 1 stop bit, and no parity by default. The baud rate is set to a default value specified by XPAR_DEFAULT_BAUD_RATE if the symbol is defined, otherwise it is set to 19.2K baud. If the device has FIFOs (16550), they are enabled and the a receive FIFO threshold is set for 8 bytes. The default operating mode of the driver is polled mode.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XUartNs550** instance. Passing in a device id associates the generic **XUartNs550** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_NOT_FOUND if the device ID could not be found in the configuration table
- XST_UART_BAUD_ERROR if the baud rate is not possible because the input clock frequency is not divisible with an acceptable amount of error

Note:

None.

```
XUartNs550_Config* XUartNs550_LookupConfig( Xuint16 DeviceId)
```

Looks up the device configuration based on the unique device ID. A table contains the configuration info for each device in the system.

Parameters:

DeviceId contains the ID of the device to look up the configuration for.

Returns:

A pointer to the configuration found or XNULL if the specified device ID was not found.

Note:

None.

```
unsigned int XUartNs550_ReceiveBuffer( XUartNs550 * InstancePtr)
```

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartNs550** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

```
unsigned int XUartNs550_Recv( XUartNs550 * InstancePtr,  
                             Xuint8 * BufferPtr,  
                             unsigned int NumBytes  
                             )
```

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue receiving data until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

BufferPtr is pointer to buffer for data to be received into

NumBytes is the number of bytes to be received. A value of zero will stop a previous receive operation that is in progress in interrupt mode.

Returns:

The number of bytes received.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

```
unsigned int XUartNs550_Send( XUartNs550 * InstancePtr,  
                              Xuint8 * BufferPtr,  
                              unsigned int NumBytes  
                              )
```


This function sends the specified buffer of data using the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART. If the UART is busy sending data, it will return and indicate zero bytes were sent.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue sending data until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

BufferPtr is pointer to a buffer of data to be sent.

NumBytes contains the number of bytes to be sent. A value of zero will stop a previous send operation that is in progress in interrupt mode. Any data that was already put into the transmit FIFO will be sent.

Returns:

The number of bytes actually sent.

Note:

The number of bytes is not asserted so that this function may be called with a value of zero to stop an operation that is already in progress.

This function and the **XUartNs550_SetOptions()** function modify shared data such that there may be a need for mutual exclusion in a multithreaded environment and if **XUartNs550_SetOptions()** is called from a handler.

```
unsigned int XUartNs550_SendBuffer( XUartNs550 * InstancePtr)
```

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartNs550** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent (put into the UART transmitter and/or FIFO).

Note:

None.

uartns550/v1_00_b/src/xuartns550_l.h

File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. High-level driver functions are defined in [xuartns550.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XUartNs550_mReadReg(BaseAddress, RegOffset)  
#define XUartNs550_mWriteReg(BaseAddress, RegOffset, RegisterValue)  
#define XUartNs550_mGetLineStatusReg(BaseAddress)  
#define XUartNs550_mGetLineControlReg(BaseAddress)  
#define XUartNs550_mSetLineControlReg(BaseAddress, RegisterValue)
```

```
#define XUartNs550_mEnableIntr(BaseAddress)
#define XUartNs550_mDisableIntr(BaseAddress)
#define XUartNs550_mIsReceiveData(BaseAddress)
#define XUartNs550_mIsTransmitEmpty(BaseAddress)
```

Functions

```
void XUartNs550_SendByte (Xuint32 BaseAddress, Xuint8 Data)
Xuint8 XUartNs550_RecvByte (Xuint32 BaseAddress)
```

Define Documentation

```
#define XUartNs550_mDisableIntr( BaseAddress )
```

Disable the transmit and receive interrupts of the UART.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XUartNs550_mEnableIntr( BaseAddress )
```

Enable the transmit and receive interrupts of the UART.

Parameters:

BaseAddress contains the base address of the device.

Returns:

None.

Note:

None.

```
#define XUartNs550_mGetLineControlReg( BaseAddress )
```

Get the UART Line Status Register.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XUartNs550_mGetLineStatusReg( BaseAddress )
```

Get the UART Line Status Register.

Parameters:

BaseAddress contains the base address of the device.

Returns:

The value read from the register.

Note:

None.

```
#define XUartNs550_mIsReceiveData( BaseAddress )
```

Determine if there is receive data in the receiver and/or FIFO.

Parameters:

BaseAddress contains the base address of the device.

Returns:

XTRUE if there is receive data, XFALSE otherwise.

Note:

None.

```
#define XUartNs550_mIsTransmitEmpty( BaseAddress )
```

Determine if a byte of data can be sent with the transmitter.

Parameters:

BaseAddress contains the base address of the device.

Returns:

XTRUE if a byte can be sent, XFALSE otherwise.

Note:

None.

```
#define XUartNs550_mReadReg( BaseAddress,  
                             RegOffset )
```

Read a UART register.

Parameters:

BaseAddress contains the base address of the device.

RegOffset contains the offset from the 1st register of the device to select the specific register.

Returns:

The value read from the register.

Note:

None.

```
#define XUartNs550_mSetLineControlReg( BaseAddress,  
                                       RegisterValue )
```

Set the UART Line Status Register.

Parameters:

BaseAddress contains the base address of the device.

RegisterValue is the value to be written to the register.

Returns:

None.

Note:

None.

```
#define XUartNs550_mWriteReg( BaseAddress,  
                              RegOffset,  
                              RegisterValue )
```

Write to a UART register.

Parameters:

BaseAddress contains the base address of the device.

RegOffset contains the offset from the 1st register of the device to select the specific register.

Returns:

The value read from the register.

Note:

None.

Function Documentation

Xuint8 XUartNs550_RecvByte(**Xuint32** *BaseAddress*)

This function receives a byte from the UART. It operates in a polling mode and blocks until a byte of data is received.

Parameters:

BaseAddress contains the base address of the UART.

Returns:

The data byte received by the UART.

Note:

None.

```
void XUartNs550_SendByte( Xuint32 BaseAddress,  
                          Xuint8 Data  
                          )
```


This function sends a data byte with the UART. This function operates in the polling mode and blocks until the data has been put into the UART transmit holding register.

Parameters:

BaseAddress contains the base address of the UART.

Data contains the data byte to be sent.

Returns:

None.

Note:

None.

uartns550/v1_00_b/src/xuartns550_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xuartns550_l.h,v 1.2 2002/05/17 18:31:39 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 **
00024 *
00025 * @file uartns550/v1_00_b/src/xuartns550_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  The user should refer to the
00029 * hardware device specification for more details of the device operation.
00030 * High-level driver functions are defined in xuartns550.h.
00031 *
00032 * <pre>
00033 * MODIFICATION HISTORY:
00034 *
00035 * Ver   Who   Date       Changes
00036 * -----
00037 * 1.00b jhl   04/24/02 First release
00038 * </pre>
00039 *
00040 *****/
00041
00042 #ifndef XUARTNS550_L_H /* prevent circular inclusions */
```

```

00043 #define XUARTNS550_L_H /* by using protection macros */
00044
00045 /***** Include Files *****/
00046
00047 #include "xbasic_types.h"
00048 #include "xio.h"
00049
00050 /***** Constant Definitions *****/
00051
00052 /*
00053  * Offset from the device base address (IPIF) to the IP registers.
00054  */
00055 #define XUN_REG_OFFSET 0x1000
00056
00057 /* 16450/16550 compatible UART, register offsets as byte registers */
00058
00059 #define XUN_RBR_OFFSET (XUN_REG_OFFSET + 0x03) /* receive buffer, read only */
00060 #define XUN_THR_OFFSET (XUN_REG_OFFSET + 0x03) /* transmit holding register */
00061 #define XUN_IER_OFFSET (XUN_REG_OFFSET + 0x07) /* interrupt enable */
00062 #define XUN_IIR_OFFSET (XUN_REG_OFFSET + 0x0B) /* interrupt id, read only */
00063 #define XUN_FCR_OFFSET (XUN_REG_OFFSET + 0x0B) /* fifo control, write only */
00064 #define XUN_LCR_OFFSET (XUN_REG_OFFSET + 0x0F) /* line control register */
00065 #define XUN_MCR_OFFSET (XUN_REG_OFFSET + 0x13) /* modem control register */
00066 #define XUN_LSR_OFFSET (XUN_REG_OFFSET + 0x17) /* line status register */
00067 #define XUN_MSR_OFFSET (XUN_REG_OFFSET + 0x1B) /* modem status register */
00068 #define XUN_DRLS_OFFSET (XUN_REG_OFFSET + 0x03) /* divisor register LSB */
00069 #define XUN_DRLM_OFFSET (XUN_REG_OFFSET + 0x07) /* divisor register MSB */
00070
00071 /* the following constant specifies the size of the FIFOs, the size of the
00072  * FIFOs includes the transmitter and receiver such that it is the total number
00073  * of bytes that the UART can buffer
00074  */
00075 #define XUN_FIFO_SIZE 16
00076
00077 /* Interrupt Enable Register bits */
00078
00079 #define XUN_IER_MODEM_STATUS 0x08 /* modem status interrupt */
00080 #define XUN_IER_RX_LINE 0x04 /* receive status interrupt */
00081 #define XUN_IER_TX_EMPTY 0x02 /* transmitter empty interrupt */
00082 #define XUN_IER_RX_DATA 0x01 /* receiver data available */
00083
00084 /* Interrupt ID Register bits */
00085
00086 #define XUN_INT_ID_MASK 0x0F /* only the interrupt ID */
00087 #define XUN_INT_ID_FIFOS_ENABLED 0xC0 /* only the FIFOs enable */
00088
00089 /* FIFO Control Register bits */
00090
00091 #define XUN_FIFO_RX_TRIG_MSB 0x80 /* trigger level MSB */
00092 #define XUN_FIFO_RX_TRIG_LSB 0x40 /* trigger level LSB */
00093 #define XUN_FIFO_TX_RESET 0x04 /* reset the transmit FIFO */
00094 #define XUN_FIFO_RX_RESET 0x02 /* reset the receive FIFO */

```

```

00095 #define XUN_FIFO_ENABLE          0x01    /* enable the FIFOs */
00096 #define XUN_FIFO_RX_TRIGGER      0xC0    /* both trigger level bits */
00097
00098 /* Line Control Register bits */
00099
00100 #define XUN_LCR_DLAB              0x80    /* divisor latch access */
00101 #define XUN_LCR_SET_BREAK        0x40    /* cause a break condition */
00102 #define XUN_LCR_STICK_PARITY     0x20
00103 #define XUN_LCR_EVEN_PARITY      0x10    /* 1 = even, 0 = odd parity */
00104 #define XUN_LCR_ENABLE_PARITY    0x08
00105 #define XUN_LCR_2_STOP_BITS      0x04    /* 1 = 2 stop bits, 0 = 1 stop bit
*/
00106 #define XUN_LCR_8_DATA_BITS      0x03
00107 #define XUN_LCR_7_DATA_BITS      0x02
00108 #define XUN_LCR_6_DATA_BITS      0x01
00109 #define XUN_LCR_LENGTH_MASK     0x03    /* both length bits mask */
00110 #define XUN_LCR_PARITY_MASK     0x18    /* both parity bits mask */
00111
00112 /* Modem Control Register bits */
00113
00114 #define XUN_MCR_LOOP              0x10    /* local loopback */
00115 #define XUN_MCR_OUT_2            0x08    /* general output 2 signal */
00116 #define XUN_MCR_OUT_1            0x04    /* general output 1 signal */
00117 #define XUN_MCR_RTS              0x02    /* RTS signal */
00118 #define XUN_MCR_DTR              0x01    /* DTR signal */
00119
00120 /* Line Status Register bits */
00121
00122 #define XUN_LSR_RX_FIFO_ERROR    0x80    /* an errored byte is in the FIFO
*/
00123 #define XUN_LSR_TX_EMPTY        0x40    /* transmitter is empty */
00124 #define XUN_LSR_TX_BUFFER_EMPTY  0x20    /* transmit holding reg empty */
00125 #define XUN_LSR_BREAK_INT       0x10    /* break detected interrupt */
00126 #define XUN_LSR_FRAMING_ERROR    0x08    /* framing error on current byte */
00127 #define XUN_LSR_PARITY_ERROR     0x04    /* parity error on current byte */
00128 #define XUN_LSR_OVERRUN_ERROR    0x02    /* overrun error on receive FIFO */
00129 #define XUN_LSR_DATA_READY      0x01    /* receive data ready */
00130 #define XUN_LSR_ERROR_BREAK     0x1E    /* errors except FIFO error and
break detected */
00131
00132
00133 #define XUN_DIVISOR_BYTE_MASK    0xFF
00134
00135 /****** Type Definitions *****/
00136
00137
00138 /****** Macros (Inline Functions) Definitions *****/
00139
00140
00141 /******
00142 *
00143 * Low-level driver macros. The list below provides signatures to help the
00144 * user use the macros.
00145 *

```

```

00146 * Xuint8 XUartNs550_mReadReg(Xuint32 BaseAddress, int RegOffset)
00147 * void XUartNs550_mWriteReg(Xuint32 BaseAddress, int RegOffset,
00148 *                               Xuint8 RegisterValue)
00149 *
00150 * Xuint8 XUartNs550_mGetLineStatusReg(Xuint32 BaseAddress)
00151 * Xuint8 XUartNs550_mGetLineControlReg(Xuint32 BaseAddress)
00152 * void XUartNs550_mSetLineControlReg(Xuint32 BaseAddress, Xuint8 RegisterValue)
00153 *
00154 * void XUartNs550_mEnableIntr(Xuint32 BaseAddress)
00155 * void XUartNs550_mDisableIntr(Xuint32 BaseAddress)
00156 *
00157 * Xboolean XUartNs550_mIsReceiveData(Xuint32 BaseAddress)
00158 * Xboolean XUartNs550_mIsTransmitEmpty(Xuint32 BaseAddress)
00159 *
00160 *****/
00161
00162 /***/
00163 /**
00164 * Read a UART register.
00165 *
00166 * @param BaseAddress contains the base address of the device.
00167 * @param RegOffset contains the offset from the 1st register of the device
00168 * to select the specific register.
00169 *
00170 * @return The value read from the register.
00171 *
00172 * @note None.
00173 *
00174 *****/
00175 #define XUartNs550_mReadReg(BaseAddress, RegOffset) \
00176     XIo_In8((BaseAddress) + (RegOffset))
00177
00178 *****/
00179 /**
00180 * Write to a UART register.
00181 *
00182 * @param BaseAddress contains the base address of the device.
00183 * @param RegOffset contains the offset from the 1st register of the device
00184 * to select the specific register.
00185 *
00186 * @return The value read from the register.
00187 *
00188 * @note None.
00189 *
00190 *****/
00191 #define XUartNs550_mWriteReg(BaseAddress, RegOffset, RegisterValue) \
00192     XIo_Out8((BaseAddress) + (RegOffset), (RegisterValue))
00193
00194 *****/
00195 /**
00196 * Get the UART Line Status Register.
00197 *

```

```

00198 * @param      BaseAddress contains the base address of the device.
00199 *
00200 * @return      The value read from the register.
00201 *
00202 * @note        None.
00203 *
00204 *****/
00205 #define XUartNs550_mGetLineStatusReg(BaseAddress) \
00206     XIo_In8((BaseAddress) + XUN_LSR_OFFSET)
00207
00208 *****/
00209 /**
00210 * Get the UART Line Status Register.
00211 *
00212 * @param      BaseAddress contains the base address of the device.
00213 *
00214 * @return      The value read from the register.
00215 *
00216 * @note        None.
00217 *
00218 *****/
00219 #define XUartNs550_mGetLineControlReg(BaseAddress) \
00220     XIo_In8((BaseAddress) + XUN_LCR_OFFSET)
00221
00222 *****/
00223 /**
00224 * Set the UART Line Status Register.
00225 *
00226 * @param      BaseAddress contains the base address of the device.
00227 * @param      RegisterValue is the value to be written to the register.
00228 *
00229 * @return      None.
00230 *
00231 * @note        None.
00232 *
00233 *****/
00234 #define XUartNs550_mSetLineControlReg(BaseAddress, RegisterValue) \
00235     XIo_Out8((BaseAddress) + XUN_LCR_OFFSET, (RegisterValue))
00236
00237 *****/
00238 /**
00239 * Enable the transmit and receive interrupts of the UART.
00240 *
00241 * @param      BaseAddress contains the base address of the device.
00242 *
00243 * @return      None.
00244 *
00245 * @note        None.
00246 *
00247 *****/
00248 #define XUartNs550_mEnableIntr(BaseAddress) \
00249     XIo_Out8((BaseAddress) + XUN_LCR_OFFSET, \

```

```

00250         XIo_In8((BaseAddress) + XUN_IER_OFFSET) | \
00251         (XUN_IER_RX_LINE | XUN_IER_TX_EMPTY | XUN_IER_RX_DATA))
00252
00253 /*****
00254 /**
00255 * Disable the transmit and receive interrupts of the UART.
00256 *
00257 * @param   BaseAddress contains the base address of the device.
00258 *
00259 * @return  None.
00260 *
00261 * @note    None.
00262 *
00263 *****/
00264 #define XUartNs550_mDisableIntr(BaseAddress) \
00265     XIo_Out8((BaseAddress) + XUN_LCR_OFFSET, \
00266     XIo_In8((BaseAddress) + XUN_IER_OFFSET) & \
00267     ~(XUN_IER_RX_LINE | XUN_IER_TX_EMPTY | XUN_IER_RX_DATA))
00268
00269 /*****
00270 /**
00271 * Determine if there is receive data in the receiver and/or FIFO.
00272 *
00273 * @param   BaseAddress contains the base address of the device.
00274 *
00275 * @return  XTRUE if there is receive data, XFALSE otherwise.
00276 *
00277 * @note    None.
00278 *
00279 *****/
00280 #define XUartNs550_mIsReceiveData(BaseAddress) \
00281     (XIo_In8((BaseAddress) + XUN_LSR_OFFSET) & XUN_LSR_DATA_READY)
00282
00283 /*****
00284 /**
00285 * Determine if a byte of data can be sent with the transmitter.
00286 *
00287 * @param   BaseAddress contains the base address of the device.
00288 *
00289 * @return  XTRUE if a byte can be sent, XFALSE otherwise.
00290 *
00291 * @note    None.
00292 *
00293 *****/
00294 #define XUartNs550_mIsTransmitEmpty(BaseAddress) \
00295     (XIo_In8((BaseAddress) + XUN_LSR_OFFSET) & XUN_LSR_TX_BUFFER_EMPTY)
00296
00297 /***** Function Prototypes *****/
00298
00299 void XUartNs550_SendByte(Xuint32 BaseAddress, Xuint8 Data);
00300
00301 Xuint8 XUartNs550_RecvByte(Xuint32 BaseAddress);

```

```
00302
00303 void XUartNs550_SetBaud(Xuint32 BaseAddress, Xuint32 InputClockHz,
00304                          Xuint32 BaudRate);
00305
00306 /***** Variable Definitions *****/
00307
00308 #endif          /* end of protection macro */
00309
```

Generated on 29 May 2003 for Xilinx Device Drivers

uartns550/v1_00_b/src/xuartns550_l.c

File Reference

Detailed Description

This file contains low-level driver functions that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00b	jhl	04/24/02	First release

```
#include "xuartns550_l.h"
```

Functions

```
void XUartNs550_SendByte (Xuint32 BaseAddress, Xuint8 Data)  
Xuint8 XUartNs550_RecvByte (Xuint32 BaseAddress)
```

Function Documentation

```
Xuint8 XUartNs550_RecvByte( Xuint32 BaseAddress)
```

This function receives a byte from the UART. It operates in a polling mode and blocks until a byte of data is received.

Parameters:

BaseAddress contains the base address of the UART.

Returns:

The data byte received by the UART.

Note:

None.

```
void XUartNs550_SendByte( Xuint32 BaseAddress,  
                          Xuint8 Data  
                          )
```

This function sends a data byte with the UART. This function operates in the polling mode and blocks until the data has been put into the UART transmit holding register.

Parameters:

BaseAddress contains the base address of the UART.

Data contains the data byte to be sent.

Returns:

None.

Note:

None.

XUartNs550_Config Struct Reference

```
#include <xuartns550.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddress](#)

[Xuint32 InputClockHz](#)

Field Documentation

[Xuint32 XUartNs550_Config::BaseAddress](#)

Base address of device (IPIF)

[Xuint16 XUartNs550_Config::DeviceId](#)

Unique ID of device

[Xuint32 XUartNs550_Config::InputClockHz](#)

Input clock frequency

The documentation for this struct was generated from the following file:

- [uartns550/v1_00_b/src/xuartns550.h](#)

Generated on 29 May 2003 for Xilinx Device Drivers

XUartNs550Format Struct Reference

```
#include <xuartns550.h>
```

Detailed Description

This data type allows the data format of the device to be set and retrieved.

Data Fields

[Xuint32 BaudRate](#)

[Xuint32 DataBits](#)

[Xuint32 Parity](#)

[Xuint8 StopBits](#)

Field Documentation

[Xuint32 XUartNs550Format::BaudRate](#)

In bps, ie 1200

[Xuint32 XUartNs550Format::DataBits](#)

Number of data bits

[Xuint32 XUartNs550Format::Parity](#)

Parity

Xuint8 XUartNs550Format::StopBits

Number of stop bits

The documentation for this struct was generated from the following file:

- [uartns550/v1_00_b/src/xuartns550.h](#)
-

Generated on 29 May 2003 for Xilinx Device Drivers

XUartNs550Stats Struct Reference

```
#include <xuartns550.h>
```

Detailed Description

UART statistics

Data Fields

[Xuint16 TransmitInterrupts](#)

[Xuint16 ReceiveInterrupts](#)

[Xuint16 StatusInterrupts](#)

[Xuint16 ModemInterrupts](#)

[Xuint16 CharactersTransmitted](#)

[Xuint16 CharactersReceived](#)

[Xuint16 ReceiveOverrunErrors](#)

[Xuint16 ReceiveParityErrors](#)

[Xuint16 ReceiveFramingErrors](#)

[Xuint16 ReceiveBreakDetected](#)

Field Documentation

[Xuint16 XUartNs550Stats::CharactersReceived](#)

Number of characters received

Xuint16 XUartNs550Stats::CharactersTransmitted

Number of characters transmitted

Xuint16 XUartNs550Stats::ModemInterrupts

Number of modem interrupts

Xuint16 XUartNs550Stats::ReceiveBreakDetected

Number of receive breaks

Xuint16 XUartNs550Stats::ReceiveFramingErrors

Number of receive framing errors

Xuint16 XUartNs550Stats::ReceiveInterrupts

Number of receive interrupts

Xuint16 XUartNs550Stats::ReceiveOverrunErrors

Number of receive overruns

Xuint16 XUartNs550Stats::ReceiveParityErrors

Number of receive parity errors

Xuint16 XUartNs550Stats::StatusInterrupts

Number of status interrupts

Xuint16 XUartNs550Stats::TransmitInterrupts

Number of transmit interrupts

The documentation for this struct was generated from the following file:

- [uartns550/v1_00_b/src/xuartns550.h](#)

XUartNs550 Struct Reference

```
#include <xuartns550.h>
```

Detailed Description

The XUartNs550 driver instance data. The user is required to allocate a variable of this type for every UART 16550/16450 device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [uartns550/v1_00_b/src/xuartns550.h](#)

uartns550/v1_00_b/src/xuartns550_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xuartns550_i.h,v 1.4 2002/05/02 20:56:21 linnj Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file uartns550/v1_00_b/src/xuartns550_i.h
00026 *
00027 * This header file contains internal identifiers, which are those shared
00028 * between the files of the driver. It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date      Changes
00034 * -----
00035 * 1.00a  ecm    08/16/01  First release
00036 * 1.00b  jhl    03/11/02  Repartitioned driver for smaller files.
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XUARTNS550_I_H /* prevent circular inclusions */
00042 #define XUARTNS550_I_H /* by using protection macros */
```

```

00043
00044 /***** Include Files *****/
00045
00046 #include "xuartns550.h"
00047
00048 /***** Constant Definitions *****/
00049
00050
00051 /***** Constant Definitions *****/
00052
00053
00054 /***** Type Definitions *****/
00055
00056
00057 /***** Macros (Inline Functions) Definitions *****/
00058
00059
00060 /*****
00061 *
00062 * This macro updates the status based upon a specified line status register
00063 * value. The stats that are updated are based upon bits in this register. It
00064 * also keeps the last errors instance variable updated. The purpose of this
00065 * macro is to allow common processing between the modules of the component
00066 * with less overhead than a function in the required module.
00067 *
00068 * @param InstancePtr is a pointer to the XUartNs550 instance to be worked
on.
00069 * @param CurrentLsr contains the Line Status Register value to be used for
00070 * the update.
00071 *
00072 * @return
00073 *
00074 * None.
00075 *
00076 * @note
00077 *
00078 * Signature:
00079 * void XUartNs550_mUpdateStats(XUartNs550 *InstancePtr, Xuint8 CurrentLsr)
00080 *
00081 *****/
00082 #define XUartNs550_mUpdateStats(InstancePtr, CurrentLsr) \
00083 { \
00084     InstancePtr->LastErrors |= CurrentLsr; \
00085 \
00086     if (CurrentLsr & XUN_LSR_OVERRUN_ERROR) \
00087     { \
00088         InstancePtr->Stats.ReceiveOverrunErrors++; \
00089     } \
00090     if (CurrentLsr & XUN_LSR_PARITY_ERROR) \
00091     { \
00092         InstancePtr->Stats.ReceiveParityErrors++; \
00093     } \

```

```

00094     if (CurrentLsr & XUN_LSR_FRAMING_ERROR) \
00095     { \
00096         InstancePtr->Stats.ReceiveFramingErrors++; \
00097     } \
00098     if (CurrentLsr & XUN_LSR_BREAK_INT) \
00099     { \
00100         InstancePtr->Stats.ReceiveBreakDetected++; \
00101     } \
00102 } \
00103 \
00104 /***** \
00105 * \
00106 * This macro clears the statistics of the component instance. The purpose of \
00107 * this macro is to allow common processing between the modules of the \
00108 * component with less overhead than a function in the required module. \
00109 * \
00110 * @param InstancePtr is a pointer to the XUartNs550 instance to be worked \
00111 * on. \
00112 * @return \
00113 * \
00114 * None. \
00115 * \
00116 * @note \
00117 * \
00118 * Signature: void XUartNs550_mClearStats(XUartNs550 *InstancePtr) \
00119 * \
00120 *****/ \
00121 #define XUartNs550_mClearStats(InstancePtr) \
00122 { \
00123     InstancePtr->Stats.TransmitInterrupts = 0UL; \
00124     InstancePtr->Stats.ReceiveInterrupts = 0UL; \
00125     InstancePtr->Stats.StatusInterrupts = 0UL; \
00126     InstancePtr->Stats.ModemInterrupts = 0UL; \
00127     InstancePtr->Stats.CharactersTransmitted = 0UL; \
00128     InstancePtr->Stats.CharactersReceived = 0UL; \
00129     InstancePtr->Stats.ReceiveOverrunErrors = 0UL; \
00130     InstancePtr->Stats.ReceiveFramingErrors = 0UL; \
00131     InstancePtr->Stats.ReceiveParityErrors = 0UL; \
00132     InstancePtr->Stats.ReceiveBreakDetected = 0UL; \
00133 } \
00134 \
00135 /***** Function Prototypes *****/ \
00136 \
00137 XStatus XUartNs550_SetBaudRate(XUartNs550 *InstancePtr, Xuint32 BaudRate); \
00138 \
00139 unsigned int XUartNs550_SendBuffer(XUartNs550 *InstancePtr); \
00140 \
00141 unsigned int XUartNs550_ReceiveBuffer(XUartNs550 *InstancePtr); \
00142 \
00143 /***** Variable Definitions *****/ \
00144

```

```
00145 extern XUartNs550_Config XUartNs550_ConfigTable[];
00146
00147 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

uartns550/v1_00_b/src/xuartns550_i.h

File Reference

Detailed Description

This header file contains internal identifiers, which are those shared between the files of the driver. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xuartns550.h"
```

[Go to the source code of this file.](#)

Functions

unsigned int [XUartNs550_SendBuffer](#) ([XUartNs550](#) *InstancePtr)

unsigned int [XUartNs550_ReceiveBuffer](#) ([XUartNs550](#) *InstancePtr)

Variables

Function Documentation

unsigned int XUartNs550_ReceiveBuffer(XUartNs550 * InstancePtr)

This function receives a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartNs550** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function will attempt to receive a specified number of bytes of data from the UART and store it into the specified buffer. This function is designed for either polled or interrupt driven modes. It is non-blocking such that it will return if there is no data has already received by the UART.

In a polled mode, this function will only receive as much data as the UART can buffer, either in the receiver or in the FIFO if present and enabled. The application may need to call it repeatedly to receive a buffer. Polled mode is the default mode of operation for the driver.

In interrupt mode, this function will start receiving and then the interrupt handler of the driver will continue until the buffer has been received. A callback function, as specified by the application, will be called to indicate the completion of receiving the buffer or when any receive errors or timeouts occur. Interrupt mode must be enabled using the SetOptions function.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The number of bytes received.

Note:

None.

unsigned int XUartNs550_SendBuffer(XUartNs550 * InstancePtr)

This function sends a buffer that has been previously specified by setting up the instance variables of the instance. This function is designed to be an internal function for the **XUartNs550** component such that it may be called from a shell function that sets up the buffer or from an interrupt handler.

This function sends the specified buffer of data to the UART in either polled or interrupt driven modes. This function is non-blocking such that it will return before the data has been sent by the UART.

In a polled mode, this function will only send as much data as the UART can buffer, either in the transmitter or in the FIFO if present and enabled. The application may need to call it repeatedly to send a buffer.

In interrupt mode, this function will start sending the specified buffer and then the interrupt handler of the driver will continue until the buffer has been sent. A callback function, as specified by the application, will be called to indicate the completion of sending the buffer.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

NumBytes is the number of bytes actually sent (put into the UART transmitter and/or FIFO).

Note:

None.

Variable Documentation

XUartNs550_Config XUartNs550_ConfigTable[]()

The configuration table for UART 16550/16450 devices in the table. Each device should have an entry in this table.

uartns550/v1_00_b/src/xuartns550_options.c

File Reference

Detailed Description

The implementation of the options functions for the [XUartNs550](#) driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00b	jhl	03/11/02	Repartitioned driver for smaller files.
-------	-----	----------	---

```
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Data Structures

```
struct Mapping
```

Functions

```
Xuint16 XUartNs550_GetOptions (XUartNs550 *InstancePtr)  
XStatus XUartNs550_SetOptions (XUartNs550 *InstancePtr, Xuint16 Options)  
Xuint8 XUartNs550_GetFifoThreshold (XUartNs550 *InstancePtr)  
XStatus XUartNs550_SetFifoThreshold (XUartNs550 *InstancePtr, Xuint8 TriggerLevel)  
Xuint8 XUartNs550_GetLastErrors (XUartNs550 *InstancePtr)  
Xuint8 XUartNs550_GetModemStatus (XUartNs550 *InstancePtr)
```

Function Documentation

Xuint8 XUartNs550_GetFifoThreshold(**XUartNs550** * *InstancePtr*)

This function gets the receive FIFO trigger level. The receive trigger level indicates the number of bytes in the receive FIFO that cause a receive data event (interrupt) to be generated.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The current receive FIFO trigger level. Constants which define each trigger level are contained in the file **xuartns550.h** and named XUN_FIFO_TRIGGER_*.

Note:

None.

Xuint8 XUartNs550_GetLastErrors(**XUartNs550** * *InstancePtr*)

This function returns the last errors that have occurred in the specified UART. It also clears the errors such that they cannot be retrieved again. The errors include parity error, receive overrun error, framing error, and break detection.

The last errors is an accumulation of the errors each time an error is discovered in the driver. A status is checked for each received byte and this status is accumulated in the last errors.

If this function is called after receiving a buffer of data, it will indicate any errors that occurred for the bytes of the buffer. It does not indicate which bytes contained errors.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

Returns:

The last errors that occurred. The errors are bit masks that are contained in the file **xuartns550.h** and named XUN_ERROR_*.

Note:

None.

Xuint8 XUartNs550_GetModemStatus(XUartNs550 * InstancePtr)

This function gets the modem status from the specified UART. The modem status indicates any changes of the modem signals. This function allows the modem status to be read in a polled mode. The modem status is updated whenever it is read such that reading it twice may not yield the same results.

Parameters:

InstancePtr is a pointer to the XUartNs550 instance to be worked on.

Returns:

The modem status which are bit masks that are contained in the file [xuartns550.h](#) and named XUN_MODEM_*.

Note:

The bit masks used for the modem status are the exact bits of the modem status register with no abstraction.

Xuint16 XUartNs550_GetOptions(XUartNs550 * InstancePtr)

Gets the options for the specified driver instance. The options are implemented as bit masks such that multiple options may be enabled or disabled simultaneously.

Parameters:

InstancePtr is a pointer to the XUartNs550 instance to be worked on.

Returns:

The current options for the UART. The options are bit masks that are contained in the file [xuartns550.h](#) and named XUN_OPTION_*.

Returns:

None.

Xboolean XUartNs550_IsSending(XUartNs550 * InstancePtr)

This function determines if the specified UART is sending data. If the transmitter register is not empty, it is sending data.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

Returns:

A value of XTRUE if the UART is sending data, otherwise XFALSE.

Note:

None.

```
XStatus XUartNs550_SetFifoThreshold( XUartNs550 * InstancePtr,  
                                     Xuint8      TriggerLevel  
                                     )
```

This functions sets the receive FIFO trigger level. The receive trigger level specifies the number of bytes in the receive FIFO that cause a receive data event (interrupt) to be generated. The FIFOs must be enabled to set the trigger level.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

TriggerLevel contains the trigger level to set. Constants which define each trigger level are contained in the file [xuartns550.h](#) and named XUN_FIFO_TRIGGER_*.

Returns:

- o XST_SUCCESS if the trigger level was set
- o XST_UART_CONFIG_ERROR if the trigger level could not be set, either the hardware does not support the FIFOs or FIFOs are not enabled

Note:

None.

```
XStatus XUartNs550_SetOptions( XUartNs550 * InstancePtr,  
                               Xuint16     Options  
                               )
```

Sets the options for the specified driver instance. The options are implemented as bit masks such that multiple options may be enabled or disabled simultaneously.

The `GetOptions` function may be called to retrieve the currently enabled options. The result is ORed in the desired new settings to be enabled and ANDed with the inverse to clear the settings to be disabled. The resulting value is then used as the options for the `SetOption` function call.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

Options contains the options to be set which are bit masks contained in the file [xuartns550.h](#) and named `XUN_OPTION_*`.

Returns:

- `XST_SUCCESS` if the options were set successfully.
- `XST_UART_CONFIG_ERROR` if the options could not be set because the hardware does not support FIFOs

Note:

None.

uartns550/v1_00_b/src/xuartns550_format.c

File Reference

Detailed Description

This file contains the data format functions for the 16450/16550 UART driver. The data format functions allow the baud rate, number of data bits, number of stop bits and parity to be set and retrieved.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	------	---------

1.00b	jhl	03/11/02	Repartitioned driver for smaller files.
1.00b	rmm	05/15/03	Fixed diab compiler warnings.

```
#include "xstatus.h"  
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Functions

XStatus XUartNs550_SetDataFormat (XUartNs550 *InstancePtr, XUartNs550Format *FormatPtr)
void **XUartNs550_GetDataFormat** (XUartNs550 *InstancePtr, XUartNs550Format *FormatPtr)

Function Documentation

```
void XUartNs550_GetDataFormat( XUartNs550 * InstancePtr,  
                               XUartNs550Format * FormatPtr  
                               )
```

Gets the data format for the specified UART. The data format includes the baud rate, number of data bits, number of stop bits, and parity.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FormatPtr is a pointer to a format structure that will contain the data format after this call completes.

Returns:

None.

Note:

None.

```
XStatus XUartNs550_SetDataFormat( XUartNs550 * InstancePtr,  
                                   XUartNs550Format * FormatPtr  
                                   )
```

Sets the data format for the specified UART. The data format includes the baud rate, number of data bits, number of stop bits, and parity. It is the caller's responsibility to ensure that the UART is not sending or receiving data when this function is called.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FormatPtr is a pointer to a format structure containing the data format to be set.

Returns:

- XST_SUCCESS if the data format was successfully set.
- XST_UART_BAUD_ERROR indicates the baud rate could not be set because of the amount of error with the baud rate and the input clock frequency.
- XST_INVALID_PARAM if one of the parameters was not valid.

Note:

The data types in the format type, data bits and parity, are 32 bit fields to prevent a compiler warning that is a bug with the GNU PowerPC compiler. The asserts in this function will cause a warning if these fields are bytes.

The baud rates tested include: 1200, 2400, 4800, 9600, 19200, 38400, 57600 and 115200.

uartns550/v1_00_b/src/xuartns550_intr.c

File Reference

Detailed Description

This file contains the functions that are related to interrupt processing for the 16450/16550 UART driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Functions

```
void XUartNs550_SetHandler (XUartNs550 *InstancePtr, XUartNs550_Handler FuncPtr, void  
    *CallBackRef)  
void XUartNs550_InterruptHandler (XUartNs550 *InstancePtr)
```

Function Documentation

```
void XUartNs550_InterruptHandler( XUartNs550 * InstancePtr)
```

This function is the interrupt handler for the 16450/16550 UART driver. It must be connected to an interrupt system by the user such that it is called when an interrupt for any 16450/16550 UART occurs. This function does not save or restore the processor context such that the user must ensure this occurs.

Parameters:

InstancePtr contains a pointer to the instance of the UART that the interrupt is for.

Returns:

None.

Note:

None.

```
void XUartNs550_SetHandler( XUartNs550 *      InstancePtr,  
                           XUartNs550_Handler FuncPtr,  
                           void *          CallbackRef  
                           )
```

This function sets the handler that will be called when an event (interrupt) occurs in the driver. The purpose of the handler is to allow application specific processing to be performed.

Parameters:

InstancePtr is a pointer to the **XUartNs550** instance to be worked on.

FuncPtr is the pointer to the callback function.

CallbackRef is the upper layer callback reference passed back when the callback function is invoked.

Returns:

None.

Note:

There is no assert on the *CallbackRef* since the driver doesn't know what it is (nor should it)

uartns550/v1_00_b/src/xuartns550_selftest.c

File Reference

Detailed Description

This file contains the self-test functions for the 16450/16550 UART driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xstatus.h"  
#include "xuartns550.h"  
#include "xuartns550_i.h"  
#include "xio.h"
```

Functions

XStatus XUartNs550_SelfTest (XUartNs550 *InstancePtr)

Function Documentation

XStatus XUartNs550_SelfTest(XUartNs550 * *InstancePtr*)

This function runs a self-test on the driver and hardware device. This self test performs a local loopback and verifies data can be sent and received.

The statistics are cleared at the end of the test. The time for this test to execute is proportional to the baud rate that has been set prior to calling this function.

Parameters:

InstancePtr is a pointer to the [XUartNs550](#) instance to be worked on.

Returns:

- XST_SUCCESS if the test was successful
- XST_UART_TEST_FAIL if the test failed looping back the data

Note:

This function can hang if the hardware is not functioning properly.

uartns550/v1_00_b/src/xuartns550_g.c

File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of NS16550 devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	03/11/02	Repartitioned driver for smaller files.

```
#include "xuartns550.h"  
#include "xparameters.h"
```

Variables

[XUartNs550_Config XUartNs550_ConfigTable](#) []

Variable Documentation

[XUartNs550_Config XUartNs550_ConfigTable](#) []

The configuration table for UART 16550/16450 devices in the table. Each device should have an entry in this table.

Generated on 29 May 2003 for Xilinx Device Drivers

common/v1_00_a/src/xutil.h File Reference

Detailed Description

This file contains utility functions such as memory test functions.

Memory test description

A subset of the memory tests can be selected or all of the tests can be run in order. If there is an error detected by a subtest, the test stops and the failure code is returned. Further tests are not run even if all of the tests are selected.

Subtest descriptions:

`XUT_ALLMEMTESTS:`

Runs all of the following tests

`XUT_INCREMENT:`

Incrementing Value Test.

This test starts at 'XUT_MEMTEST_INIT_VALUE' and uses the incrementing value as the test value for memory.

`XUT_WALKONES:`

Walking Ones Test.

This test uses a walking '1' as the test value for memory.

location 1 = 0x00000001

location 2 = 0x00000002

...

`XUT_WALKZEROS:`

Walking Zero's Test.

This test uses the inverse value of the walking ones test as the test value for memory.

location 1 = 0xFFFFFFFFE

location 2 = 0xFFFFFFFFD

...

`XUT_INVERSEADDR:`

Inverse Address Test.

This test uses the inverse of the address of the location under test as the test value for memory.

XUT_FIXEDPATTERN:

Fixed Pattern Test.

This test uses the provided patterns as the test value for memory.

If zero is provided as the pattern the test uses '0xDEADBEEF'.

WARNING

The tests are **DESTRUCTIVE**. Run before any initialized memory spaces have been set up.

The address, Addr, provided to the memory tests is not checked for validity except for the XNULL case. It is possible to provide a code-space pointer for this test to start with and ultimately destroy executable code causing random failures.

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	11/01/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

[Go to the source code of this file.](#)

Memory subtests

```
#define XUT_ALLMEMTESTS  
#define XUT_INCREMENT  
#define XUT_WALKONES  
#define XUT_WALKZEROS
```



```
#define XUT_INVERSEADDR
#define XUT_FIXEDPATTERN
#define XUT_MAXTEST
```

Functions

XStatus XUtil_MemoryTest32 (**Xuint32** *Addr, **Xuint32** Words, **Xuint32** Pattern, **Xuint8** Subtest)

XStatus XUtil_MemoryTest16 (**Xuint16** *Addr, **Xuint32** Words, **Xuint16** Pattern, **Xuint8** Subtest)

XStatus XUtil_MemoryTest8 (**Xuint8** *Addr, **Xuint32** Words, **Xuint8** Pattern, **Xuint8** Subtest)

Define Documentation

```
#define XUT_ALLMEMTESTS
```

See the detailed description of the subtests in the file description.

```
#define XUT_FIXEDPATTERN
```

See the detailed description of the subtests in the file description.

```
#define XUT_INCREMENT
```

See the detailed description of the subtests in the file description.

```
#define XUT_INVERSEADDR
```

See the detailed description of the subtests in the file description.

```
#define XUT_MAXTEST
```

See the detailed description of the subtests in the file description.

```
#define XUT_WALKONES
```

See the detailed description of the subtests in the file description.

```
#define XUT_WALKZEROS
```

See the detailed description of the subtests in the file description.

Function Documentation

```
XStatus XUtil_MemoryTest16( Xuint16 * Addr,  
                             Xuint32  Words,  
                             Xuint16  Pattern,  
                             Xuint8   Subtest  
                             )
```

Performs a destructive 16-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

```
XStatus XUtil_MemoryTest32( Xuint32 * Addr,  
                             Xuint32  Words,  
                             Xuint32  Pattern,  
                             Xuint8   Subtest  
                             )
```

Performs a destructive 32-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundry of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

```
XStatus XUtil_MemoryTest8( Xuint8 * Addr,  
                          Xuint32 Words,  
                          Xuint8 Pattern,  
                          Xuint8 Subtest  
                          )
```

Performs a destructive 8-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundry of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

common/v1_00_a/src/xutil.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xutil.h,v 1.4 2002/03/12 23:56:48 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file common/v1_00_a/src/xutil.h
00026 *
00027 * This file contains utility functions such as memory test functions.
00028 *
00029 * <b>Memory test description</b>
00030 *
00031 * A subset of the memory tests can be selected or all of the tests can be run
00032 * in order.  If there is an error detected by a subtest, the test stops and the
00033 * failure code is returned.  Further tests are not run even if all of the tests
00034 * are selected.
00035 *
00036 * Subtest descriptions:
00037 * <pre>
00038 * XUT_ALLMEMTESTS:
00039 *     Runs all of the following tests
00040 *
00041 * XUT_INCREMENT:
00042 *     Incrementing Value Test.
```

```
00043 *      This test starts at 'XUT_MEMTEST_INIT_VALUE' and uses the incrementing
00044 *      value as the test value for memory.
00045 *
00046 * XUT_WALKONES:
00047 *      Walking Ones Test.
00048 *      This test uses a walking '1' as the test value for memory.
00049 *      location 1 = 0x00000001
00050 *      location 2 = 0x00000002
00051 *      ...
00052 *
00053 * XUT_WALKZEROS:
00054 *      Walking Zero's Test.
00055 *      This test uses the inverse value of the walking ones test
00056 *      as the test value for memory.
00057 *      location 1 = 0xFFFFFFFF
00058 *      location 2 = 0xFFFFFFFFD
00059 *      ...
00060 *
00061 * XUT_INVERSEADDR:
00062 *      Inverse Address Test.
00063 *      This test uses the inverse of the address of the location under test
00064 *      as the test value for memory.
00065 *
00066 * XUT_FIXEDPATTERN:
00067 *      Fixed Pattern Test.
00068 *      This test uses the provided patterns as the test value for memory.
00069 *      If zero is provided as the pattern the test uses '0xDEADBEEF".
00070 * </pre>
00071 *
00072 * <i>WARNING</i>
00073 *
00074 * The tests are <b>DESTRUCTIVE</b>. Run before any initialized memory spaces
00075 * have been set up.
00076 *
00077 * The address, Addr, provided to the memory tests is not checked for
00078 * validity except for the XNULL case. It is possible to provide a code-space
00079 * pointer for this test to start with and ultimately destroy executable code
00080 * causing random failures.
00081 *
00082 * @note
00083 *
00084 * Used for spaces where the address range of the region is smaller than
00085 * the data width. If the memory range is greater than 2 ** width,
00086 * the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a
00087 * boundary of a power of two making it more difficult to detect addressing
00088 * errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same
00089 * problem. Ideally, if large blocks of memory are to be tested, break
00090 * them up into smaller regions of memory to allow the test patterns used
00091 * not to repeat over the region tested.
00092 *
00093 * <pre>
00094 * MODIFICATION HISTORY:
00095 *
```



```
00147 #endif          /* end of protection macro */
```

Generated on 29 May 2003 for Xilinx Device Drivers

common/v1_00_a/src/xutil_memtest.c

File Reference

Detailed Description

Contains the memory test utility functions.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	11/01/01	First release

```
#include "xbasic_types.h"  
#include "xstatus.h"  
#include "xutil.h"
```

Functions

XStatus XUtil_MemoryTest32 (**Xuint32** *Addr, **Xuint32** Words, **Xuint32** Pattern, **Xuint8** Subtest)
XStatus XUtil_MemoryTest16 (**Xuint16** *Addr, **Xuint32** Words, **Xuint16** Pattern, **Xuint8** Subtest)
XStatus XUtil_MemoryTest8 (**Xuint8** *Addr, **Xuint32** Words, **Xuint8** Pattern, **Xuint8** Subtest)

Function Documentation


```
XStatus XUtil_MemoryTest16( Xuint16 * Addr,  
                             Xuint32   Words,  
                             Xuint16   Pattern,  
                             Xuint8    Subtest  
                             )
```

Performs a destructive 16-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2 ** width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

```
XStatus XUtil_MemoryTest32( Xuint32 * Addr,  
                             Xuint32   Words,  
                             Xuint32   Pattern,  
                             Xuint8    Subtest  
                             )
```

Performs a destructive 32-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2^{**} width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundary of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

```
XStatus XUtil_MemoryTest8( Xuint8 * Addr,  
                           Xuint32 Words,  
                           Xuint8 Pattern,  
                           Xuint8 Subtest  
                           )
```

Performs a destructive 8-bit wide memory test.

Parameters:

Addr is a pointer to the region of memory to be tested.

Words is the length of the block.

Pattern is the constant used for the constant pattern test, if 0, 0xDEADBEEF is used.

Subtest is the test selected. See [xutil.h](#) for possible values.

Returns:

- XST_MEMTEST_FAILED is returned for a failure
- XST_SUCCESS is returned for a pass

Note:

Used for spaces where the address range of the region is smaller than the data width. If the memory range is greater than 2^{**} width, the patterns used in XUT_WALKONES and XUT_WALKZEROS will repeat on a boundry of a power of two making it more difficult to detect addressing errors. The XUT_INCREMENT and XUT_INVERSEADDR tests suffer the same problem. Ideally, if large blocks of memory are to be tested, break them up into smaller regions of memory to allow the test patterns used not to repeat over the region tested.

Generated on 29 May 2003 for Xilinx Device Drivers

wdttb/v1_00_b/src/xwdttb_i.h File Reference

Detailed Description

This file contains data which is shared between files and internal to the **XWdtTb** component. It is intended for internal use only.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/06/02	First release
1.00b	rpm	04/26/02	Moved register definitions to xwdttb_1.h

```
#include "xwdttb\_1.h"
```

[Go to the source code of this file.](#)

Variables

[XWdtTb_Config](#) [XWdtTb_ConfigTable](#) []

Variable Documentation

XWdtTb_Config XWdtTb_ConfigTable[]()

This table contains configuration information for each watchdog timer device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

XWdtTb Struct Reference

```
#include <xwdttb.h>
```

Detailed Description

The XWdtTb driver instance data. The user is required to allocate a variable of this type for every watchdog/timer device in the system. A pointer to a variable of this type is then passed to the driver API functions.

The documentation for this struct was generated from the following file:

- [wdttb/v1_00_b/src/xwdttb.h](#)

wdttb/v1_00_b/src/xwdttb.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xwdttb.h,v 1.7 2002/05/02 20:25:26 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file wdttb/v1_00_b/src/xwdttb.h
00026 *
00027 * The Xilinx watchdog timer/timebase component supports the Xilinx watchdog
00028 * timer/timebase hardware. More detailed description of the driver operation
00029 * for each function can be found in the xwdttb.c file.
00030 *
00031 * The Xilinx watchdog timer/timebase driver supports the following features:
00032 * - Polled mode
00033 * - enabling and disabling (if allowed by the hardware) the watchdog timer
00034 * - restarting the watchdog.
00035 * - reading the timebase.
00036 *
00037 * It is the responsibility of the application to provide an interrupt handler
00038 * for the timebase and the watchdog and connect them to the interrupt
00039 * system if interrupt driven mode is desired.
00040 *
00041 * The watchdog timer/timebase component ALWAYS generates an interrupt output
00042 * when:
```

```

00043 *   - the watchdog expires the first time
00044 *   - the timebase rolls over
00045 *
00046 * and ALWAYS generates a reset output when the watchdog timer expires a second
00047 * time. This is not configurable in any way from the software driver's
00048 * perspective.
00049 *
00050 * The Timebase is reset to 0 when the Watchdog Timer is enabled.
00051 *
00052 * If the hardware interrupt signal is not connected, polled mode is the only
00053 * option (using IsWdtExpired) for the watchdog. Reset output will occur for the
00054 * second watchdog timeout regardless. Polled mode for the timebase rollover is
00055 * just reading the contents of the register and seeing if the MSB has
00056 * transitioned from 1 to 0.
00057 *
00058 * The IsWdtExpired function is used for polling the watchdog timer and it is
00059 * also used to check if the watchdog was the cause of the last reset. In this
00060 * situation, call Initialize then call WdtIsExpired. If the result is true
00061 * watchdog timeout caused the last system reset. It is then acceptable to
further
00062 * initialize the component which will reset this bit.
00063 *
00064 * This driver is intended to be RTOS and processor independent. It works with
00065 * physical addresses only. Any needs for dynamic memory management, threads
00066 * or thread mutual exclusion, virtual memory, or cache control must be
00067 * satisfied by the layer above this driver.
00068 *
00069 * <pre>
00070 * MODIFICATION HISTORY:
00071 *
00072 * Ver    Who    Date      Changes
00073 * -----
00074 * 1.00a  ecm    08/16/01  First release
00075 * 1.00b  jhl    02/21/02  Repartitioned driver for smaller files
00076 * 1.00b  rpm    04/26/02  Made LookupConfig public and added XWdtTb_Config
00077 * </pre>
00078 *
00079 * *****/
00080
00081 #ifndef XWDTTB_H /* prevent circular inclusions */
00082 #define XWDTTB_H /* by using protection macros */
00083
00084 /***** Include Files *****/
00085
00086 #include "xbasic_types.h"
00087 #include "xstatus.h"
00088
00089 /***** Constant Definitions *****/
00090
00091
00092 /***** Type Definitions *****/
00093

```



```

00094 /**
00095  * This typedef contains configuration information for the device.
00096  */
00097 typedef struct
00098 {
00099     Xuint16 DeviceId;      /**< Unique ID of device */
00100     Xuint32 BaseAddr;     /**< Base address of the device */
00101 } XWdtTb_Config;
00102
00103
00104 /**
00105  * The XWdtTb driver instance data. The user is required to allocate a
00106  * variable of this type for every watchdog/timer device in the system.
00107  * A pointer to a variable of this type is then passed to the driver API
00108  * functions.
00109  */
00110 typedef struct
00111 {
00112     Xuint32 RegBaseAddress; /* Base address of registers */
00113     Xuint32 IsReady;        /* Device is initialized and ready */
00114     Xuint32 IsStarted;      /* Device watchdog timer is running */
00115 } XWdtTb;
00116
00117 /***** Macros (Inline Functions) Definitions *****/
00118
00119
00120 /***** Function Prototypes *****/
00121
00122 /*
00123  * Required functions in xwdttb.c
00124  */
00125 XStatus XWdtTb_Initialize(XWdtTb *InstancePtr, Xuint16 DeviceId);
00126
00127 void XWdtTb_Start(XWdtTb *InstancePtr);
00128
00129 XStatus XWdtTb_Stop(XWdtTb *InstancePtr);
00130
00131 Xboolean XWdtTb_IsWdtExpired(XWdtTb *InstancePtr);
00132
00133 void XWdtTb_RestartWdt(XWdtTb *InstancePtr);
00134
00135 Xuint32 XWdtTb_GetTbValue(XWdtTb *InstancePtr);
00136
00137 XWdtTb_Config *XWdtTb_LookupConfig(Xuint16 DeviceId);
00138
00139 /*
00140  * Self-test functions in xwdttb_selftest.c
00141  */
00142 XStatus XWdtTb_SelfTest(XWdtTb *InstancePtr);
00143

```

00144 #endif

/* end of protection macro */

Generated on 29 May 2003 for Xilinx Device Drivers

wdttb/v1_00_b/src/xwdttb.h File Reference

Detailed Description

The Xilinx watchdog timer/timebase component supports the Xilinx watchdog timer/timebase hardware. More detailed description of the driver operation for each function can be found in the [xwdttb.c](#) file.

The Xilinx watchdog timer/timebase driver supports the following features:

- Polled mode
- enabling and disabling (if allowed by the hardware) the watchdog timer
- restarting the watchdog.
- reading the timebase.

It is the responsibility of the application to provide an interrupt handler for the timebase and the watchdog and connect them to the interrupt system if interrupt driven mode is desired.

The watchdog timer/timebase component ALWAYS generates an interrupt output when:

- the watchdog expires the first time
- the timebase rolls over

and ALWAYS generates a reset output when the watchdog timer expires a second time. This is not configurable in any way from the software driver's perspective.

The Timebase is reset to 0 when the Watchdog Timer is enabled.

If the hardware interrupt signal is not connected, polled mode is the only option (using `IsWdtExpired`) for the watchdog. Reset output will occur for the second watchdog timeout regardless. Polled mode for the timebase rollover is just reading the contents of the register and seeing if the MSB has transitioned from

1 to 0.

The `IsWdtExpired` function is used for polling the watchdog timer and it is also used to check if the watchdog was the cause of the last reset. In this situation, call `Initialize` then call `WdtIsExpired`. If the result is true watchdog timeout caused the last system reset. It is then acceptable to further initialize the component which will reset this bit.

This driver is intended to be RTOS and processor independent. It works with physical addresses only. Any needs for dynamic memory management, threads or thread mutual exclusion, virtual memory, or cache control must be satisfied by the layer above this driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	02/21/02	Repartitioned driver for smaller files
1.00b	rpm	04/26/02	Made <code>LookupConfig</code> public and added <code>XWdtTb_Config</code>

```
#include "xbasic_types.h"  
#include "xstatus.h"
```

[Go to the source code of this file.](#)

Data Structures

```
struct XWdtTb  
struct XWdtTb\_Config
```

Functions

```
XStatus XWdtTb\_Initialize (XWdtTb *InstancePtr, Xuint16 DeviceId)  
void XWdtTb\_Start (XWdtTb *InstancePtr)  
XStatus XWdtTb\_Stop (XWdtTb *InstancePtr)  
Xboolean XWdtTb\_IsWdtExpired (XWdtTb *InstancePtr)  
void XWdtTb\_RestartWdt (XWdtTb *InstancePtr)
```

Xuint32 XWdtTb_GetTbValue (**XWdtTb** *InstancePtr)

XStatus XWdtTb_SelfTest (**XWdtTb** *InstancePtr)

Function Documentation

Xuint32 XWdtTb_GetTbValue(**XWdtTb** * *InstancePtr*)

Returns the current contents of the timebase.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

The contents of the timebase.

Note:

None.

XStatus XWdtTb_Initialize(**XWdtTb** * *InstancePtr*,
Xuint16 *DeviceId*
)

Initialize a specific watchdog timer/timebase instance/driver. This function must be called before other functions of the driver are called.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XWdtTb** instance. Passing in a device id associates the generic **XWdtTb** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if the configuration for device ID was not found

Note:

None.

Xboolean XWdtTb_IsWdtExpired(**XWdtTb** * *InstancePtr*)

Check if the watchdog timer has expired. This function is used for polled mode and it is also used to check if the last reset was caused by the watchdog timer.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

XTRUE if the watchdog has expired, and XFALSE otherwise.

Note:

None.

void XWdtTb_RestartWdt(**XWdtTb** * *InstancePtr*)

Restart the watchdog timer. An application needs to call this function periodically to keep the timer from asserting the reset output.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

None.

Note:

None.

XStatus XWdtTb_SelfTest(**XWdtTb** * *InstancePtr*)

Run a self-test on the timebase. This test verifies that the timebase is incrementing. The watchdog timer is not tested due to the time required to wait for the watchdog timer to expire. The time consumed by this test is dependant on the system clock and the configuration of the dividers in for the input clock of the timebase.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

- XST_SUCCESS if self-test was successful
- XST_WDTTB_TIMER_FAILED if the timebase is not incrementing

Note:

None.

```
void XWdtTb_Start( XWdtTb * InstancePtr )
```

Start the watchdog timer of the device.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

None.

Note:

The Timebase is reset to 0 when the Watchdog Timer is started. The Timebase is always incrementing

```
XStatus XWdtTb_Stop( XWdtTb * InstancePtr )
```

Disable the watchdog timer.

It is the caller's responsibility to disconnect the interrupt handler of the watchdog timer from the interrupt source, typically an interrupt controller, and disable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

- XST_SUCCESS if the watchdog was stopped successfully
- XST_NO_FEATURE if the watchdog cannot be stopped

Note:

The hardware configuration controls this functionality. If it is not allowed by the hardware the failure will be returned and the timer will continue without interruption.

wdttb/v1_00_b/src/xwdttb.c File Reference

Detailed Description

Contains the required functions of the [XWdtTb](#) driver component. See [xwdttb.h](#) for a description of the driver.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ecm	08/16/01	First release
1.00b	jhl	02/21/02	Repartitioned the driver for smaller files
1.00b	rpm	04/26/02	Made LookupConfig public

```
#include "xbasic_types.h"  
#include "xparameters.h"  
#include "xio.h"  
#include "xwdttb.h"  
#include "xwdttb_i.h"
```

Functions

XStatus [XWdtTb_Initialize](#) ([XWdtTb](#) *InstancePtr, [Xuint16](#) DeviceId)

void [XWdtTb_Start](#) ([XWdtTb](#) *InstancePtr)

XStatus [XWdtTb_Stop](#) ([XWdtTb](#) *InstancePtr)

Xboolean **XWdtTb_IsWdtExpired** (**XWdtTb** *InstancePtr)

void **XWdtTb_RestartWdt** (**XWdtTb** *InstancePtr)

Xuint32 **XWdtTb_GetTbValue** (**XWdtTb** *InstancePtr)

Function Documentation

Xuint32 **XWdtTb_GetTbValue**(**XWdtTb** * *InstancePtr*)

Returns the current contents of the timebase.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

The contents of the timebase.

Note:

None.

XStatus **XWdtTb_Initialize**(**XWdtTb** * *InstancePtr*,
 Xuint16 *DeviceId*
)

Initialize a specific watchdog timer/timebase instance/driver. This function must be called before other functions of the driver are called.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

DeviceId is the unique id of the device controlled by this **XWdtTb** instance. Passing in a device id associates the generic **XWdtTb** instance to a specific device, as chosen by the caller or application developer.

Returns:

- XST_SUCCESS if initialization was successful
- XST_DEVICE_IS_STARTED if the device has already been started
- XST_DEVICE_NOT_FOUND if the configuration for device ID was not found

Note:

None.

Xboolean XWdtTb_IsWdtExpired(XWdtTb * InstancePtr)

Check if the watchdog timer has expired. This function is used for polled mode and it is also used to check if the last reset was caused by the watchdog timer.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

XTRUE if the watchdog has expired, and XFALSE otherwise.

Note:

None.

void XWdtTb_RestartWdt(XWdtTb * InstancePtr)

Restart the watchdog timer. An application needs to call this function periodically to keep the timer from asserting the reset output.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

None.

Note:

None.

void XWdtTb_Start(XWdtTb * InstancePtr)

Start the watchdog timer of the device.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

None.

Note:

The Timebase is reset to 0 when the Watchdog Timer is started. The Timebase is always incrementing

XStatus XWdtTb_Stop(XWdtTb * InstancePtr)

Disable the watchdog timer.

It is the caller's responsibility to disconnect the interrupt handler of the watchdog timer from the interrupt source, typically an interrupt controller, and disable the interrupt in the interrupt controller.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

- XST_SUCCESS if the watchdog was stopped successfully
- XST_NO_FEATURE if the watchdog cannot be stopped

Note:

The hardware configuration controls this functionality. If it is not allowed by the hardware the failure will be returned and the timer will continue without interruption.

wdttb/v1_00_b/src/xwdttb_i.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xwdttb_i.h,v 1.7 2002/05/02 20:25:26 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file wdttb/v1_00_b/src/xwdttb_i.h
00026 *
00027 * This file contains data which is shared between files and internal to the
00028 * XWdtTb component. It is intended for internal use only.
00029 *
00030 * <pre>
00031 * MODIFICATION HISTORY:
00032 *
00033 * Ver    Who    Date      Changes
00034 * -----
00035 * 1.00b  jhl    02/06/02  First release
00036 * 1.00b  rpm    04/26/02  Moved register definitions to xwdttb_1.h
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XWDTTB_I_H /* prevent circular inclusions */
00042 #define XWDTTB_I_H /* by using protection macros */
```

```
00043
00044 /***** Include Files *****/
00045
00046 #include "xwdttb_1.h"
00047
00048 /***** Constant Definitions *****/
00049
00050 /* The following constant controls how long the loop in the self-test
00051  * executes
00052  */
00053 #define XWT_MAX_SELFTEST_LOOP_COUNT 0x00010000
00054
00055 /***** Type Definitions *****/
00056
00057
00058 /***** Macros (Inline Functions) Definitions *****/
00059
00060
00061 /***** Variable Definitions *****/
00062
00063 extern XWdtTb_Config XWdtTb_ConfigTable[];
00064
00065 /***** Function Prototypes *****/
00066
00067
00068 #endif
00069
```

wdttb/v1_00_b/src/xwdttb_l.h File Reference

Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. High-level driver functions are defined in [xwdttb.h](#).

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	----	-----	-----
1.00b	rpm	04/26/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"
```

[Go to the source code of this file.](#)

Defines

```
#define XWdtTb_mGetTimebaseReg(BaseAddress)  
#define XWdtTb_mEnableWdt(BaseAddress)  
#define XWdtTb_mDisableWdt(BaseAddress)  
#define XWdtTb_mRestartWdt(BaseAddress)  
#define XWdtTb_mHasReset(BaseAddress)  
#define XWdtTb_mHasExpired(BaseAddress)
```

Define Documentation

```
#define XWdtTb_mDisableWdt( BaseAddress )
```

Disable the watchdog timer.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XWdtTb_mEnableWdt( BaseAddress )
```

Enable the watchdog timer. Clear previous expirations. The timebase is reset to 0.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

```
#define XWdtTb_mGetTimebaseReg( BaseAddress )
```


Get the contents of the timebase register.

Parameters:

BaseAddress is the base address of the device

Returns:

A 32-bit value representing the timebase.

Note:

None.

```
#define XWdtTb_mHasExpired( BaseAddress )
```

Check to see if the watchdog timer has expired.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the watchdog did expire, XFALSE otherwise.

Note:

None.

```
#define XWdtTb_mHasReset( BaseAddress )
```

Check to see if the last system reset was caused by the timer expiring.

Parameters:

BaseAddress is the base address of the device

Returns:

XTRUE if the watchdog did cause the last reset, XFALSE otherwise.

Note:

None.

```
#define XWdtTb_mRestartWdt( BaseAddress )
```

Restart the watchdog timer.

Parameters:

BaseAddress is the base address of the device

Returns:

None.

Note:

None.

Generated on 29 May 2003 for Xilinx Device Drivers

wdttb/v1_00_b/src/xwdttb_l.h

[Go to the documentation of this file.](#)

```
00001 /* $Id: xwdttb_l.h,v 1.1 2002/05/02 20:21:24 moleres Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /**
00024 *
00025 * @file wdttb/v1_00_b/src/xwdttb_l.h
00026 *
00027 * This header file contains identifiers and low-level driver functions (or
00028 * macros) that can be used to access the device.  High-level driver functions
00029 * are defined in xwdttb.h.
00030 *
00031 * <pre>
00032 * MODIFICATION HISTORY:
00033 *
00034 * Ver    Who    Date      Changes
00035 * -----
00036 * 1.00b rpm  04/26/02  First release
00037 * </pre>
00038 *
00039 *****/
00040
00041 #ifndef XWDTTB_L_H /* prevent circular inclusions */
00042 #define XWDTTB_L_H /* by using protection macros */
```

```

00043
00044 /***** Include Files *****/
00045
00046 #include "xbasic_types.h"
00047 #include "xio.h"
00048
00049 /***** Constant Definitions *****/
00050
00051 /* Offsets of registers from the start of the device */
00052
00053 #define XWT_TWCSR0_OFFSET          0x0
00054 #define XWT_TWCSR1_OFFSET          0x4
00055 #define XWT_TBR_OFFSET            0x8
00056
00057 /* TWCSR0 Control/Status Register 0 bits */
00058
00059 #define XWT_CSR0_WRS_MASK          0x00000008 /* reset status */
00060 #define XWT_CSR0_WDS_MASK          0x00000004 /* timer state */
00061 #define XWT_CSR0_EWDT1_MASK       0x00000002 /* enable bit 1 */
00062
00063 /* TWCSR0/1 Control/Status Register 0/1 bits */
00064
00065 #define XWT_CSRX_EWDT2_MASK        0x00000001 /* enable bit 2 */
00066
00067
00068 /***** Type Definitions *****/
00069
00070
00071 /***** Macros (Inline Functions) Definitions *****/
00072
00073 /*****
00074 *
00075 * Low-level driver macros and functions. The list below provides signatures
00076 * to help the user use the macros.
00077 *
00078 * Xuint32 XWdtTb_mGetTimebaseReg(Xuint32 BaseAddress)
00079 *
00080 * void XWdtTb_mEnableWdt(Xuint32 BaseAddress)
00081 * void XWdtTb_mDisableWdt(Xuint32 BaseAddress)
00082 * void XWdtTb_mRestartWdt(Xuint32 BaseAddress)
00083 *
00084 * Xboolean XWdtTb_mHasReset(Xuint32 BaseAddress)
00085 * Xboolean XWdtTb_mHasExpired(Xuint32 BaseAddress)
00086 *
00087 *****/
00088
00089
00090 /*****
00091 /**
00092 *
00093 * Get the contents of the timebase register.
00094 *

```

```

00095 * @param      BaseAddress is the base address of the device
00096 *
00097 * @return      A 32-bit value representing the timebase.
00098 *
00099 * @note        None.
00100 *
00101 *****/
00102 #define XWdtTb_mGetTimebaseReg(BaseAddress) \
00103         XIo_In32((BaseAddress) + XWT_TBR_OFFSET)
00104
00105
00106 /***/
00107 /**
00108 *
00109 * Enable the watchdog timer. Clear previous expirations. The timebase is
00110 * reset to 0.
00111 *
00112 * @param      BaseAddress is the base address of the device
00113 *
00114 * @return      None.
00115 *
00116 * @note        None.
00117 *
00118 *****/
00119 #define XWdtTb_mEnableWdt(BaseAddress) \
00120 { \
00121     XIo_Out32((BaseAddress) + XWT_TWCSR0_OFFSET, XWT_CSR0_EWDT1_MASK | \
00122             XWT_CSR0_WRS_MASK | XWT_CSR0_WDS_MASK); \
00123     XIo_Out32((BaseAddress) + XWT_TWCSR1_OFFSET, XWT_CSRX_EWDT2_MASK); \
00124 }
00125
00126
00127 /***/
00128 /**
00129 *
00130 * Disable the watchdog timer.
00131 *
00132 * @param      BaseAddress is the base address of the device
00133 *
00134 * @return      None.
00135 *
00136 * @note        None.
00137 *
00138 *****/
00139 #define XWdtTb_mDisableWdt(BaseAddress) \
00140 { \
00141     XIo_Out32((BaseAddress) + XWT_TWCSR0_OFFSET, 0); \
00142     XIo_Out32((BaseAddress) + XWT_TWCSR1_OFFSET, 0); \
00143 }
00144
00145
00146 *****/

```

```

00147 /**
00148 *
00149 * Restart the watchdog timer.
00150 *
00151 * @param   BaseAddress is the base address of the device
00152 *
00153 * @return  None.
00154 *
00155 * @note    None.
00156 *
00157 *****/
00158 #define XWdtTb_mRestartWdt(BaseAddress) \
00159     XIo_Out32((BaseAddress) + XWT_TWCSR0_OFFSET, \
00160             XWT_CSR0_EWDT1_MASK | XWT_CSR0_WRS_MASK |
XWT_CSR0_WDS_MASK)
00161
00162
00163 *****/
00164 /**
00165 *
00166 * Check to see if the last system reset was caused by the timer expiring.
00167 *
00168 * @param   BaseAddress is the base address of the device
00169 *
00170 * @return  XTRUE if the watchdog did cause the last reset, XFALSE otherwise.
00171 *
00172 * @note    None.
00173 *
00174 *****/
00175 #define XWdtTb_mHasReset(BaseAddress) \
00176     (XIo_In32((BaseAddress) + XWT_TWCSR0_OFFSET) &
XWT_CSR0_WRS_MASK)
00177
00178
00179 *****/
00180 /**
00181 *
00182 * Check to see if the watchdog timer has expired.
00183 *
00184 * @param   BaseAddress is the base address of the device
00185 *
00186 * @return  XTRUE if the watchdog did expire, XFALSE otherwise.
00187 *
00188 * @note    None.
00189 *
00190 *****/
00191 #define XWdtTb_mHasExpired(BaseAddress) \
00192     (XIo_In32((BaseAddress) + XWT_TWCSR0_OFFSET) &
XWT_CSR0_WDS_MASK)
00193
00194
00195 *****/ Function Prototypes *****/

```

```
00196
00197
00198 /***** Variable Definitions *****/
00199
00200
00201 #endif
00202
```

XWdtTb_Config Struct Reference

```
#include <xwdttb.h>
```

Detailed Description

This typedef contains configuration information for the device.

Data Fields

[Xuint16 DeviceId](#)

[Xuint32 BaseAddr](#)

Field Documentation

[Xuint32 XWdtTb_Config::BaseAddr](#)

Base address of the device

[Xuint16 XWdtTb_Config::DeviceId](#)

Unique ID of device

The documentation for this struct was generated from the following file:

- `wdttb/v1_00_b/src/xwdttb.h`
-

wdttb/v1_00_b/src/xwdttb_selftest.c File Reference

Detailed Description

Contains diagnostic self-test functions for the [XWdtTb](#) component.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00b	jhl	02/06/02	First release

```
#include "xbasic_types.h"  
#include "xio.h"  
#include "xwdttb.h"  
#include "xwdttb_i.h"
```

Functions

[XStatus XWdtTb_SelfTest](#) ([XWdtTb](#) *InstancePtr)

Function Documentation

[XStatus XWdtTb_SelfTest](#)([XWdtTb](#) * *InstancePtr*)

Run a self-test on the timebase. This test verifies that the timebase is incrementing. The watchdog timer is not tested due to the time required to wait for the watchdog timer to expire. The time consumed by this test is dependant on the system clock and the configuration of the dividers in for the input clock of the timebase.

Parameters:

InstancePtr is a pointer to the **XWdtTb** instance to be worked on.

Returns:

- XST_SUCCESS if self-test was successful
- XST_WDTTB_TIMER_FAILED if the timebase is not incrementing

Note:

None.

wdttb/v1_00_b/src/xwdttb_g.c File Reference

Detailed Description

This file contains a table that specifies the configuration of all watchdog timer devices in the system. Each device should have an entry in the table.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	01/18/01	First release
1.00b	jhl	02/21/02	Repartitioned driver for smaller files

```
#include "xwdttb.h"  
#include "xparameters.h"
```

Variables

[XWdtTb_Config](#) [XWdtTb_ConfigTable](#) [XPAR_XWDTTB_NUM_INSTANCES]

Variable Documentation

[XWdtTb_Config](#) [XWdtTb_ConfigTable](#)[XPAR_XWDTTB_NUM_INSTANCES]

This table contains configuration information for each watchdog timer device in the system.

Generated on 29 May 2003 for Xilinx Device Drivers

channel_fifo/v1_00_a/src/xchannel_fifo_v1_00_a.h

```
00001 /* $Id: xchannel_fifo_v1_00_a.h,v 1.1 2003/05/20 22:20:57 meinelte Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 */
00024 *
00025 * @file channel_fifo/v1_00_a/src/xchannel_fifo_v1_00_a.h
00026 *
00027 * This component is a common component because it's primary purpose is to
00028 * prevent code duplication in drivers. A driver which must handle a channel
00029 * FIFO uses this component rather than directly manipulating a channel FIFO
00030 * directly.
00031 *
00032 * A FIFO is a device which has dual port memory such that one user may be
00033 * inserting data into the FIFO while another is consuming data from the FIFO.
00034 * A channel FIFO is designed for use with channel protocols such as HDLC.
00035 * It is typically only used with devices when DMA and/or Scatter Gather
00036 * is used.
00037 *
00038 * @note
00039 *
00040 * This component has the capability to generate an interrupt when an error
00041 * condition occurs.  It is the user's responsibility to provide the interrupt
00042 * processing to handle the interrupt.  This component provides the ability to
00043 * determine if that interrupt is active, a deadlock condition, and the ability
00044 * to reset the FIFO to clear the condition.  In this condition, the device which
```

```

00045 * is using the FIFO should also be reset to prevent other problems. This error
00046 * condition could occur as a normal part of operation if the size of the FIFO
00047 * is not setup correctly. See the hardware IP specification for more details.
00048 *
00049 * <pre>
00050 * MODIFICATION HISTORY:
00051 *
00052 * Ver    Who    Date      Changes
00053 * -----
00054 * 1.00a ecm 04/18/03  First release
00055 * </pre>
00056 *
00057 *****/
00058 #ifndef XCHANNEL_FIFO_H    /* prevent circular inclusions */
00059 #define XCHANNEL_FIFO_H    /* by using protection macros */
00060
00061 /***** Include Files *****/
00062
00063 #include "xbasic_types.h"
00064 #include "xstatus.h"
00065
00066 /***** Constant Definitions *****/
00067
00068 #define XST_CFIFO_BAD_REG_VALUE        0                /* s/b XStatus
*/
00069 #define XST_CFIFO_LACK_OF_DATA        0                /* s/b XStatus
*/
00070 #define XST_CFIFO_NO_ROOM                0                /* s/b
XStatus */
00071
00072 /* need to fix these */
00073 #define XCF_RESET_REG_OFFSET            0x00
00074 #define XCF_COUNT_STATUS_REG_OFFSET    0x00
00075
00076 /*
00077  * This constant specifies the width of the FIFOs.
00078  */
00079 #define XCF_FIFO_WIDTH_BYTE_COUNT        4
00080
00081 /*
00082  * These constants specify the FIFO type and are mutually exclusive
00083  */
00084 #define XCF_READ_FIFO_TYPE            0                /* a read FIFO */
00085 #define XCF_WRITE_FIFO_TYPE            1                /* a write FIFO */
00086
00087 /*
00088  * These constants define the offsets to each of the registers from the
00089  * register base Addr, each of the constants are a number of bytes
00090  */
00091 #define XCF_TXFIFO_DATA_OFFSET            0x0000UL
00092 #define XCF_TXFIFO_STATUS_OFFSET        0x0000UL
00093 #define XCF_RXFIFO_STATUS_OFFSET        0x2000UL
00094 #define XCF_RXFIFO_DATA_OFFSET            0x2004UL

```

```

00095
00096 /*
00097  * This constant is used with the Reset Register
00098  */
00099 #define XCF_RESET_FIFO_MASK                0x0000000A
00100
00101 /*
00102  * These constants are used with the Occupancy/Vacancy Count Register. This
00103  * register also contains FIFO status
00104  */
00105 #define XCF_COUNT_MASK                      0x01FFFFFF
00106 #define XCF_HALF_EMPTY_FULL_MASK          0x20000000
00107 #define XCF_ALMOST_EMPTY_FULL_MASK       0x40000000
00108 #define XCF_EMPTY_FULL_MASK              0x80000000
00109
00110 /*
00111  * These constants are used with the FIFO write operation. Used to
00112  * indicate the number of valid bytes in the last write operation
00113  * which also indicates the end of packet (EOP).
00114  */
00115 #define XCF_1_BYTES_VALID_OFFSET          0x00000014    /* only 1 byte
valid */
00116 #define XCF_2_BYTES_VALID_OFFSET          0x00000018    /* only 2 bytes
valid */
00117 #define XCF_3_BYTES_VALID_OFFSET          0x0000001C    /* only 3 bytes
valid */
00118 #define XCF_4_BYTES_VALID_OFFSET          0x00000010    /* all 4 bytes
valid */
00119
00120 /***** Type Definitions *****/
00121
00122 /***** Macros (Inline Functions) Definitions *****/
00123
00124 /*****
00125  */
00126 *
00127 * Reset the specified channel FIFO. Resetting a FIFO will cause any data
00128 * contained in the FIFO to be lost.
00129 *
00130 * @param ChannelBaseAddr contains the base address of the FIFO to operate on.
00131 *
00132 * @return
00133 *
00134 * None.
00135 *
00136 * @note
00137 *
00138 * Signature: void XCF_V100A_RESET(XChannelFifoAddrV100a InstancePtr)
00139 *
00140 *****/
00141 #define XCF_V100A_RESET(ChannelBaseAddr) \
00142     XIo_Out32((ChannelBaseAddr) + XCF_RESET_REG_OFFSET, XCF_RESET_FIFO_MASK);
00143

```



```

00144
00145 /*****
00146 /*
00147 *
00148 * Get the occupancy count for a read channel FIFO and the vacancy count for a
00149 * write channel FIFO. These counts indicate the number of 32-bit words
00150 * contained (occupancy) in the FIFO or the number of 32-bit words available
00151 * to write (vacancy) in the FIFO.
00152 *
00153 * @param ChannelBaseAddr contains the base address of the FIFO to operate on.
00154 *
00155 * @return
00156 *
00157 * The occupancy or vacancy count for the specified channel FIFO.
00158 *
00159 * @note
00160 *
00161 * Signature: Xuint32 XCF_V100A_GET_COUNT(XChannelFifoAddrV100a InstancePtr)
00162 *
00163 *****/
00164 #define XCF_V100A_GET_COUNT(ChannelBaseAddr) \
00165     (XIo_In32((ChannelBaseAddr) + XCF_COUNT_STATUS_REG_OFFSET) & \
00166     XCF_COUNT_MASK)
00167
00168
00169 /*****
00170 /*
00171 *
00172 * Determine if the specified channel FIFO is almost empty. Almost empty is
00173 * defined for a read FIFO when there is only one data word in the FIFO.
00174 *
00175 * @param ChannelBaseAddr contains the base address of the FIFO to operate on.
00176 *
00177 * @return
00178 *
00179 * XTRUE if the channel FIFO is almost empty, XFALSE otherwise.
00180 *
00181 * @note
00182 *
00183 * Signature: Xboolean XCF_V100A_IS_ALMOST_EMPTY(XChannelFifoAddrV100a
InstancePtr)
00184 *
00185 *****/
00186 #define XCF_V100A_IS_ALMOST_EMPTY(ChannelBaseAddr) \
00187     (XIo_In32((ChannelBaseAddr) + XCF_COUNT_STATUS_REG_OFFSET) & \
00188     XCF_ALMOST_EMPTY_FULL_MASK)
00189
00190
00191 /*****
00192 /*
00193 *
00194 * Determine if the specified channel FIFO is almost full. Almost full is
00195 * defined for a write FIFO when there is only one available data word in the

```

```

00196 * FIFO.
00197 *
00198 * @param ChannelBaseAddr contains the base address of the FIFO to operate on.
00199 *
00200 * @return
00201 *
00202 * XTRUE if the channel FIFO is almost full, XFALSE otherwise.
00203 *
00204 * @note
00205 *
00206 * Signature: Xboolean XCF_V100A_IS_ALMOST_FULL(XChannelFifoAddrV100a
InstancePtr)
00207 *
00208 *****/
00209 #define XCF_V100A_IS_ALMOST_FULL(ChannelBaseAddr) \
00210     (XIo_In32((ChannelBaseAddr) + XCF_COUNT_STATUS_REG_OFFSET) & \
00211     XCF_ALMOST_EMPTY_FULL_MASK)
00212
00213
00214 *****/
00215 /*
00216 *
00217 * Determine if the specified channel FIFO is empty. This applies only to a
00218 * read FIFO.
00219 *
00220 * @param ChannelBaseAddr contains the base address of the FIFO to operate on.
00221 *
00222 * @return
00223 *
00224 * XTRUE if the channel FIFO is empty, XFALSE otherwise.
00225 *
00226 * @note
00227 *
00228 * Signature: Xboolean XCF_V100A_IS_EMPTY(XChannelFifoAddrV100a InstancePtr)
00229 *
00230 *****/
00231 #define XCF_V100A_IS_EMPTY(ChannelBaseAddr) \
00232     (XIo_In32((ChannelBaseAddr) + XCF_COUNT_STATUS_REG_OFFSET) & \
00233     XCF_EMPTY_FULL_MASK)
00234
00235
00236 *****/
00237 /*
00238 *
00239 * Determine if the specified channel FIFO is full. This applies only to a
00240 * write FIFO.
00241 *
00242 * @param ChannelBaseAddr contains the base address of the FIFO to operate on.
00243 *
00244 * @return
00245 *
00246 * XTRUE if the channel FIFO is full, XFALSE otherwise.
00247 *

```

```

00248 * @note
00249 *
00250 * Signature: Xboolean XCF_V100A_IS_FULL(XChannelFifoAddrV100a InstancePtr)
00251 *
00252 *****/
00253 #define XCF_V100A_IS_FULL(ChannelBaseAddr) \
00254     (XIo_In32((ChannelBaseAddr) + XCF_COUNT_STATUS_REG_OFFSET) & \
00255     XCF_EMPTY_FULL_MASK)
00256
00257
00258 /*
00259 */
00260 *
00261 * Determine if the specified channel FIFO is deadlocked. This condition occurs
00262 * when the FIFO is full and empty at the same time and is caused by a channel
00263 * being written to the FIFO which exceeds the total data capacity of the FIFO.
00264 * It occurs because of the mark/restore features of the channel FIFO which
allow
00265 * retransmission of a channel. The software should reset the FIFO and any
devices
00266 * using the FIFO when this condition occurs.
00267 *
00268 * @param ChannelBaseAddr contains the base address of the FIFO to operate on.
00269 *
00270 * @return
00271 *
00272 * XTRUE if the channel FIFO is deadlocked, XFALSE otherwise.
00273 *
00274 * @note
00275 *
00276 * This component has the capability to generate an interrupt when an error
00277 * condition occurs. It is the user's responsibility to provide the interrupt
00278 * processing to handle the interrupt. This function provides the ability to
00279 * determine if a deadlock condition, and the ability to reset the FIFO to
00280 * clear the condition.
00281 *
00282 * In this condition, the device which is using the FIFO should also be reset
00283 * to prevent other problems. This error condition could occur as a normal part
00284 * of operation if the size of the FIFO is not setup correctly.
00285 *
00286 * Signature: Xboolean XCF_V100A_IS_DEADLOCKED(XChannelFifoAddrV100a
InstancePtr)
00287 *
00288 *****/
00289 #define XCF_V100A_IS_DEADLOCKED(ChannelBaseAddr) \
00290     (XIo_In32((ChannelBaseAddr) + XCF_COUNT_STATUS_REG_OFFSET) & \
00291     XCF_DEADLOCK_MASK)
00292
00293
00294 /****** Function Prototypes *****/
00295
00296 /* Standard functions */
00297

```

```
00298 XStatus XChannelFifoV100a_SelfTest(Xuint32 ChannelBaseAddr,
00299
Xuint32
FifoType);
00300
00301 /* Data functions */
00302
00303 XStatus XChannelFifoV100a_Read(Xuint32 ChannelBaseAddr,
00304
Xuint32 *ReadBufferPtr,
00305
Xuint32 ByteCount,
00306
Xuint32 *StatusRegister);
00307
00308 XStatus XChannelFifoV100a_Write(Xuint32 ChannelBaseAddr,
00309
Xuint32 *WriteBufferPtr,
00310
Xuint32 ByteCount,
00311
Xuint32 *StatusRegister);
00312
00313 #endif
/* end of protection macro */
00314
00315
00316
00317
```

Generated on 29 May 2003 for Xilinx Device Drivers

emc/v1_00_a/src/xemc_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of EMC devices in the system. In addition, there is a lookup function used by the driver to access its configuration information.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	rmm	02/04/02	First release
1.00a	rpm	05/14/02	Made configuration table/lookup public

```
#include "xemc.h"  
#include "xparameters.h"
```

iic/v1_01_c/src/xiic_slave.c File Reference

Detailed Description

Contains slave functions for the **XIic** component. This file is necessary when slave operations, sending and receiving data as a slave on the IIC bus, are desired.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	---	-----	-----
1.01b	jhl	3/26/02	repartitioned the driver
1.01c	ecm	12/05/02	new rev

```
#include "xiic.h"  
#include "xiic_i.h"  
#include "xio.h"
```

intc/v1_00_b/src/xintc_lg.c File Reference

Detailed Description

This file contains the generated configuration data for the low level driver of the interrupt controller.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	jhl	05/02/02	First release

```
#include "xbasic_types.h"  
#include "xintc_1.h"  
#include "xparameters.h"
```

packet_fifo/v2_00_a/src/xpacket_fifo_v2_00_a.h

```
00001 /* $Id: xpacket_fifo_v2_00_a.h,v 1.1 2003/01/08 17:45:20 meinelte Exp $ */
00002 /*****
00003 *
00004 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
00005 *     AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
00006 *     SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,
00007 *     OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
00008 *     APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
00009 *     THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
00010 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
00011 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
00012 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
00013 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
00014 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
00015 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
00016 *     FOR A PARTICULAR PURPOSE.
00017 *
00018 *     (c) Copyright 2002 Xilinx Inc.
00019 *     All rights reserved.
00020 *
00021 *****/
00022 /*****
00023 /*
00024 *
00025 * @file packet_fifo/v2_00_a/src/xpacket_fifo_v2_00_a.h
00026 *
00027 * This component is a common component because it's primary purpose is to
00028 * prevent code duplication in drivers. A driver which must handle a packet
00029 * FIFO uses this component rather than directly manipulating a packet FIFO.
00030 *
00031 * A FIFO is a device which has dual port memory such that one user may be
00032 * inserting data into the FIFO while another is consuming data from the FIFO.
00033 * A packet FIFO is designed for use with packet protocols such as Ethernet and
00034 * ATM. It is typically only used with devices when DMA and/or Scatter Gather
00035 * is used. It differs from a nonpacket FIFO in that it does not provide any
00036 * interrupts for thresholds of the FIFO such that it is less useful without
00037 * DMA.
00038 *
00039 * @note
00040 *
00041 * This component has the capability to generate an interrupt when an error
00042 * condition occurs. It is the user's responsibility to provide the interrupt
00043 * processing to handle the interrupt. This component provides the ability to
00044 * determine if that interrupt is active, a deadlock condition, and the ability
```



```

00045 * to reset the FIFO to clear the condition. In this condition, the device which
00046 * is using the FIFO should also be reset to prevent other problems. This error
00047 * condition could occur as a normal part of operation if the size of the FIFO
00048 * is not setup correctly. See the hardware IP specification for more details.
00049 *
00050 * <pre>
00051 * MODIFICATION HISTORY:
00052 *
00053 * Ver    Who    Date      Changes
00054 * -----
00055 * 2.00a ecm 12/30/02  First release
00056 * </pre>
00057 *
00058 *****/
00059 #ifndef XPACKET_FIFO_V200A_H    /* prevent circular inclusions */
00060 #define XPACKET_FIFO_V200A_H    /* by using protection macros */
00061
00062 /***** Include Files *****/
00063
00064 #include "xbasic_types.h"
00065 #include "xstatus.h"
00066
00067 /***** Constant Definitions *****/
00068
00069 /*
00070  * These constants specify the FIFO type and are mutually exclusive
00071  */
00072 #define XPF_V200A_READ_FIFO_TYPE    0    /* a read FIFO */
00073 #define XPF_V200A_WRITE_FIFO_TYPE    1    /* a write FIFO */
00074
00075 /*
00076  * These constants define the offsets to each of the registers from the
00077  * register base address, each of the constants are a number of bytes
00078  */
00079 #define XPF_V200A_RESET_REG_OFFSET    0UL
00080 #define XPF_V200A_MODULE_INFO_REG_OFFSET    0UL
00081 #define XPF_V200A_COUNT_STATUS_REG_OFFSET    4UL
00082
00083 /*
00084  * This constant is used with the Reset Register
00085  */
00086 #define XPF_V200A_RESET_FIFO_MASK    0x0000000A
00087
00088 /*
00089  * These constants are used with the Occupancy/Vacancy Count Register. This
00090  * register also contains FIFO status
00091  */
00092 #define XPF_V200A_COUNT_MASK    0x00FFFFFF
00093 #define XPF_V200A_DEADLOCK_MASK    0x20000000
00094 #define XPF_V200A_ALMOST_EMPTY_FULL_MASK    0x40000000
00095 #define XPF_V200A_EMPTY_FULL_MASK    0x80000000
00096 #define XPF_V200A_VACANCY_SCALED_MASK    0x10000000

```

```

00097
00098 /*
00099  * This constant is used to mask the Width field
00100  */
00101 #define XPF_V200A_FIFO_WIDTH_MASK                0x0E000000
00102
00103 /*
00104  * These constants are used with the Width field
00105  */
00106 #define XPF_V200A_FIFO_WIDTH_LEGACY_TYPE        0x00000000
00107 #define XPF_V200A_FIFO_WIDTH_8BITS_TYPE        0x02000000
00108 #define XPF_V200A_FIFO_WIDTH_16BITS_TYPE       0x04000000
00109 #define XPF_V200A_FIFO_WIDTH_32BITS_TYPE       0x06000000
00110 #define XPF_V200A_FIFO_WIDTH_64BITS_TYPE       0x08000000
00111 #define XPF_V200A_FIFO_WIDTH_128BITS_TYPE      0x0A000000
00112 #define XPF_V200A_FIFO_WIDTH_256BITS_TYPE      0x0C000000
00113 #define XPF_V200A_FIFO_WIDTH_512BITS_TYPE      0x0E000000
00114
00115
00116 /***** Type Definitions *****/
00117
00118 /*
00119  * The XPacketFifo driver instance data. The driver is required to allocate a
00120  * variable of this type for every packet FIFO in the device.
00121  */
00122 typedef struct
00123 {
00124     Xuint32 RegBaseAddress;    /* Base address of registers */
00125     Xuint32 IsReady;          /* Device is initialized and ready */
00126     Xuint32 DataBaseAddress;   /* Base address of data for FIFOs */
00127 } XPacketFifoV200a;
00128
00129 /***** Macros (Inline Functions) Definitions *****/
00130
00131 /*****
00132  */
00133 *
00134 * Reset the specified packet FIFO. Resetting a FIFO will cause any data
00135 * contained in the FIFO to be lost.
00136 *
00137 * @param InstancePtr contains a pointer to the FIFO to operate on.
00138 *
00139 * @return
00140 *
00141 * None.
00142 *
00143 * @note
00144 *
00145 * Signature: void XPF_V200A_RESET(XPacketFifoV200a *InstancePtr)
00146 *
00147 *****/
00148 #define XPF_V200A_RESET(InstancePtr) \

```

```

00149     XIo_Out32((InstancePtr)->RegBaseAddress + XPF_V200A_RESET_REG_OFFSET,
XPF_V200A_RESET_FIFO_MASK);
00150
00151
00152 /*****
00153 /*
00154 *
00155 * Get the occupancy count for a read packet FIFO and the vacancy count for a
00156 * write packet FIFO. These counts indicate the number of 32-bit words
00157 * contained (occupancy) in the FIFO or the number of 32-bit words available
00158 * to write (vacancy) in the FIFO.
00159 *
00160 * @param InstancePtr contains a pointer to the FIFO to operate on.
00161 *
00162 * @return
00163 *
00164 * The occupancy or vacancy count for the specified packet FIFO.
00165 *
00166 * @note
00167 *
00168 * Signature: Xuint32 XPF_V200A_GET_COUNT(XPacketFifoV200a *InstancePtr)
00169 *
00170 *****/
00171 #define XPF_V200A_GET_COUNT(InstancePtr) \
00172     (XIo_In32((InstancePtr)->RegBaseAddress +
XPF_V200A_COUNT_STATUS_REG_OFFSET) & \
00173     XPF_V200A_COUNT_MASK)
00174
00175
00176 /*****
00177 /*
00178 *
00179 * Determine if the specified packet FIFO is almost empty. Almost empty is
00180 * defined for a read FIFO when there is only one data word in the FIFO.
00181 *
00182 * @param InstancePtr contains a pointer to the FIFO to operate on.
00183 *
00184 * @return
00185 *
00186 * XTRUE if the packet FIFO is almost empty, XFALSE otherwise.
00187 *
00188 * @note
00189 *
00190 * Signature: Xboolean XPF_V200A_IS_ALMOST_EMPTY(XPacketFifoV200a *InstancePtr)
00191 *
00192 *****/
00193 #define XPF_V200A_IS_ALMOST_EMPTY(InstancePtr) \
00194     (XIo_In32((InstancePtr)->RegBaseAddress +
XPF_V200A_COUNT_STATUS_REG_OFFSET) & \
00195     XPF_V200A_ALMOST_EMPTY_FULL_MASK)
00196
00197
00198 /*****

```

```

00199 /*
00200 *
00201 * Determine if the specified packet FIFO is almost full. Almost full is
00202 * defined for a write FIFO when there is only one available data word in the
00203 * FIFO.
00204 *
00205 * @param InstancePtr contains a pointer to the FIFO to operate on.
00206 *
00207 * @return
00208 *
00209 * XTRUE if the packet FIFO is almost full, XFALSE otherwise.
00210 *
00211 * @note
00212 *
00213 * Signature: Xboolean XPF_V200A_IS_ALMOST_FULL(XPacketFifoV200a *InstancePtr)
00214 *
00215 *****/
00216 #define XPF_V200A_IS_ALMOST_FULL(InstancePtr) \
00217     (XIo_In32((InstancePtr)->RegBaseAddress +
XPF_V200A_COUNT_STATUS_REG_OFFSET) & \
00218     XPF_V200A_ALMOST_EMPTY_FULL_MASK)
00219
00220
00221 *****/
00222 /*
00223 *
00224 * Determine if the specified packet FIFO is empty. This applies only to a
00225 * read FIFO.
00226 *
00227 * @param InstancePtr contains a pointer to the FIFO to operate on.
00228 *
00229 * @return
00230 *
00231 * XTRUE if the packet FIFO is empty, XFALSE otherwise.
00232 *
00233 * @note
00234 *
00235 * Signature: Xboolean XPF_V200A_IS_EMPTY(XPacketFifoV200a *InstancePtr)
00236 *
00237 *****/
00238 #define XPF_V200A_IS_EMPTY(InstancePtr) \
00239     (XIo_In32((InstancePtr)->RegBaseAddress +
XPF_V200A_COUNT_STATUS_REG_OFFSET) & \
00240     XPF_V200A_EMPTY_FULL_MASK)
00241
00242
00243 *****/
00244 /*
00245 *
00246 * Determine if the specified packet FIFO is full. This applies only to a
00247 * write FIFO.
00248 *

```

```

00249 * @param InstancePtr contains a pointer to the FIFO to operate on.
00250 *
00251 * @return
00252 *
00253 * XTRUE if the packet FIFO is full, XFALSE otherwise.
00254 *
00255 * @note
00256 *
00257 * Signature: Xboolean XPF_V200A_IS_FULL(XPacketFifoV200a *InstancePtr)
00258 *
00259 *****/
00260 #define XPF_V200A_IS_FULL(InstancePtr) \
00261     (XIo_In32((InstancePtr)->RegBaseAddress +
XPF_V200A_COUNT_STATUS_REG_OFFSET) & \
00262     XPF_V200A_EMPTY_FULL_MASK)
00263
00264
00265 /******/
00266 /*
00267 *
00268 * Determine if the specified packet FIFO is deadlocked. This condition occurs
00269 * when the FIFO is full and empty at the same time and is caused by a packet
00270 * being written to the FIFO which exceeds the total data capacity of the FIFO.
00271 * It occurs because of the mark/restore features of the packet FIFO which allow
00272 * retransmission of a packet. The software should reset the FIFO and any
devices
00273 * using the FIFO when this condition occurs.
00274 *
00275 * @param InstancePtr contains a pointer to the FIFO to operate on.
00276 *
00277 * @return
00278 *
00279 * XTRUE if the packet FIFO is deadlocked, XFALSE otherwise.
00280 *
00281 * @note
00282 *
00283 * This component has the capability to generate an interrupt when an error
00284 * condition occurs. It is the user's responsibility to provide the interrupt
00285 * processing to handle the interrupt. This function provides the ability to
00286 * determine if a deadlock condition, and the ability to reset the FIFO to
00287 * clear the condition.
00288 *
00289 * In this condition, the device which is using the FIFO should also be reset
00290 * to prevent other problems. This error condition could occur as a normal part
00291 * of operation if the size of the FIFO is not setup correctly.
00292 *
00293 * Signature: Xboolean XPF_V200A_IS_DEADLOCKED(XPacketFifoV200a *InstancePtr)
00294 *
00295 *****/
00296 #define XPF_V200A_IS_DEADLOCKED(InstancePtr) \
00297     (XIo_In32((InstancePtr)->RegBaseAddress +
XPF_V200A_COUNT_STATUS_REG_OFFSET) & \
00298     XPF_V200A_DEADLOCK_MASK)

```

```
00299
00300
00301 /***** Function Prototypes *****/
00302
00303 /* Standard functions */
00304
00305 XStatus XPacketFifoV200a_Initialize(XPacketFifoV200a *InstancePtr,
00306                                     Xuint32 RegBaseAddress,
00307                                     Xuint32 DataBaseAddress);
00308 XStatus XPacketFifoV200a_SelfTest(XPacketFifoV200a *InstancePtr, Xuint32
FifoType);
00309
00310 /* Data functions */
00311
00312 XStatus XPacketFifoV200a_Read(XPacketFifoV200a *InstancePtr,
00313                               Xuint8 *ReadBufferPtr,
00314                               Xuint32 ByteCount);
00315 XStatus XPacketFifoV200a_Write(XPacketFifoV200a *InstancePtr,
00316                                Xuint8 *WriteBufferPtr,
00317                                Xuint32 ByteCount);
00318
00319 #endif          /* end of protection macro */
00320
00321
00322
00323
```

pci/v1_00_a/src/xpci_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of PCI devices in the system.

Note:

None.

```
#include "xpci.h"  
#include "xparameters.h"
```

ps2_ref/v1_00_a/src/xps2_g.c File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of PS/2 devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
1.00a	ch	06/18/02	First release

```
#include "xps2.h"
```

```
#include "xparameters.h"
```


tmrctr/v1_00_b/src/xtmrctr_1.c File Reference

Detailed Description

This file contains low-level driver functions that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation.

MODIFICATION HISTORY:

Ver	Who	Date	Changes

1.00b	jhl	04/24/02	First release

```
#include "xbasic_types.h"  
#include "xtmrctr_1.h"
```

touchscreen_ref/v1_00_a/src/xtouchscreen_g.c

File Reference

Detailed Description

This file contains a configuration table that specifies the configuration of touchscreen devices in the system.

MODIFICATION HISTORY:

Ver	Who	Date	Changes
-----	-----	-----	-----
1.00a	ch	08/15/02	First release

```
#include "xtouchscreen.h"  
#include "xparameters.h"
```