Mechatronics Laboratory Assignment 2
Text LCD screen, SYS/BIOS and Transmission Control Protocol/Internet Protocol (TCP/IP) Communication

Recommended Due Date: By your lab time the week of February 10th
Possible Points: If checked off before your lab time the week of Feb. 17th ... 10 points
If checked off after your lab time the week of Feb. 17th and before your lab time the week of Feb. 24th ... 7 points
If work not finished before your lab time the week of Feb. 24th ... 0 points

Goals for this Lab Assignment:
1. Introduce the LABVIEW environment for PC-based monitoring of the robot
2. Learn to interface to the robot’s text LCD screen
3. Learn about SYS/BIOS tasks and software interrupts

SYS/BIOS Objects Used:
clock, Task, Semaphore_pend, Semaphore_post, Swi, SWI_post

Library Functions Used:
LCDPrintfLine

Prelab: Explore the TMS320 SYS/BIOS User’s Guide (http://coecsl.ece.illinois.edu/ge423/SYSBIOS_usersguide.pdf)
1. Read the following sections to give you an overview of SYS/BIOS. You do not have to understand every detail of these sections. Focus more on the general explanation of what each of these SYS/BIOS items are and their purpose. Read: Sections 1.1 and 1.2, Sections 3.1, 3.2 (3.2.1, 3.2.2, 3.2.4, 3.2.5), Hardware Interrupts sections 3.4 (3.4.1, 3.4.2), Software Interrupts sections 3.5 (3.5.2, 3.5.4, 3.5.5), Task sections 3.6 (3.6.1.2, 3.6.2) and Semaphore section 4.1.
2. Read your C book for information on how to use the ANSI C “printf” and “sprintf” functions. Make sure you know how to use the formatters for int %d, char %c, float %f and char array (string) %s. For example, understand what this piece of code is doing (assume here that printf prints to your PC command console):

```c
char mystring[256];
int strsize;
float gain = 1.3472;
void myfunc(void) {
    strsize = sprintf(mystring,"Gain=%.2f",gain);
    printf("Size = %d, string=%s",strsize,mystring);
}
```
3. 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 clock object, writing a function, creating a new project, etc.

Laboratory Exercise 1: A simple LABVIEW GUI interface
In this portion of the lab, you will create a simple LABVIEW VI from scratch to communicate with your robot.
1. Open a new LABVIEW project and then its VI file. Remember Ctrl-E switches you back and forth between the front panel view and the block diagram view.
2. Use the below block diagram picture and the remaining instructions to build your LABVIEW program.

3. Add a 4 frame flat sequence structure to your block diagram. The flat sequence will allow for initializations, the main body of the program and termination.

4. In the second frame add a While Loop.

5. Add a Stop button in the front panel view but do not wire the Stop button in the block diagram yet. Change the Mechanical action of the Stop button to “Switch When Released.” For now in the block diagram view, place the Stop button outside the While Loop.

6. I like using the Mechanical Action “Switch When Released” (or “Switch When Pressed”) for the Stop button but it requires initialization and termination code to make sure it is in the state you wish when you get to your main code. To control the state of the Stop button you can use a local variable to change its state. In the first sequence frame add a local variable located under “Structures” items. Click on the “?” of the local variable and select “Stop” (or whatever you named your stop button.) Connect a “False” Boolean constant to the input of the local variable. This will force the state of the Stop button to be false at the beginning of the program. The while loop stops when the Stop button returns true. For completeness add a “stop” local variable in the third sequence frame and also set it to false there.

7. In below steps we are going to setup a TCPIP connection between the LABVIEW program and a Linux program running on the robot. A small delay is necessary at the beginning and end during the TCP/IP server connection and termination. So in the first and third sequence frame add a “Wait (ms)” of 1 second (1000 ms) to make sure the first and third frames take at least 1 second.

8. We will finish up the initialization and termination. Add a “TCP Open Connection” found under “Data Communication->Protocols->TCP to the first sequence frame. Right click at TCP Open Connection’s “address”, “remote port” and “timeout (ms)” ports and select create constant. Set the timeout to 1000 ms, remote port to 10001 and address to “192.168.1.7?” where ? is dependent on the robot you are using.

9. In the fourth sequence frame add a “TCP Close Connection.”

10. Now we will add the main code to our VI. The goal of the VI is to open a TCP/IP connection to a Linux program running on the robot car. Our front panel will have a string control, a Send button and a string indicator. The user will type a string to send to the robot car in the string control and then press the send button to send the text to the robot car. Then the robot car will receive that data and echo it back to the LABVIEW program and it will be displayed in the string indicator. This will be repeated until the Stop button is pressed.

11. Add an OK button to the front panel and make its Mechanical action “Switch Until Released.” Also change the OK text to SEND, and change the button’s name to something like “sendbutton.” For now place this button outside of the while loop.

12. Inside the While loop add an Event Structure. Add three events to this structure. One (labeled event “0” in the block diagram) being the default Timeout event. Second (labeled “1”) a Send button Mouse Down event. Third (labeled “2”) a Stop button Value Change event.

13. In the Send button event first place a TCP Write block and a Concatenate String block. Wire the “Connection ID” output of the “TCP Open Connection” block to the “Connection ID” input of the TCP Write block. Wire the TCP
Write block timeout to a constant setting it to 100 ms. Wire the output of the Concatenate String block to the “data in” port of the TCP Write block. Finally create an Indicator to display the number of bytes written output port.

14. Drag down on the Concatenate String block so that three items will be concatenated together. Create a constant string for the first element, a control string for the second element (call it “Send String”) and a constant for the third element. The Linux program is waiting for the start character 0xFD to begin the string and the stop character 0xFF to end the string. To send these hexadecimal numbers as a character select the constant string and right click on it. Down towards the bottom of the displayed menu you should find and select “‘\’ Codes Display”. Do this for the two constants and type in the first “\FD” and the second “\FF”.

15. That completes the sending of a string to the robot, now wire the code to receive the string sent back to LABVIEW. Place a “TCP Read” block after the TCP Write block. First wire the “Connection ID” output of the TCP Write block to the “Connection ID” input of the “TCP Read” block. Also while we are here, wire the “Connection ID” output of the “TCP Read” block to the “Connection ID” input of the “TCP Close Connection” block which is located in the fourth sequence panel.

16. Create constants for the “TCP Read” block setting the Timeout to 3000 ms and the bytes to read to 100. Also create an indicator in the front panel view and connect its input to the “data out” of the “TCP Read” block in the block diagram. Name this indicator “Received String.”

17. The final addition for the “TCP Read” block is to create a constant to connect to the “mode” input. Once again, this is easiest to do by right clicking on the “mode” input node and selecting “Create->Constant”. The data transfer method between the robot and LABVIEW is slightly different than then the LABVIEW to robot transfer. For the LABVIEW to robot transfer we placed a start and stop character at the beginning and end of the string. The “TCP Read” block can be set to implement a different transfer waiting until the carriage return and line feed characters are seen in sequence. This sequence indicates the end of the string. To put the “TCP Read” block in this mode, click the down arrow of the “mode” constant and select “CRLF.”

18. So that the event structure works correctly with the Send button that was created in step 12, move the Send button’s icon inside the Send Button Mouse Down Event.

19. The Timeout and Stop events are simple. In both events pass the “Connection ID” output of the “TCP Open Connection” block through to the output on its way to the “TCP Close Connection” block. For the Timeout event connect a False Boolean constant to the Stop button port. Also in the top left of the event structure there is a small hour glass icon. Right click on the input of this hour glass icon and create a constant and set the Timeout to 50ms. In the Stop button event connect a True Boolean constant to the Stop button port. Also here in the Stop button event place the Stop button icon.

20. At your robot’s Linux console (Tera Term connected to COM1) first login with “root” as the user name and nothing for the password. Run the Linux application LVecho by typing “./LVecho”. The “./” tells Linux to look in the current directory for the application to execute. This application will be used to verify your LABVIEW GUI is working. It simply receives the messages you send from LABVIEW, prints them to the Linux console and echoes them back to your LABVIEW application.

21. Run your LABVIEW application, and type a message into Send String control. Click on the Send button and your message should print at the Linux console and echo back to the Received String indicator.
22. When you are done playing with this end, click the STOP button and close the LVecho application at that Linux console by typing Ctrl-C.

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

Laboratory Exercise 2: Printing to the text LCD screen

The robot has two LCD screens. The color LCD screen is dedicated to displaying the camera’s image. The other LCD is a black and white (actually black and green) text only LCD. We will use this LCD for printing sensor data or any other information that will help us develop our code for the robot. To print to this LCD use the given function `LCDPrintfLine`. This function is similar to the ANSI C function “printf” except that the first parameter is the LCD line you would like to print to and only 20 characters can be printed. For instance these two instructions would write to LCD line 1 and 2.

GE423, Mechatronic Systems
LCDPrintfLine(1,"Robot ready");
LCDPrintfLine(2,"Let’s Roll!");

To test out printing, add a clock object to your project. Have this clock call its function every 1 ms. See Lab 1 exercise 3 part c) for a reminder how to add a clock object. Inside the function, increment a global integer by one each time it is called by SYS/BIOS, therefore becoming a count variable keeping track of time in units of milliseconds. Use this integer along with the modulus “%” operator to determine when 250ms have passed and write a message including the count value to the LCD screen’s line 2 every 250ms. Write another personalized message to LCD line 1.
For example if mytimecnt is incremented every 1ms. this piece of code would call some function to do something every 33ms.

if (0==(mytimecnt%33)) {
    do something here;
}

See Notes, 1 and 2 at the end of the lab.

Demonstrate this working to your instructor.

**Laboratory Exercise 3**: Tasks (Task), Semaphores (Semaphore) and communicating with a Linux application through shared memory.

As you learn SYS/BIOS or any other operating system/scheduler, 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 SYS/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.

A Linux application, **LVDSPComm**, will be used in this exercise to receive and send TCP/IP data to and from LABVIEW and your robot. **LVDSPComm** is similar to the **LVecho** application you just used but **LVDSPComm** communicates the received TCP/IP data to the DSP core of the OMAPL138 processor and waits for return data from the DSP core to send back to the LABVIEW GUI. To perform this communication between Linux on the ARM core and your application on the DSP core a section of shared memory is used. The include file “sharedmem_com/sharedmem.h” defines the structure of this shared memory. If you choose to develop your own Linux application as a part of the project for this class you may very well change this structure to communicate different information. For this lab we are going to use the DSPrec_size, DSPrec_buf, DSPSend_size and DSPSend_buf variables in the structure along with the LVDATA_FROM_LINUX and LVDATA_TO_LINUX flags. In your DSP code there are 6 Macros that you will be using to check status and to set and clear the communication flags. They are defined in “COECSL_registers.h” and are GET_LVDATA_FROM_LINUX, SET_LVDATA_FROM_LINUX, CLR_LVDATA_FROM_LINUX, GET_LVDATA_TO_LINUX, SET_LVDATA_TO_LINUX, CLR_LVDATA_TO_LINUX. The GET Macros return the state of the flag, whether 1 or 0. The SET Macros set the flag to 1 and the CLR Macros clear the flag to 0. Using these flags the given Linux program **LVDSPComm** communicates with your DSP program as follows:

1. **Data from Linux communicated to the DSP** Note: This is an overview. Lab procedure starts under the section “Add the Task to your Project.”
a. When Linux has data to communicate to the DSP (in our case this will be when the LV program sends data over Ethernet to Linux) it checks if the LVDATA_FROM_LINUX flag is 0. If it is 0 Linux knows that the DSP is ready for new data. If the flag is 1 the DSP is not ready, and the data is ignored and not sent to DSP. (This could be improved to retry until DSP is ready, but more error checking would be involved.)

b. The flag was found to be 0 so Linux writes the data to the array in shared memory

\[\text{ptrshrdmem->DSPRec_buf}\] 

Also, the number of characters written to this array is stored in the variable \[\text{ptrshrdmem->DSP_Rec_size}\].

c. Now that the data and the data size have been written to the shared memory, Linux sets the LVDATA_FROM_LINUX flag to 1 indicating to your DSP program that there is new data ready to be read in the shared memory.

d. Your DSP program should periodically check the LVDATA_FROM_LINUX flag to see if Linux has changed it to 1. (We are going to perform this continuous checking inside a Task function.)

e. When your DSP program finds that the LVDATA_FROM_LINUX flag has been set to 1 it starts the process of copying the data from the shared memory variables into its own local or global variables.

i. First step is to invalidate the DSP’s cache:

\[
\text{Cache_inv((void*)ptrshrdmem, sizeof(sharedmemstruct), Cache_Type_ALL, EDMA3_CACHE_WAIT);}
\]

(We will talk more about cache memory in lecture.) We do not invalidate all the DSP’s cache, just the size of the shared memory structure. We need to perform this cache operation because we are dealing with two processors accessing the same memory. Linux, on the ARM processor, is writing data to a place in memory that the DSP has already accessed and may have placed in its cache memory. This invalidating of the cache memory for the shared memory section, forces the DSP to read the memory from the actual memory instead of the faster cached memory.

ii. Second step then is to copy the data from the shared memory into a local or global variable only used by the DSP program.

f. Now after the reading of the shared memory is complete, the DSP clears the LVDATA_FROM_LINUX flag back to zero telling Linux that the DSP has read the data in shared memory and is ready for another communication transfer.

2. Data from your DSP program communicated to Linux. Note: This is an overview. Lab procedure starts under the section “Add the Task to your Project.”

a. When there is data to communicate from the DSP program to Linux (and in our case then onto LABVIEW) the DSP program checks the flag LVDATA_TO_LINUX to see if it is one. If it is one the DSP program knows that the Linux program is ready to read new data in the shared memory. Again there are two steps to copy the data correctly into the shared memory.

i. First from the DSP program, write the data from a local or global variable into the shared memory array \[\text{ptrshrdmem->DSPSend_buf}\]. Also write the number of characters copied to this array in the shared memory variable \[\text{ptrshrdmem->DSPSend_size}\].
ii. The second step deals with the cache memory. The copy instructions to shared memory in the above step could have written the data into faster cache memory instead of the shared memory location. To make sure the data is not sitting in cache memory, the DSP needs to perform a cache write back command:

```
Cache_wb((void*)ptrshrdmem,sizeof(sharedmemstruct),Cache_Type_ALL,
EDMA3_CACHE_WAIT);
```

This forces the DSP to copy the data possibly written to cache into the correct shared memory location. This is necessary because the ARM processor running Linux does not use the same cache memory as the DSP and cannot read data out of the DSP’s cache.

b. Now that the data is sitting in shared memory the DSP program clears the LVDATA_TO_LINUX flag to 0 indicating to Linux that the shared memory is ready to be read.

c. The Linux program periodically checks the state of the LVDATA_TO_LINUX flag.

d. When Linux finds the flag set to 0, it proceeds to copy `ptrshrdmem->DSPSend_size` of data from the `ptrshrdmem->DSPSend_buf` into its own local array.

e. As a final step, Linux sets the LVDATA_TO_LINUX flag back to 1 to communicate to the DSP program that it is ready for the next communication of data. (In our case it then also sends the data over TCP/IP to LABVIEW.)

Below you will be asked to use a task (Task) to implement this communication scheme described. A Task is a good priority level for this procedure since there or no real-time requirements for this transmitted data. One, this data is coming from you (the user) who compared to the processor’s data rates cannot type very fast. Two, the data is sent over Ethernet which is fast most of the time but we can’t guarantee real-time performance every transfer. In other words, there are much more important processes running in you DSP code than communicating with your LABVIEW application.

**Add the Task to your project**

Continue working with the CCS project you created in Exercise 2. Verify that in your project’s Task/BIOS configuration there is already a Task object called TASK_ComWithLinux that is associated with the function ComWithLinux. Along with verifying these names in the Task object, make sure the Stack size is 2048. Save those settings and go back to your project’s main C file and find the function ComWithLinux. Follow the steps below to add code to this function. A few of the below steps have already been done for you.

**The Task should have the following functionality**

1. The Task’s function, inside its continuous loop, will every 40ms check to see if the GET_LVDATA_FROM_LINUX Macro returns 1.

   a. Find the Task function `void ComwithLinux(void) {`

   b. We never want this Task to exit so notice that a continuous while loop surrounds all the code of the TSK.

   c. To have the Task check the LV DATA_FROM_LINUX flag every 40ms use the Task_sleep(num_ms) function to put the TSK to sleep for num_ms milliseconds. Leave this at the bottom of your Task function’s continuous while loop.

   ```
   Task_sleep(40);
   ```

   d. To check if Linux has put data in the shared memory you will monitor the LVDATA_FROM_LINUX flag with the `GET_LVDATA_FROM_LINUX` macro. For example:
if (GET_LVDATA_FROM_LINUX == 1) {
    // read shared memory
}

2. When GET_LVDATA_FROM_LINUX does return 1 there is data from Linux sitting in shared memory. The Task code should then follow steps 1e-1f of the section Data from Linux communicated to the DSP above. Summarizing these steps and adding some tips:
   a. When your DSP program finds that the LVDATA_FROM_LINUX flag has been set to 1 it should start the process of copying the data from the shared memory variables into its own local or global variables.
      i. First step is to invalidate the DSP’s cache: *This line of code is given to you. Just make sure to move it inside the If statement you just created.*
         Cache_inv((void*)ptrshrdmem,sizeof(sharedmemstruct),Cache_Type_ALL,EDMA3_CACHE_WAIT);
      ii. Second step is to copy the data from the shared memory into a local or global variable only used by the DSP program. Copying the char array (string) from/to shared memory to/from a global char array requires you to use a for loop and copy one character at a time. i.e.
         for (i=0;i< ptrshrdmem->DSPRec_size;i++) {
             mystring[i] = ptrshrdmem->DSPRec_buf[i];
         }
   b. Now after the reading of the shared memory is complete, the DSP clears the LVDATA_FROM_LINUX flag back to zero telling Linux that the DSP has read the data in shared memory and is ready for another communication transfer. To set the LVDATA_FROM_LINUX flag to 1 or 0 and the LVDATA_TO_LINUX flag to 1 or 0 simply call the macros like this:
      
      SET_LVDATA_FROM_LINUX;  // set to 1
      CLR_LVDATA_FROM_LINUX; // set to 0
      SET_LVDATA_TO_LINUX;  // set to 1
      CLR_LVDATA_TO_LINUX;  // set to 0

3. Here in future labs you would do something with the data sent from Linux. Maybe it would be a new color of golf ball to search for. But in this lab, just to test communication between LABVIEW and the robot, we are going to keep the data in a string (character array) format, add some additional text to the string and send it back to LABVIEW for viewing.

4. The below steps should only occur if data has been received from Linux, so in other words the code for the below steps show be inside the if statement from above.

5. Using the “sprintf” function, build a string to send back to Linux/LABVIEW. Use another character array to store this string i.e. char TXstring[256];. This string to be sent back to LABVIEW should include the string just received and the value of the count variable you are incrementing in your Clock. For example if you typed “This is a test” into LABVIEW and clicked the Send button something like the following should be echoed back to LABVIEW: “This is a test, Count = 12240”. Remember also that “sprintf” returns the size of the string built.

6. To send this string back to Linux/LABVIEW perform these steps:
   a. When sprintf returns, check if Linux is ready for new data by checking if the LVDATA_TO_LINUX flag is set to 1 If it is not 1 do nothing and the string is lost (This should never happen). If it is 1, perform these 2 steps to copy the string into shared memory.
i. Write the data from this transmit string into the shared memory array
   
   `ptrshrdmem->DSPSend_buf`. Remember you will need to use a for loop to copy each character at a time. Also write the number of characters copied to this array in the shared memory variable `ptrshrdmem->DSPSend_size`.

ii. The second step deals with the cache memory. The copy instructions to shared memory in the above step could have written the data into faster cache memory instead of the shared memory location. To make sure the data is not sitting in cache memory, the DSP needs to perform a cache write back command:
   
   ```c
   Cache_wb((void*)ptrshrdmem,sizeof(sharedmemstruct),Cache_Type_ALL,
            EDMA3_CACHE_WAIT);
   ```
   This forces the DSP to copy the data possibly written to cache into the correct shared memory location. This is necessary because the ARM processor running Linux does not use the same cache memory as the DSP and cannot read data out of the DSP’s cache.

b. Now that the data is sitting in shared memory your DSP program should clear the LVDATA_TO_LINUX flag to 0 indicating to Linux that the shared memory is ready to be read.

7. For now, only respond to Linux/LABVIEW after receiving data from Linux/LABVIEW. In future labs we will change this code so that the send and receive from LABVIEW do not have to be synchronized. For example we will want to send the robot’s position to LABVIEW every 250ms without having LABVIEW send us any information.

Running/Testing your code

When you have successfully built your project download it to the DSP core. Before you run this code, first switch to the Linux console and execute the LVDSPComm application. Type ./LVDSPComm. Second launch your LABVIEW application. You should see at the Linux console that a successful TCP/IP connection was made between the Linux application and LABVIEW. Finally run your DSP code.

Type in a number of text messages in your LABVIEW application and make sure that the correct messages are being returned. Demonstrate this working to your instructor.

Add another Task and use a Semaphore (semaphore) to activate this TSK

To introduce you to semaphores and a bit more experience with tasks, add a second task and a semaphore (found in SYS/BIOS .cfg file, in outline towards right) to your project. The functionality of this Task will be much simpler than our first Task. We again do not want this Task to exit, so all the Task function’s code will be inside a continuous while-loop. Also make sure the Stack size for this Task is set to 2048. The first line of code inside this while-loop should be the function call `Semaphore_pend(sem_<whateveryoucalledit>,BIOS_WAIT_FOREVER);`. This puts the Task to sleep until another process calls the function `Semaphore_post(sem_<whateveryoucalledit>);`. The `BIOS_WAIT_FOREVER` parameter indicates that the Semaphore_pend function should never time out and suspend forever until a Semaphore_post call. `BIOS_WAIT_FOREVER` could be replaced with an integer that indicates to Semaphore_pend either to suspend until a Semaphore_post call is made or the integer number of milliseconds have elapsed.
Once Semaphore_post is called in another SYS/BIOS process the scheduling kernel wakes up the Task sitting in the pending state and runs the code after the SEM_pend call. All I want you to do in this simple Task is count the number of times the Task wakes up from the pending state and print that value to the text LCD screen. If somewhere else in your code you are writing to the same line of the LCD, comment it out so it does not conflict with this LCD print statement. Once your Task function is finished calling the LCDprintfLine function it should loop back to the top of the while loop repeating the process. Post the semaphore from inside your first Task when data is received from Linux/LABVIEW. This way the count value displayed to the LCD screen will be the number of times you clicked the Send button in LABVIEW. Again build and run this code and demonstrate to your instructor.

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 SYS/BIOS application. A Clock object is a special class of a Swi object. With a Clock, SYS/BIOS initiates this object at a periodic rate. A general SWI object does not get initiated (or posted) by SYS/BIOS internally. Instead your code needs to activate the SWI when it is needed by calling Swi_post.

Along with a Clock object we will add two Swi objects to experiment with the priorities of the Swi object. As stated in lecture, SWIs are usually used to off load processing from a hardware interrupt (Hwi) or other Clock or Swi and many times this off loaded code can take a while to process. To simulate this heavy processing load of the SWI, you are given code for a function called timeload(int waittime).(see below) All the timeload function does is delay for the number of milliseconds passed in its parameter waittime.

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 Clock object and a single Swi object and get that working first. Add a Clock object that calls its function every 1 millisecond. Give it a name like “CLK_every1ms”. Also add a new Swi to your SYS/BIOS configuration. Give it a name like “SWI_counttime”. Now that you have added both a Clock and a Swi object you need to go to your C file and create a function that the Clock will call every 1ms and a function that the Swi will call when the SWI is posted. The code for the Clock’s function is very simple for this exercise. All it needs to do every 1ms is post the new SWI you just created. So if you named your Swi “SWI_counttime”, the only line of code in your PRD’s function should be Swi_post(SWI_counttime); Then inside the SWI’s function we want to keep track of expired time. Since this SWI is being posted every 1ms by your new Clock object, incrementing an integer by 1 each time into this function creates a variable that is equal to elapsed time in units of milliseconds. In addition to incrementing this integer each call, also print time to the text LCD screen every 100ms. LCD Line one should display the time in units of milliseconds and to give you practice at printing a floating point variable, line 2 should display the elapsed 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. It should be.

4. Now we are going to add a second Swi object to our code. First cut and paste the following function into your code. The Clock_getTicks() function returns the amount of time elapsed since the start of your program, so what this function does is simulate a time load. It does not return until “waittime” number of milliseconds have elapsed.

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

5. Explaining again to be thorough, 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 SYS/BIOS configuration tool. Open the SYS/BIOS configuration tool and add a second new SWI to your project. Give this Swi object a name, i.e. SWI_loaded. Then in the object’s properties we need to tell SYS/BIOS what function to call when this SWI is posted. Enter a name of a function in the property window that you will create in the next steps, i.e. stallfunc.

6. Still in the SYS/BIOS configuration tool, we need to change the priority levels of all the SWI objects to get this simple demo to work correctly. To set these priorities click each Swi object and there should be an option of ‘Interrupt Priority’. For this first run, set the priorities as follows: SWI_RobotControl priority 5; SWI_vision priority 4; SWI_LCD priority 3; your first added SWI “SWI_counttime” priority 2; your second created SWI “SWI_loaded” priority 1.

7. Because there is a large amount of processing code running on the DSP when the camera data is being processed and displayed to the color LCD screen, we need to disable this code to make this simple example work properly. To disable the vision processing open the file ColorVision.c. In ColorVision.c find the function `void vision(void)`. In this function comment out the line `userProcessColorImageFunc_laser(Image_data);`

8. Then back in your SWI_counttime’s function we want to also post the SWI_loaded object every 20ms. Reviewing again, 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_loadedSWI_loaded` is a structure that is declared for you in the file `<yourprojectname_pe674.h>`. You can find the #include of this file in “projectinclude.h”.

9. Finally, create the C function that this second Swi object has been configured to call. If you used the name given above, the function declaration is `void stallfunc(void)`. This function simply calls the given function `timeload`. Pass a value of 10ms. to this function. Thus, creating a schedule that posts this second SWI every 20ms and this SWI simulates a processing time of 10ms.

10. 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 first Swi object has higher priority than the second simulated heavy processing Swi.

11. Now go back to SYS/BIOS configuration tool and swap the priorities of your 2 SWIs. i.e. SWI_counttime priority 1 and SWI_loaded priority 2.
12. Build, download and run your program. Now what is the difference in time? Let the DSP run for 30 seconds or so. How does the time a stop watch displays compared to the time displayed on the robot’s LCD screen?

What factor is time off by? Sketch a time load graph to verify this factor.

Lab Check Off:

1. Demo your LABVIEW application working with the LVecho Linux application. Explain the TCP/IP settings for the Winsock control.
2. Demonstrate your “Hello World” LCD printing and continuous printing of time.
3. Demo your working Robot/PC communication using the LVDSPComm Linux application. Also Demo your second Task waking up and printing to the LCD screen.
4. Demo the 2 priority cases for the SWI 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. As good coding practice, the expression: \texttt{if (1==a)} is preferred over: \texttt{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 \texttt{a}, and the ‘if’ statement will \textit{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.