CSEE W4840 Embedded System Design Lab 3

Stephen A. Edwards

Due February 19, 2004

Abstract

Use your character generator from last time as part of a TV Typewriter. Write a C program that uses the provided UART to receive and display a stream of ASCII characters. Handle newlines, carriage returns, and scrolling the screen when the cursor reaches the bottom.

1 Introduction

The Xilinx-provided UART can receive characters as well as transmit them; we will use this feature in this lab to turn the XSB-300E into a quasi-useful peripheral: a TV typewriter.

The basic idea is for the board to receive characters through the serial port (transmitted, in this case, by the minicom program) and display them as text on the video screen. Use the character generator from Lab 2 to display the characters, an interrupt routine to receive the character from the UART, and a main loop that copies the characters from the buffer where the interrupt routine has placed them onto the screen.

For printable characters, your program should simply display them and advance the cursor. Non-printing characters, specifically carriage return (control-M) and newline (control-J), should move the cursor. Specifically, a carriage return should move the cursor to the leftmost position *on the same line*, while newline should move the cursor down a line *without affecting its horizontal position*. This behavior is typical for terminals and is left over from Teletype days.

If the cursor tries to go off the bottom of the screen, scroll the characters on the screen up one line to ensure the cursor stays at the bottom.

2 Interrupts

Your TV typewriter will accept characters at 9600 baud, meaning a new character can arrive every

$$\frac{8 \text{ data bits} + 1 \text{ start bit} + 1 \text{ stop bit}}{9600 \text{ bits} \,/\, \text{second}} \approx 1\text{ms}$$

The Microblaze runs at 50 MHz, so this gives us at most

$$\frac{50 \times 10^6 \text{ instructions}}{\text{second}} \cdot 1\text{ms} = 50\,000 \text{ instructions},$$

which is plenty of time to display a single character and move the cursor. However, when it becomes necessary to scroll the screen, we need to copy $640 \times 480 = 307\,200$ bytes (one per pixel). At the very least, it would take at least twice this many cycles (one read and one write) to completely scroll the screen, so we could easily miss at least ten characters every time the screen scrolls if we do not check for incoming characters during a scroll.

One possibility is to modify the scrolling routine to periodically check whether a new character has arrived on the serial port, but this is difficult, obfuscates the code, and might need to be changed if the baud rate, processor speed, screen size, or some other parameter changed.

Interrupts are the preferred solution for handling communication from a peripheral to a processor. Rather than having to repeatedly check the peripheral, the peripheral sends an interrupt to the processor that causes it to stop what it is doing, save its state, and run an interrupt routine that quickly gathers data from the peripheral before returning to the program that was running before the interrupt occurred.

Interrupts are the solution to the scrolling problem: by making it possible for the UART to interrupt the scrolling routine, characters that arrive during a scroll can be saved for after the scrolling finishes. While such an approach does not help us if the program simply cannot keep up with its input (e.g., when the time to process a character is longer than the time between characters), we expect that scrolling happens fairly infrequently.

Interrupt service routines are written to run as quickly as possible and do as little work as possible. While it would be possible to have the interrupt routine itself display characters and scroll the screen, this defeats the purpose of using an interrupt. Instead, the interrupt routine should only check whether a new character has arrived (other sources of interrupts might have inadvertantly invoked the routine), get the new character from the UART, acknowledge the interrupt so the UART is ready for the next character, and enqueue the character into a buffer for the main routine to handle later.

A tricky aspect of having an interrupt routine is that it may be invoked at any time. This is not a problem provided the interrupt routine does not modify anything the main routine is trying to read or write, but at least something needs to be shared since some form of communication must take place.

The danger comes, for example, when the interrupt routine is writing a character into the buffer at the "same time" main routine is reading a character. If the execution of these two operations is not carefully interleaved, the buffer used to communicate between the two systems might become corrupted (e.g.,

appear to have a character in it when it does not).

The usual solution is to disable interrupts while accessing memory locations that are shared with an interrupt routine. This guarantees that the interrupt routine will not modify this memory during this time, albeit at the possible expense of increasing *interrupt latency*—the maximum time between when a peripheral issues an interrupt and when the program acknowledges it.

3 Memory Layout

To give you more headroom in this assignment, we have written a linker script (called, imaginatively, "mylinkscript") that directs most of the program into the off-chip SRAM instead of the small on-chip RAM. The script is fairly complicated since it sends certain parts of the program (specifically, system initialization and the interrupt service routines) into the on-chip RAM.

The first few lines of the linker script describe the two available memory regions. Note that the SRAM region actually starts after the end of the memory used for the framebuffer.

```
MEMORY {
  BRAM : ORIGIN = 0x00000000, LENGTH = 0x01000
  SRAM : ORIGIN = 0x00860000, LENGTH = 0x20000
}
```

The linker is a fairly complex program that collects a set of object files generated by the asssembler (which, in turn, takes its output from the C compiler) into an Executable and Linking Format (ELF) file. This file ultimately contains blocks of data along with the addresses at which they should be located in memory. The linker script instructs the linker how to construct this file.

The new Makefile for this lab downloads two files to the XSB-300E board: a bitstream for the FPGA and a .hex file for the on-board SRAM. The Makefile uses the data2mem program to insert the appropriate segments from the .elf file into the .bit file for the FPGA and objcopy (one of the binutils programs) combined with bin2hex to extract and convert the data for the SRAM.

4 The Assignment

In `~sedwards/4840/lab3.tar.gz`, you will find a skeleton for this lab that includes two C files: main.c and isr.c. Main.c is fairly straightforward: it enables interrupts and registers the handler before going into a (boring) main loop that periodically prints the number of characters the interrupt routine has received and the most recent character.

The interrupt routine in isr.c is more interesting. The `setup_interrupts()` function in main.c installs this function as a handler for interrupts generated by the UART. It checks whether a new character has come in (see the Xilinx UART lite datasheet for documentation) and if so, saves the character and increments a counter.

Implement a circular buffer that communicates from the interrupt routine to the main character routine. Use two pointers: one pointing to where the next character will be written into the buffer and one pointing to the next character to be taken from the buffer. Make the two wrap around and be sure to avoid a buffer overflow condition. Be careful when reading from the buffer—disable interrupts when necessary and do so for as little time as possible.

The main routine should look like

```
for (;;) {
  while (no character in buffer)
    /* do nothing */
  get character from buffer
  display character on screen
  if necessary, scroll the screen
}
```

The interrupt routine should look like

```
if (there is a new character) {
  get the character
  clear the interrupt
  if (the buffer is not full) {
    write the character into the buffer
    advance the buffer pointer
  }
}
```

Use the character generator you wrote for lab 2 to draw the characters. The video display circuitry is included in the skeleton for lab 3.

As usual, show your working TV typewriter to a TA, have him sign a printout of your solution (i.e., all .c files), and hand that it.

Shorter, elegant, readable solutions will score higher, as usual.