Laboratory IV

Universal Asynchronous Receiver Transmitter

Introduction

The purpose of this laboratory is to explore the design of RS-232 communication links and the implementation of the Celoxica Platform Abstraction Layer (PAL). In particular, you will implement the equivalent of a Universal Asynchronous Receiver Transmitter (UART) to interface the FPGA to the RS-232 connector on the RC200E. In doing so, you will be implementing the equivalent of a subset of the PalRS232Port API for the RC200E provided as part of Celoxica's PDK Platform Abstraction Layer (PAL).

For background material on the design of a UART, consult the CS-343 UART Web Page. That web page describes the typical use of a UART as a device controller that acts as an interface between a CPU and serial devices such as modems, mice, etc. As a result, that material includes a discussion about software interfaces to the UART, which are irrelevant for this project.

The RC200E provides a "DB9" connector with pins numbered 2, 3, 7, and 8 connected, respectively, to pins T19, U20, U19, and V20 of the FPGA through a transceiver circuit that translates between RS-232 voltage levels used for serial I/O and the voltage levels used for pin I/O by the FPGA. See page 28 of the RC200 Hardware Manual to see the RS-232 functions of these four pins. The function of the UART is to translate serial data received on the RxD pin and to convert it to parallel data for processing by the application. At the same time, the UART receives parallel data from the application and transmits it serially on the TxD pin. The CTS and RTS pins are used to implement hardware flow control, if enabled.

Project Specifications

Your project is to provide a serial connection between the FPGA and a PC through the RS-232 connector on the RC200E and one of the COM ports on a PC. Writing characters to the COM port on the PC is to cause them to display on the LCD screen of the RC200E. At the same time, typing characters on a keyboard attached to the PS/2 keyboard port of the RC200 is to cause them to display on the screen of the PC. Details of the requirements are given in the sections describing individual Lab Activities.

Lab Activities

1. **Read from Keyboard, Write to LCD Console**
2. **Read Characters From PS/2 Keyboard, Write to RS-232 Port**
3. **Read Data From the RS-232 Port, Write to Console**
4. **Submit a Report of Your Lab Activities**

Read from Keyboard, Write to LCD Console

Create a DK Workspace named "Laboratory IV" for this lab in your "My Projects" directory. Add a new project named "LCD_Console" to this workspace, and add a Handel-C source file named *lcd_console.hcc* to the project. Configure the project for simulation and RC200E configuration in the usual way. For this project you will need to include the *pal_console.hch* and *pal_keyboard.hch* header files in your source code, and you will have to link to the *pal_console.hcl* and *pal_keyboard.hcl* libraries on the Build Configuration Linker tab. These libraries are separate from the *pal_master.hcl* and *pal_rc200e.hcl* libraries; see pages 1-12 of the PAL Reference Manual for more information on the use of these "PAL Core" modules.

Use the keyboard and console PAL APIs to construct a Handel-C program that reads characters from the keyboard and displays them as strings on the console. Each character typed is to cause a line to be displayed on the console in the form, "`c (0xXX)`," where *c* is the character and *XX* is the ASCII code for the character.

Because the PalKeyboard and Pal Console cores are not fully integrated into the PDK at this time, their use is not documented in the PAL Reference Manual. But they are consistent with the rest of the PAL APIs (see pages 14-16 of the PAL Reference Manual, so examining the *pal_console.hch* and *pal_keyboard.hch* header files should be sufficient documentation for you to use them. In addition, the PAL examples supplied by Celoxica include sample programs that illustrate their use if you want to look at them.

The correct clock rate to use for any code that writes on the LCD (such as the PalConsole functions) is 25.175 MHz. You can speed up simulation by using a slower clock rate, but any rate below 25.175 will chop off the right side of the simulated display. The lower the clock rate, the narrower the simulated display.

The easiest way to generate the hexadecimal part of the output is to use a rom lookup table containing the ASCII codes for all 16 possible hex digits:

```
rom unsigned 8 hexTab[ ] = "0123456789ABCDEF";
```

Just use a 4-bit unsigned value as a subscript to get the character to print.

When you simulate this code, you will find that the Pal Virtual Console (see page 9 of the PAL manual) has two tabs. The one you see at first shows you the scan codes for the characters emitted by the keyboard (you'll see the keyboard's initialization output), but the other tab will show you the simulated LCD screen and will give you a place to type in keyboard characters. Note that the simulation is pretty slow, so it takes several seconds for a complete screen refresh to complete. In real time, the screen refreshes 60 times a second. The virtual console shows you the scan line number (0 to 479) in the status bar as the display is updated.

A note on *scan codes:* When you press any key on the keyboard a one or two byte number called the scan code is sent from the keyboard to the PS2 keyboard port. In a PC, software then converts the scan code to a character code (ASCII, or other code depending on what software the operating system uses). Some keys, such as the function keys, don't have ASCII equivalents, and the operating system has to handle them in special ways. On the RC200E, the Pal_Keyboard functions take care of the mapping from scan codes to ASCII characters, but they don't provide the full functionality you might be accustomed to when working with a software interface.

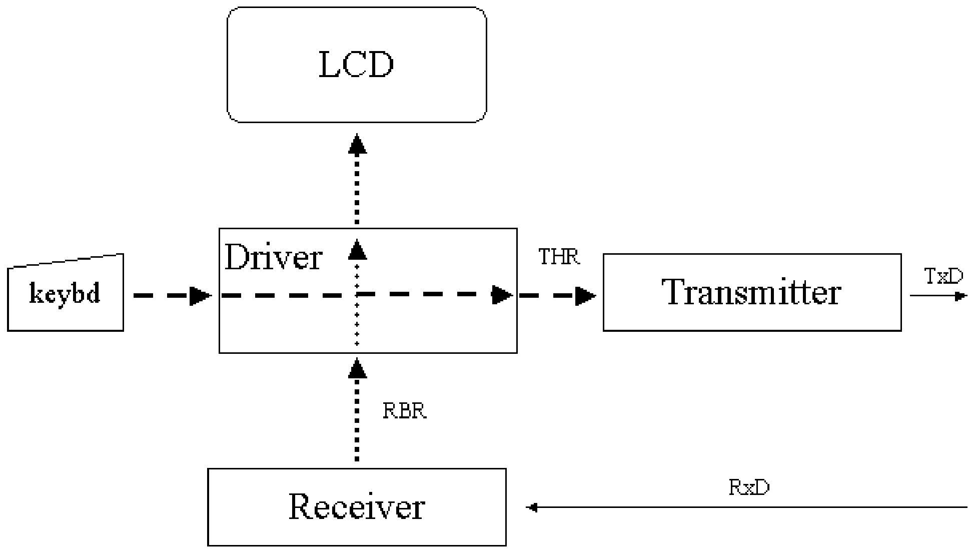
Read Characters From PS/2 Keyboard, Write to RS-232 Port

Create a second project, called UART, and insert a source file, *uart.cc* into it. Configure it for simulation and EDIF in the usual way.

Before proceeding further, consult the UART Web Page for some background information on the role of a UART in a conventional modem-connected-to-a-PC situation.

In designing your project, keep in mind that an RS-232 transmission link has several parameters that have to be set the same way on each side of the link. These parameters include the *baud rate* (the reciprocal of the amount of time used for each bit), the number of *data bits per character* (5, 6, 7, or 8), the nature of the *parity bit* transmitted after the data bits (none, even, odd, mark, or space), the number of stop bits (1, 1.5, or 2), and the type of *flow control* (none, hardware, or Xon/Xoff). You can use the Windows “Hyperterminal” application to send and receive characters over the COM1 port on the PC, and you can set these parameters as you wish from within the Hyperterminal application. You should use 8 data bits, no parity, no flow control, one 1 stop bit, and whatever baud rate you like for now, and use these same parameters values as you write your Handel-C code. But design your project so that all the parameters can be changed easily in the future by using macro definitions for each parameter.

There will be three threads. We’ll call them *driver*, *transmitter*, and *receiver*, and you could use functions with those names, invoked in parallel from a single *main()* if you like. In the diagram below, wide arrows represent 8-bit data paths, and the narrow lines (TxD and RxD) represent 1-bit paths.



You are to implement the Transmitter and its interactions with the driver in this segment of the laboratory. Once you have those working, proceed to the next segment, where you will implement the Receiver and its interactions with the Driver.

The driver is to read characters from the keyboard using the PalKeyboard API, as you did in the previous project. Each time the driver reads a character, it is to pass it to the transmitter through a channel called “THR,” which stands for Transmitter Holding Register. This is a classic producer-consumer design you should be familiar with from your operating systems courses. In a real UART, synchronization between the CPU (the producer) and the UART (the consumer) is managed by means of a semaphore bit, normally called “thrEmpty” or “TE.” And in a typical UART environment, there is normally an option for generating an interrupt request signal to the CPU whenever TE goes true. (That is, whenever the UART is ready to receive more data.) But your Driver is performing all the chores of the producer, with no CPU and no interrupt system to interact with. So we can achieve all the synchronization we need using a simple Handel-C channel to link the driver with the transmitter.

When the transmitter receives a character from the THR, it has to generate a start bit, eight data bits, and a stop bit on the FPGA pin that is connected to the serial output pin of the DB9 connector on the RC200E. The start bit is the “space” voltage, which corresponds to the Handel-C value 0. The stop bit is the “mark” voltage, and the data bits are space for zero and mark for 1. Each bit is 1/baud_rate seconds long.

You might want set up the code to link to the Waveform Analyzer to see the pattern you program sends to the TxD pin, but doing so is optional for this assignment. If your characters are not getting through from the keyboard to the PC, you can use the seven segment displays to show the characters as they move from the driver to the transmitter. If necessary, you can use an oscilloscope to look at the waveform for characters as they leave the DB9 connector.

Read Data From the RS-232 Port, Write to Console

Once the transmitter is working, you can start developing the receiver. This part of the assignment is more complex than the transmitter because it includes handling certain (but not all) error conditions the way a real UART does. The two error conditions your receiver will check for are called *framing* and *data overrun* errors. The third type of error that UARTs handle is parity errors, but doing that is not part of the assignment.

The receiver is to read serial characters from the RxD pin and put them in a global unsigned 8 variable named RBR (Receive Buffer Register). Each time the receiver puts a new character in the RBR, the driver displays it on the LCD console. The driver and receiver share three additional one-bit registers, DO, FE, and RF (rbrFull). Each time the receiver puts a byte in the RBR, it makes RF true, and each time the driver writes a byte from the RBR to the console, it makes RF false. You could almost use a Handel-C semaphore for RF, but that would make detecting data overrun errors more difficult, so I suggest you stay with a simple shared variable for RF.

Any time the receiver has a character ready to put into the RBR and finds that RF is true a *data overrun* has occurred, and the receiver makes the DO bit true. The receiver continues to monitor the RxD pin when DO is true, but never updates the RBR or RF bit while DO is true. A separate thread monitors the state of the DO bit and turns on one of the LEDs whenever it is true, and turns the LED off otherwise. Another thread monitors the state of one of the pushbuttons, and forces DO to be false whenever the button is pressed. Pressing the button is the only way to turn off the DO bit.

One of the jobs of the receiver is to make sure the RxD pin is still in the space state in the middle of the start bit and to make sure it is in the mark state in the middle of the (first) stop bit. If the start bit check fails, it's considered a "glitch on the line" and ignored; the receiver just goes back to looking for the start of the next start bit. However, if the stop bit check fails, a *framing error* has occurred. In this case, the receiver turns on the FE bit and stops updating the RBR and rbrFull bit until it is cleared. Just like the DO bit, separate threads are to monitor the second pushbutton and the FE bit, and pressing the second pushbutton is the only way to clear the FE bit once it is set. Note that the receiver is to determine the values of all the bits by reading the state of the RxD pin in the middle of the time allocated for each of the bits. That is, after detecting the leading edge of the start bit, the receiver should sample RxD one half bit time later, and should determine the values of all the other bits by sampling RxD at one bit-time intervals after the middle of the start bit.

Simulating this project is problematic at best. The PAL virtual console does not simulate the RS-232 port, so you have to provide your own simulated input signal in order to perform a simulation. This is done using the Pattern Generation Language described in Chapter 2 of the Waveform Analyzer Manual. But generating a waveform with the correct timing to simulate receipt of a character through the RS-232 interface at a particular baud rate, parity, etc. is a significant undertaking in itself, and doing so is an optional part of this assignment. If you do do it, be sure to make prominent note of it in your lab report!

Instead of simulation to verify and test your design, you could use an oscilloscope to view the signals arriving on the RxD wire. Use the following code to connect the RxD pin to one of the pins on the expansion header so you can look at it:

```
interface bus_out() expansion_out( unsigned int 1 data = RxD )
  with
  {
    data = { "M2" }
  };
```

You can read from the bus_in interface to the RxD bit on every clock cycle. For example, you could read the state of the RxD pin in parallel with updating the DO and FE LEDs in a single while(1) loop.

Submit a Report of Your Lab Activities

Use a word processor to write a report of your lab activities that follows the format of the Lab Report Guidelines for this course, and email it to me by midnight of the due date.