CSEE W4840 Embedded System Design Lab 2

Stephen A. Edwards

Due February 10, 2005

Abstract

Write a C program to make the board behave as a simple terminal emulator. Have your program use the provided UART and character display 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

In this lab, you will write C code that integrates two peripherals: a UART and a text-mode video controller. The result will be a simple system that takes characters from the serial port (you can send them to the board by typing in a `minicom` window) and displays them on the flat-panel displays. Such so-called TV typewriters were all the rage in the early 1970s, when it first became possible to build such systems with off-the-shelf hardware.

Use 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 Getting Started

We have provided a skeleton project that exercises the video display and UART. Make sure it compiles, downloads, runs, and displays video before tackling the rest of this lab.

1. Create a directory called, say, `lab2` and `cd` into it.
2. Unpack the project from my home directory:

   ```
   $ tar zxvf ~sedwards/4840/lab2.tar.gz
   ```

3. Invoke `make download`. After a while, this will generate and download a `.bit` file that runs a program (`c_source_files/main.c`) that clears the screen, displays a welcome message, and monitors the most-recently-received character from the serial port.

3 The Video Display

For this assignment, I created a custom OPB peripheral: a text-mode VGA controller. On a standard 640 $\times$ 480 VGA-speed raster (a 60 Hz frame rate), it displays an 80 $\times$ 30 character matrix using an 8 $\times$ 16 font. It stores both the character and the font in on-chip "block" RAMs (2.5K for the characters, 1.5K for the font), which can be both read and written from C.

The 4K of video memory appears as a contiguous region of memory starting at 0xfeff1000. The first 2.5K of this area stores the character information (Actually, only the first 2400 bytes are used for the 80 $\times$ 30 display. Each character consumes a single byte. The remaining 160 are not used by the video system.). This is arranged in the usual manner: 0xfeff1000 stores the character in the upper-left corner, 0xfeff1001 holds the character immediately to its right, and each new row starts 80 bytes after the last one. The remaining 1.5K, starting at 0xfeff1a00, stores the font, arranged as 96 characters, each 16 bytes long. The most significant bit of each byte of the font corresponds to the leftmost pixel of a character and 1's appear as white pixels. The first character of the font (bytes 0xfeff1a00 to 0xfeff1a0f) holds the character numbered 0x20, corresponding to an ASCII space.

The video memory interface is fairly fast, but can only be accessed a byte at a time (i.e., treating the memory as an array of shorts or integers will not work properly).

The font RAM is initialized with a standard ASCII font (an IBM console font from a Linux distribution), the result being that printable strings in C can be copied directly to the screen without translation. Character 0x20 is all black, i.e., a space, and character 0x7f is a solid white box that could be used to display a cursor if desired.

4 Interrupts

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

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

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. When it becomes necessary to scroll the screen, we need to copy 80 × 30 = 2400 bytes (one per character), which can be done quickly enough so that the next input character is not missed.

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.

5 The Assignment

In `~sedwards/4840/lab2.tar.gz`, you will find a skeleton for this lab that includes a main.c file that enables interrupts and registers an interrupt handler routine 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, uart_handler(), is more interesting. The first part of the main() function 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
  }
}
```

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.