

Mechatronics Laboratory Assignment 2

Serial Communication

DSP Time-Keeping, Visual Basic, LCD Screens, and Wireless Networks

Recommended Due Date: By your lab time the week of February 9th

Possible Points: If checked off before your lab time the week of Feb. 16th ... 10 points

If checked off after your lab time the week of Feb. 16th and before your lab time the week of Feb. 23rd ... 7 points

If checked off after your lab time the week of Feb. 23rd and before your lab time the week of Mar. 2nd ... 5 points

If work not finished before your lab time the week of Mar. 2nd ... 0 points

Goals for this Lab Assignment:

1. Introduce the VB environment for PC-based monitoring of the robot
2. Learn to interface to the LCD screen on the DSP
3. Learn about DSP BIOS tasks by creating an interface to the serial wireless modem attached to the DSP

DSP/BIOS Objects Used:

PRD, TSK, SEM_pend, SEM_post, SWI, SWI_post

Daughter Card Library Functions Used:

Init_UART1and2, checkfornewcharUART1, LCDPrintLine1, LCDPrintLine2, Wireless Modem, SmallSprintf, strlen

Prelab: In CCS select the **Help**→**Contents** Menu item. In CCS Help select the item “DSP/BIOS 5.32”. If that item is not displayed the file can be found at C:\CCStudio_v3.3\bios_5_32_03\packages\ti\bios\help\doc\c6xbios.hlp or <http://coeosl.ece.uiuc.edu/ge423/datasheets/c6xbios.hlp> .

1. Read the section titled “What is DSP/BIOS?”. You don’t have to understand all the details; just skim.
2. In that same “DSP/BIOS 5.32” section select “DSP/BIOS 5.32 Tutorial”. Read the “What is DSP/BIOS?” section and the section on “Choosing Thread Types”. Also found at C:\CCStudio_v3.3\bios_5_32_03\packages\ti\bios\help\doc\cc_c6xat.hlp or http://coeosl.ece.uiuc.edu/ge423/datasheets/cc_c6xat.hlp .
3. Read your C book for information on how to use the ANSI C “printf” function. You should bring your C book to lab.
4. Each lab builds upon the result of the previous lab. You will have to recall many of the things you learned in previous labs, such as creating a PRD object, writing a function, creating a new project, etc.
5. Read chapters 1-6 in the VB book “Teach Yourself VB 6 in 24 hours”. A number of copies of this book are found in the lab.

Laboratory Exercise 1: A simple VB interface

In this portion of the lab, you will create a simple Visual Basic interface from scratch to communicate with your robot.

1. Open up VB, create a new project, and immediately save it to a local directory. Our suggestion is **c:\yourloginname\lab2\VB_Robot_interface**.
2. Open up the form view.
 - a. Add 2 text boxes

- b. Add a MSComm (Microsoft Comm Control 6.0) object . Ask your TA if you have trouble finding this item.
 - i. To add this object to your project go to the menu Project->Components.
 - ii. Under the Controls Tab, scroll down and check the Microsoft Comm Control 6.0 item and then Apply.
 - iii. Now the MSComm object is added to your toolbar and you can add it to your project by clicking on it and drawing a square in an open space on the form.
 - c. Add a button
3. Using the properties option for each of the objects, rename the first text box to TXMessage, and the second text box to RXMessage.
 4. Single-click on the modem object in the form view. Leave the modem settings to the defaults, except for the following:
 - a. Name: SerialCom1
 - b. Comm Port: 1
 - c. Settings: 57600,n,8,1 (What do these setting specify?)
 - d. RThreshold: 1 (What does this indicate to the MSComm object?)
 5. In the form view, single click on your button. Change the name to “SendMessage”, and change the caption to “Send Message”. Also change the button’s property “Default” to True. When the “Default” setting of a button is set to true, the button’s code is run when you press the “Enter” key of the keyboard.

6. Double Click on the SendMessage button and insert the following code:

```
SerialCom1.Output = Chr(253) & TXMessage.Text & Chr(255)
```

7. Double click on empty space in your form window. Add the following code to the auto-generated function

Form_Load:

```
SerialCom1.PortOpen = True 'Enable the serial port when form is loaded
```

8. Arrow to the very top of your form code above all the other functions. Add the following global variables:

```
Option Explicit 'Force you to declare all your variables
Dim strNextString As String
```

9. Go back to the form view. Double click on the MSComm object. Add the following code (cut and paste from the lab):

```
Private Sub SerialCom1_OnComm()

    Dim intStart As Integer 'the index of StartChar
    Dim intEnd As Integer 'the index of EndChar

    If SerialCom1.CommEvent = 2 Then ' Event messages... only care about event 2
    (receive)
        strNextString = strNextString & SerialCom1.Input ' Append new com data
        intStart = InStr(strNextString, Chr(253)) 'Find the index of StartChar
        If (intStart > 0) Then
            strNextString = Mid(strNextString, intStart)
            intEnd = InStr(strNextString, Chr(255)) 'Find the index of EndChar
            If (intEnd > 1) Then 'If Start&End exist and Start comes before End
                RXMessage.Text = Mid(strNextString,2, intEnd-2) 'Output the full message
                strNextString = Mid(strNextString, intEnd + 1) 'Trim off the message
                from our input stream
```

```

        End If
    End If
End If
End Sub

```

Note that this code is handling error checking for you using the 253 and 255 bytes as stop and start characters.

This means you cannot send either of these bytes in your message without disrupting the communication protocol.

You can change the communication protocol later if you wish, but we found this to work very well when sending only ASCII characters.

10. Have your TA show you how to turn on your bench's wireless modem.

11. Open Code Composer Studio and load and run the following compiled file:

V:\mechlab\lab2\modem_test\modem_test.out. This code will be used to verify your VB routine is working. It receives messages from the PC via the wireless modem, and then 'echoes' them back to the PC. You will know it is running because the message, "Waiting for PC data", will be displayed on the LCD screen.

12. Run your VB application, and type messages into the first text box (TXMessage). They will be sent to the robot, who will send them back to be received at the PC and displayed in the second text box (RXMessage).

With the VB code working, we now develop code to run on the DSP.

Laboratory Exercise 2: Printing to the LCD screen

There are two types of low-cost LCD screens: parallel and serial. The parallel LCD screens cost about \$10 for a two-line, 20 characters-per-line screen with a backlight, while the serial equivalent costs about \$60. The I2C LCD, while much more expensive, is much simpler to use and is the type installed on the robot. The first task of this lab is to print the words "Hello" and "World" on lines 1 and 2 of the LCD screen. To assist you, we have written two functions, LCDPrintLine1 and LCDPrintLine2, to make the DSP that will print to the LCD screen. For instance

```

LCDPrintfLine1("Robot ready");
LCDPrintfLine2("Let's Roll!");

```

Create a new project and add code at the end of `main()` to send some start-up message like "TI DSPs Rock". You may also find that the contrast of your LCD screen is either too dark or too light. You can adjust the contrast by changing the value passed to the `Init_LCD` function in `main()`. Once you have finished your code, demonstrate your working LCD printing to the TA.

Laboratory Exercise 3: Creating a Task and using Semaphores with the DSP BIOS to manage a serial UART

As you learn DSP/BIOS, you will find that there are often multiple ways to perform the same operation. Which way is correct is a matter of preference in many cases, but we will find that there are specific cases where one DSP/BIOS object is better suited for an operation than the others. In the following exercise, you will be introduced to the task object. The task object is particularly suitable for waiting for events that are not critical, or for performing tasks that do not require critical timing. The differences and advantages between all the primary event-handling objects, i.e. the period functions (from lab 1), task functions (this lab), software interrupts (this lab and lab 8), and hardware interrupts (lab 4), will be discussed in lecture.

We have built the robots with two serial UARTs: one is used for the wireless modem and the other will be used for debug purposes when we get to lab 8 using the digital camera. RS-232 style serial ports are relatively slow compared to the speed of the DSP, so the DSP should not constantly wait for data to arrive on the port. Therefore, we need to have a method to save the messages as they arrive, and then read them once the message is complete and your code has free time to check them. In this section you will write two task functions: one called **UART1ReceiveCharTask** will run each time a character is received on the serial port and will save the character to a data array until a stop character is received. At that point, this task will flag the second task, **NetReceiveMessageTask**, with a semaphore indicating a full message has been received. **NetReceiveMessageTask** will wait for this semaphore, then print the message to the LCD screen and ‘echo’ the message back to the PC via the wireless modem.

Switch seats so your partner can code the following:

1. Create a new project with Project Creator.
2. Add a PRD function that is called every millisecond whose purpose is to increment an integer keeping track of time in units of milliseconds. In other words create a global int variable and increment the variable by one inside your PRD function. Use the **LCDPrintLine1** function to print your integer time value to the LCD screen line 1 every 100ms. As a hint, the mod operator, %, will help you to perform your LCD printing. For example, the if statement:

```
if (0==(timeint%33)) {
    do_something();
}
```

only executes if the integer `timeint` is evenly divisible by 33.

See notes 1-3 at the end of the lab.

3. In your DSP BIOS configuration file, check that the semaphore “**SEM_UART1MessageReady**” has been created. You will be using this semaphore in your code, so you need to tell the compiler that this semaphore exists externally. At the top of your source file (after the #includes) add:

```
extern far SEM_Obj SEM_UART1MessageReady;
```

See Note 4 at the end of the lab.

4. Using the code below, we next create a task that will fill the array **UART1MessageArray** as data is sent to the DSP. To save you time – and because the code is complex – we have provided the following complete code for this task. This function should be copied over the shell function in the file “user_UARTFuncs.c”. The reason for using this separate file for this function is that in future labs you will be using the same code and can just copy this file into your new project.

```
void UART1ReceiveCharTask(void) {
    UART1sensordatatimeouterror = 0; // Initialize timeout error count
    UART1transmissionerror = 0; // Initialize transmission error count
    while(1) { // Loop forever
        // Wait forever for new character
        if ( checkfornewcharUART1(&UART1receivechar, SYS_FOREVER) ) {
            if (!UART1beginnewdata) { // Only true if have not yet begun a message
                if (253 == (unsigned char)UART1receivechar) { // Check for start char
                    if (0 == SEM_count(&SEM_UART1MessageReady)) { //Last data sent was used if = 0
                        UART1datacollect = 0; // amount of data collected in message set to 0
                    }
                }
            }
        }
    }
}
```


back to the wireless modem using the function **WirelessSend**. (Note that there is a C function **strlen** that returns the length of a null terminated character array.) You will have to use the functions given in **max3100uart.c** as well as the **SEM_pend** function, i.e.

```
SEM_pend(&SEM_UART1MessageReady, SYS_FOREVER) ;
```

This **SEM_pend** command will literally wait forever for the semaphore **SEM_UART1MessageReady** to be a non-zero value. The code will ‘go to sleep’ at this line until the DSP/BIOS kernel wakes it up. To ‘wake up’, the task **UART1ReceiveCharTask** must first increment the semaphore using the **SEM_post** function. When the kernel detects this non-zero semaphore value, it will wake up the task, decrement the semaphore, and return from the **SEM_pend** function so that the remaining code will be executed.

8. Note that any variables that you use in your main C file that were defined in `user_UARTFuncs.c` will have to be declared as “extern” in your main C file.
9. Build, download and run your DSP application. Run your VB code developed earlier. You should see the LCD print the messages you type at the PC along with the time. Demo this to your TA.
10. The **WirelessSend** function allows you to directly print to the wireless modem. This function automatically adds the start and stop bytes 253 and 255 at the beginning and end of the character message sent over the wireless channel. Use the **WirelessSend** command again to return data to the VB application. This time instead of echoing the received data, send the global integer time value back to the PC. (Note that you will need to use the given function **SmallSprintf** to convert the integer to a string character array.) Run the DSP and VB code to verify proper communication.
11. Demonstrate your working messaging to the TA to complete this part of the lab.

Laboratory Exercise 4: Creating a SWI and changing priorities.

In this section we will take a quick look at software interrupts (SWIs) and how priority assignments can change the performance of your DSP/BIOS application. A period (PRD) object is a special class of a SWI. With a PRD, DSP/BIOS initiates this object at a periodic rate. A general SWI object does not get initiated (or posted) by DSP/BIOS internally. Instead your code needs to activate the SWI when it is needed by calling `SWI_post`.

The goal of this section is to create both a PRD object and a SWI object. As stated in lecture, SWIs are usually used to off load processing from a hardware interrupt (HWI) or other PRDs or SWIs. To simulate this heavy processing load of the SWI, you are given code for a function called `timeload(int waittime)`. (see below) `timeload` delays for the number of milliseconds passed in its parameter `waittime`. You will use this function along with a PRD function and a SWI function to experiment with priority levels of SWI objects.

Switch seats so your partner can code the following:

1. Create a new project with Project Creator.
2. To start out we are just going to add a PRD object and get that working first. The SWI will be added in later steps. Add a PRD function that is called every 1 millisecond just as in exercise 4. Increment a time integer by 1 and print two lines to the LCD every 100ms. Line one should display the time in units of milliseconds and line 2 should display the time in units of seconds.

3. Compile and download your code and see if it works. Find a stop watch and check that time is counting correctly.
4. As a side note, change your code so that it prints to the LCD lines every 1 millisecond. What happens different? Show/Explain to your TA. Remember it takes about 10 to 15 ms for your text to print to the lines of the LCD.
5. Reset your code back to printing every 100 ms.
6. Now we are going to add a SWI to our code. First cut and paste the following function into your code. The `CLK_getltime()` function returns the amount of time elapsed since the start of your program.

```
void timeload(int waittime) {
    int starttime = CLK_getltime();
    while((CLK_getltime()-starttime)<waittime){ /* do nothing */}
}
```

7. SWIs have two names associated with them that are important in your code. These two names need to be different in order for you code to compile. One is the name of the SWI object created in DSP/BIOS configuration tool. Go to the configuration tool and add a SWI to your project. Give the SWI object a name, i.e. `SWI_loaded`. Then in the object's properties we need to tell DSP/BIOS what function to call when this SWI is posted. Enter a name of a function that you will create in the next steps in the function property item, i.e. `_timeloadfunc`. Don't forget the '_'. Why is the '_' needed?
8. Still in DSP/BIOS configuration tool, set the priority of both the PRD and the SWI objects. To do that select the "SWI – Software Interrupt Manager" item. To the right you should then see the different priorities listed. In that window you can drag the different SWI objects to different levels of priority. For this first run, make all the PRD objects have the highest priority. Drag the "PRD_swi" to priority 5. Make sure the new SWI has priority 4.
9. In your main C file, add code in your PRD function to post your new SWI every 20 ms. To post a SWI use the function `SWI_post`. If you named your SWI object `SWI_loaded`, use the following line to post the SWI, `SWI_post(&SWI_loaded)`. The & is needed because `SWI_post` requires the SWI object to be passed by reference. Also, your SWI object needs to be declared in your C file. Towards the top of you C file (after the #includes) add the following line (assuming that you named your SWI object `SWI_loaded`): `extern far SWI_Obj SWI_loaded;`
10. Finally, create the C function that the SWI object has been configured to call. If you used the name given above, the function declaration is `void timeloadfunc(void) {`. This function simply calls the given function `timeload`. Pass a value of 10ms. to this function.
11. Build, download and run your program. Using a stop watch, check if there is any difference in time. If your code is working properly, you should see no difference in time because the PRD has the highest priority.
12. Now go back to DSP/BIOS configuration tool and swap the priorities of your SWI and "PRD_swi". Your SWI should have priority 5 and "PRD_swi" should have priority 4.
13. Build, download and run your program. Now what is the difference in time? Let the DSP run for 30 seconds or so and compare what the DSP displays to a stop watch reading. What factor is time off by? Sketch a time load graph to verify this factor.

14. You may have noticed that there were a number of other SWI objects already being used in your default project. The SWIs are used to process the cameras data, read the robots compass and display the cameras data to the color LCD. Change the SWI time load from 10ms to 50ms. What do you notice with the color LCD display? Compare to a time load of 10ms and 0 ms.

Lab Check Off:

1. Explain the VB serial port settings.
2. Demonstrate your “Hello World” LCD printing to the TA.
3. Demo your working Robot/PC communication system via the wireless serial modem.
4. Demo the 2 priority cases for the SWI and PRD code. Show your TA the time load graph you sketched for the second priority case.

Notes:

1. Contrary to conventional programming, global variables are often preferred in your DSP programs, especially when first learning to program in its environment.
2. The I2C communication of 20 characters to the LCD takes about 15 milliseconds. If we write too often to the LCD task, the I2C queue will be filled up before the hardware can transmit the characters out and additional printing commands will be dropped. This is the reasoning for the 100 ms time spacing in LCD printing.
3. As good coding practice, the expression: `if (1==a)` is preferred over: `if (a == 1)` even though both produce the same result. If you accidentally delete one of the equal signs in the first method, the compiler will give you a nice error. If you accidentally delete an equal sign in the second method, most compilers will attempt to *assign* the value of 1 to the variable a, and the ‘if’ statement will *always* be true. This type of error is extremely hard to find, so don’t give yourself the opportunity to make this mistake in the first place.
4. The ‘extern’ field declares that the variable is defined externally in another source file, the ‘far’ field declares that the variable should be put in the DSP memory away from executing code (this is the default for DSP/BIOS objects because it is assumed that you will have more important variables needing the ‘near’ status see http://coecsl.ece.uiuc.edu/ge423/datasheets/C6713Ref_Guides/OptimizingCCompiler.pdf section 7.4.4 for more information on the ‘near’ and ‘far’ specification), and finally the ‘SEM_Obj’ is a DSP/BIOS defined type that indicates to DSP/BIOS that this variable is a semaphore.
5. Semaphores are the preferred way to allow multitasking functions to communicate or ‘flag’ each other. Therefore, they generally occur in two different functions; for instance, in this project a semaphore is posted in one function, **UART1ReceiveCharTask**, and pending in another, **NetReceiveMessageTask**.