EECS 373 Winter 2004

Lab 5: Serial Communication

Requirements

Pre-lab: Individual answers to pre-lab questions are due at the beginning of your lab section during the week of **February 9, 2004**. Your group must also have an initial software design completed and entered into SingleStep.

In Lab: In-labs are due by Friday at 4:00pm the week of **February 13, 2004**.

Post Lab: Post labs are due in your lab section the week of **February 16, 2004**.

Value: This lab is worth 5% of your total grade.

Objectives

The purpose of this lab is to:

1. Gain experience with assembly-language functions and procedures.
2. Develop and debug a modular assembly-language program of moderate complexity.
3. Understand the utility of generic programming interfaces (and specifically device drivers).
4. Provide basic experience interfacing a serial communications device (UART).

Overview

In this lab, you will write an assembly-language program that implements a simple text-based calculator with a command-line interface. This program will use a set of generic communications functions (a “device driver” interface) for its input and output. You will also implement a compatible device driver for a simple simulated serial communications device provided with the SingleStep simulator. You will run your program, combined with your device driver, on the simulator. In the lab, we will provide you with a compatible device driver for the MPC823 board serial port. You will combine your calculator program with our device driver to run on the MPC823 boards.

The purpose of a device driver is to provide a generic function-call interface for a particular type or class of device, hiding the details of the specific device from the rest of the system. The logical separation provided by the device-driver interface lets you use the same application on different systems with different devices, as long as each different device has a compatible device driver. Similarly, if you have a new device, writing a single device driver will allow all of the applications that use that driver interface to work with the new device.

A serial communications device is simply a device that communicates data one bit at a time over a single data wire. Serial interfaces are popular because they can connect devices with a minimum number of wires. There are a variety of serial communication protocols. One of the most common is the RS-232C standard, which is used for the serial ports found on most PCs. An RS-232C connection comprises several wires, including two data wires (one in each direction), a ground wire, and some flow-control signals. RS-232C is being displaced in many environments by the Universal Serial Bus (USB), a newer serial protocol that is more complex for designers, but much simpler for end users and offers much higher bandwidth.

Regardless of the protocol, a key function of a serial interface device is to convert data between the *parallel* format used by the processor data bus (i.e., multiple bits at once) and

the serial format used over the interface. Simple serial interface devices are commonly called UARTs, for Universal Asynchronous Receiver/Transmitters. The MPC823 contains a UART, but it is managed by the MPC823's communications coprocessor, and is not directly accessible by the PowerPC core. Accessing the MPC823's serial port is thus somewhat complex. However, the SingleStep environment contains a simple simulated UART. To allow you to gain experience with a simple UART interface, but spare you the complexity of the MPC823, you will write code only for the simple SingleStep device.

Design Specifications

Your software design consists of two parts:

1. A calculator program that accepts lines of text input, processes them, and generates text output. All input and output is done via calls to a generic "device driver" interface.
2. A "device driver" for the simple simulated UART device provided with the SingleStep simulator.

The calculator program

This program communicates with a serial interface, e.g., a user typing at a terminal. (In the lab, you will use the Windows HyperTerminal application to send and receive characters over the PC's serial port to and from the MPC823 board. In the simulator, the SingleStep simulated UART will provide a simulated terminal for input and output.)

Your program should have two components: an input component that processes individual characters to form command lines, and a processing component that processes complete command lines and generates responses.

The input component of your program should read characters from the serial port as they arrive and store them in a buffer array. The following characters must be treated specially:

1. Backspace (ASCII code 0x8): If at the beginning of an input line, do nothing. Otherwise, instead of storing the backspace character in the buffer, back up the buffer pointer so that the previously typed character will be overwritten if a non-backspace character is typed. (That is, make it do the right thing.)
2. Enter (ASCII "carriage return", code 0xD): Put a null character (0x00) in the buffer to terminate the string. Process the buffered string (i.e., the input line) as described below, then reset the buffer pointer to store a new line.

To process an input line, your program should interpret the first word on the line (i.e. all the characters up to the first space or the end of the line) as a command. Your program should recognize three different single-character command words, as described in Table 1.

Table 1: Calculator Commands

Command	Action
"?"	Print a help message
"+"	Interpret the next two words on the command line as integers and print their sum (arg1 + arg2)
"-"	Interpret the next two words on the command line as integers and print their difference (arg1 – arg2)

Commands and operands are separated by one *or more* white spaces. Operands are signed decimal numbers; positive numbers have no prefix, while negative numbers have a "-" (hyphen) prefix.

The calculator output is limited to a 32 bit 2's complement number.

An error in the command line (a bad command, not enough operands, badly formed operands, etc.) should cause an error message to be printed. The error message does not need to be specific (just "error" is OK, though more is better). Bad input should *not* crash your program.

You should print out a prompt at the beginning of each line so that users know the system is ready for input. The prompt should be present when the program starts and after each line is processed. You may choose the prompt character or characters.

The device-driver interface

Table 2 lists the functions of the device-driver interface. The **ser_init** function should be called exactly once at the beginning of the program. This function initializes the serial port to a default state. The other functions, **ser_putchar** and **ser_getchar**, send individual characters (bytes) to and receive characters from the interface, respectively. These are *blocking* calls, i.e., if they can't immediately perform their function they will wait inside the function until they can. For example, if **ser_getchar** is called and no character has been received, it will wait in a loop until a character is received, then return that character.

Table 2: The Device Driver Interface

Function Name	Parameter	Return Value	Purpose
void ser_init()	none	none	Initializes serial port.
void ser_putchar(char c)	ASCII code of output character	none	Writes a character to the serial port.
char ser_getchar()	none	ASCII code of input character	Waits until a character is available from the serial port, then returns it.

The UART device

To run your program on the simulator, you will need to write a device driver that implements the interface described above for the simulated UART device provided in the SingleStep simulator. The simulated UART has seven control registers, five of which you will use for this lab and are listed in Table 3. Each register is one byte wide and should only be accessed using byte loads and stores.

Table 3: UART Registers

Offset	Register	Description
0	ENABLE	Enable Transmitter/Receiver
3	RSTAT	Receiver Status
4	RDATA	Receiver Data
5	TSTAT	Transmitter Status

Table 3: UART Registers

6	TDATA	Transmitter Data

The “offset” given in the table is the offset in bytes from the starting address of the UART’s device registers. You will place the UART at address 0xFE000000, so the ENABLE register will be at 0xFE000000, the RSTAT register will be at 0xFE000003, etc.

The detailed specification of each register (taken from the SingleStep manual) follows. Note that the bit-numbering convention differs from the PowerPC standard. For this lab, do not worry about handling overrun errors (which occur when the UART receives characters faster than your device driver can read them out or when you write characters to the UART faster than it can transmit them). You may safely ignore all bits that deal with overrun errors.

ENABLE (Register #0)

MSB 7	6	5	4	3	2	1	LSB 0
RE	TE	-	-	-	-	-	ER

This read/write register contains the receiver and transmitter enable control bits as well as the clear overrun error bit.

Setting RE (bit #7) enables the receiver.

Setting TE (bit #6) enables the transmitter.

Writing a 1 to ER (bit #0) clears the transmitter and receiver overrun error flags. ER is a write-only bit which always reads as 0.

It is only necessary to set RE and TE once. It is not necessary to set RE and TE for every reception or transmission.

RSTAT (register #3)

MSB 7	6	5	4	3	2	1	LSB 0
RF	-	-	-	-	-	-	RO

This read-only register contains the receiver’s status.

If RF (bit #7) is set, the receiver’s data buffer is full (i.e., a character has been received). You may obtain the character the receiver just received by reading the RDATA register.

You do not have to clear the status register RF, the simulator will clear the register after a read from RDATA

If RO (bit #0) is set, a receiver-overrun error has occurred, indicating that the receiver has dropped at least one incoming character. You may clear the RO status bit by writing a 1 to the ER bit in the CONTROL register.

RDATA (register #4)

This read-only register is the receiver’s data buffer. If the receiver is full (i.e., the RF bit in the RSTAT register is set), this register contains the last character received by the receiver. Reading RDATA clears the RSTAT register’s RF status bit until the next character is received.

TSTAT (register #5)

MSB 7	6	5	4	3	2	1	LSB 0
TE	-	-	-	-	-	-	TO

This read-only register contains the transmitter's status.

If TE (bit #7) is set, the transmitter data buffer is empty. If the transmitter is enabled, this indicates that the transmitter is ready to transmit another character. TE is a status bit and is only read only bit. It is controlled by the simulator.

If TO (bit #0) is set, a transmitter-overrun error has occurred, indicating that the transmitter has dropped at least one outgoing character. You may clear the TO status bit by writing a 1 to the ER bit in the CONTROL register.

TDATA (register #6)

This write-only register is the transmitter's data buffer. If the transmitter is ready to transmit (the TSTAT register's TE bit is set), the UART will transmit any character written to this register. Writing to TDATA always clears the transmitter-empty status bit (TE) in the TSTAT register; the TE bit will become set when the transmitter is ready for the next character.

Design Notes and Hint

1. You must follow the ABI discussed in class and available on the course web page, including rules on register usage, stack alignment, stack frame format, etc. The device driver supplied for the lab board will use the ABI conventions; your code may not work correctly if you do not follow the same conventions.
2. See the accompanying lab document on ABI stack and register usage.
3. When writing code in assembly language, it is usually a good idea to write out your algorithm in C or pseudo-code, then translate the algorithm into assembly language.
4. Every function should start with a comment giving a C-style function prototype, a description of the input parameters, what the function does, the return value, and the register assignments. The function should also have a well-defined prolog, body and epilog. For example:

```
# int add(int x, int y);
# Adds two signed integers. Returns sum.
#
# Register usage:
#           r3: x (addend)
#           r4: y (augend)

#prolog
            stwu r1, -8(r1) #update stack pointer
            mflr r0         #save LR
            stw r0, 12(r1)

#body
            add r3, r3, r4  # retval <- x + y

#epilog
            lwz r0, 12(r1)  #reload old LR value
            mtlr r0
```

```
                addi r1, r1, 8   #release stack frame
                blr              #return
```

5. The **.skip *n*** assembler directive reserves *n* bytes of memory and initializes them to zero. Use this directive to allocate space for your input line buffer and stack. For example,

   ```
   buffer: .skip 100
   ```

   will allocate 100 bytes and set the symbol "buffer" to the address of the first byte. Similarly,

   ```
              .skip  1000
              .align 2
   init_stack:.skip  4
   ```

   allocates 1004 bytes for your stack, and sets the "init_stack" label to point to the last 4 bytes (at the highest addresses). You can then set up your stack by initializing r1 with the value of "init_stack". Your stack will grow downward into the 1000 bytes you reserved with the first **.skip**.

6. The device driver routines and the terminal program should be in separate **.s** files. Run each file through the assembler independently to generate two **.o** files. To get a complete executable, you will link the two files together by specifying both on the linker command line. For example, if you start with the files **terminal.s** and **simdriver.s**, and assemble these to **terminal.o** and **simdriver.o**, you could link them like this:

   ldppcl -dn -A lab5-lnk.txt -o lab5.elf terminal.o simdriver.o

   where lab5-lnk.txt is identical to lab2 and lab 3lnk.txt files.

7. To run your program on the simulator, you will need to tell the simulator to configure the simulated UART. You will do this using the **mem** command, which manages the simulated memory space. First, you need to open a SingleStep command window by clicking the command-window button or selecting Command from the Window pull-down menu. Then type the command **mem** with no arguments. This will display all of the current memory regions and their attributes. Verify that there are no memory regions that contain the address **0xFE000000**. If there is a region at that address, delete it by typing **mem -d 0xfe000000**.

   Finally, type the command

   mem 0xfe000000 periph=j:\perm\eecs373\lab5\xuart nooverrun

   to load the simulated UART and map its registers starting at **0xFE000000**. (The **nooverrun** option tells it not to generate overrun errors.) Load your .elf file (your terminal program and simulated driver linked together) **after** invoking the simulated UART using the line above.

8. The simulated UART will *not* run on the SingleStep demo version from the 373 web page; it will only run on the machines in the 373 lab or in the CAEN labs. You can debug individual functions on the demo version by initializing registers and/or memory locations manually, but you will need to go to a CAEN lab or the 373 lab to debug your complete program with the simulated UART.

Pre-lab Assignment

(Note: In the following problems, the term "integer" includes both positive and negative numbers.)

1. Write a function that takes a string pointer (i.e., the starting address of a null terminated array of characters) containing ASCII digits and returns the integer represented by the string. Write the function in C and assembler. The value may be negative, so represent the converted number in 2's complement form. The size of the number is limited to 32 bits.

2. Write a function in C and assembler that takes an integer value and generates a string of ASCII digits. The function should take two parameters, an integer value and the address of a character array in which to put the string. There is no return value.

3. Write the ser_putchar and ser_getchar driver functions for the simulated UART. Driver function is defined by table 2. The drivers are constructed using the simulation registers described above.

4. Write out a high-level description of how your terminal interface program will work. Decompose your program into functions, possibly including but definitely not limited to the functions from the previous three questions.

5. You may want to use the MetaWare C compiler to develop or verify your assembly code.

   See the document accompaning the lab document on the use of the compiler. The complier is available on the lab workstations, C/sds75/hcppc/bin. The compiler is **not** available on CAEN machines. You can download a demo version of SingleStep from the course web page. After installing, the compiler is found under C/sds74/hcppc/bin.

In-lab Procedure

Demonstration 1.1: Demonstrate your program running the simulated UART to your lab instructor. Your code must use a asciitoint C function. Demonstrate to your instructor that you can step through the C source for the asciitoint function.

1. Link the device driver **adsinit.o** located in j:/perm/eecs373/lab5 directory in place of your **simdriver.o** device driver. The adsinit.o device driver allows you to drive the serial port on the MPC8xx FADS boards. The link command line should appear as follows:

   ldppcl -dn -A lab5-lnk.txt -o lab5.elf terminal.o adsinit.o

2. Start SingleStep On-Chip and the download the elf file.

3. Start the terminal program on the PC by clicking on Start>Programs>On-Chip Terminal>MPC823.ht or desk top icon. This will evoke a HyperTerminal program configured to communicate with the MPC8xx FADS serial port.

4. Run your program. Ideally, typing characters in HyperTerminal should be indistinguishable from typing them in the UART window of the simulator. If not, debug your program.

5. Connect the oscilloscope probe to the transmit-data (pin 2) lines of the RS-232 connector. A break out connector will be provided to attach the scope probe to the RS-232 connector pin. As you type characters, observe the signal on the oscilloscope. What are the high and low voltages of an RS-232 signal? What is the minimum pulse width that you observed? Draw the waveforms you observed on the oscilloscope? This information is required to answer post lab questions.

6. Set the time and voltage scales on the oscilloscope to capture one character. Can you discern the relationship between the signal voltages and the transmitted data value?

(Hint: send a character with a known ASCII value (e.g., ‘A’) and copy the waveform. Then send the next character (e.g., ‘B’) and see how the waveform differs.) This information is required to answer post lab questions.

Demonstration 1.2: Demonstrate your working program to the Lab Instructor. Turn your signed listing in with your report.

Post-lab

1. Give a brief but thorough overview of your program’s structure. Include descriptions of the functions you implemented and their interfaces.
2. Include a well-commented listing of your program. Comments should include register usage, descriptions of all symbols, and explanation of all derived expressions.
3. Provide hardcopy of the assembler generated by the C compiler for your asciitoint function and answer the following questions.

   To obtain a copy of the assembler code, you may have to copy the code from the SingleStep debugger window. To copy, select code with the cursor and copy under the Edit menu.

- What instruction is used to store the LR to the stack?
- Is the LR restored from the stack? Why or why not?
- What register is used to pass the string pointer to the function?
- What register is used to return the int value from the function?
- Identify the register used in the function as the string pointer and label with the number **1** on your hard copy.
- Identify the compare and branch combination that determines the sign of the string number and label with the number **2** on your hardcopy.
- Identify the base ten multiply operation and label with the number **3** on your hardcopy.

4. What attributes of the RS-232 protocol were you able to deduce from your observations? Identify the pulse widths and amplitudes on the waveforms you drew in the lab.
5. Discuss the use of device drivers. What advantage did you gain from using device drivers rather than programming directly to the simulated UART’s registers?

Lab 5 Demonstration Sheet

Print this page and present it to your lab instructor when demonstrating the various lab sections. Turn this sheet in with your post lab or when your in lab demonstration is due. You are required to turn in only one demonstration sheet per group.

List Partners Names

_____

_____

_____

D1.1 Demonstrate your program running the simulated UART to your lab instructor.

Lab instructors initials:

D1.2 Demonstrate your working program to the Lab Instructor. Have your listing file ready to be signed. Turn your signed listing in with your report.

Lab instructors initials: