GE420 Laboratory Assignment 3

More SYS/BIOS

Goals for this Lab Assignment:

1. Introduce Software Interrupt Objects (Swis)
2. Introduce 2 X 20 character LCD functions.
3. Investigate an issue with 32 bit floating point precision when keeping track of elapsed time.
4. Introduce Semaphore and Task Objects.

SYS/BIOS Objects Used:

- ClOCK
- Swi
- Semaphore
- Task

Given Functions:

- UART_printfLine

Matlab Functions Used:

(none)

Prelab: The SYS/BIOS User's Guide: http://coecsl.ece.illinois.edu/ge420/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. If you do not know how to use the ANSI C "printf" function, dust off your old C book and read about printf. You also may want to bring your C book to lab.

Laboratory Exercise

Today we are going to create a simple application that uses a number of SYS/BIOS processes: Clock, Swi and Task. We will also look at how SYS/BIOS schedules and prioritizes these processes.

We will also investigate a problem where a floating-point number can lose precision in a calculation that involves both large and small numbers. To demonstrate this problem, we will setup an application where the SYS/BIOS kernel calls a function every **1 ms**. In this function, we will keep track of the elapsed time by incrementing both a floating-point variable (units in seconds) and in an integer variable (units in milliseconds). We will print both of these values (both scaled to units of seconds) to the character LCD and compare them as time elapses. 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.

Exercise 1: LCD Printing and Time-Keeping Application.

1. Create a new project with the Project Creator.
2. Write a function called `void timecountClock(void)`. You will setup a Clock process to call this function every 1 millisecond. Knowing this, write this function to keep track of elapsed time with both a global floating-point variable and a global long integer variable. Have the long integer variable keep track of time in units of milliseconds and have the floating point variable keep track of time in units of seconds. To see that the time variables are incrementing, print their values (both in units of seconds) to the character LCD every 2**00ms** (see below). Use the `UART_printfLine()` function within your function to print the two time values to the character LCD. For example:

   ```
   UART_printfLine(1,"timeint=%.4f",(float)(timeint*0.001));
   ```

   Comments:

   - Contrary to conventional programming, in many cases global variables are preferred in your DSP programs.
   - The serial communication of 20 characters to the serial port takes about 11 milliseconds. If we write too often to the serial port task, the serial queue will be filled up before the hardware can transmit, and additional printing commands will be dropped. This, in addition to wanting to be able to read the data on the LCD, is the reasoning for the 200 ms time spacing in LCD printing.
   - As a hint, the mod operator, **%**, will help you to perform printing every 200 milliseconds. For example, the if statement:

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

     only executes if the integer 'timeint' is evenly divisible by 33.
3. Add to the SYS/BIOS configuration a Clock. *It must have a different name then the function it is to call.* Set it up to call `timecountClock` every 1ms.
4. Build, download, and run your application. Use your phone's stop watch and check that time is counting correctly. You will need to wait for about 300 seconds or so to see the obvious numerical issues with the floating point variable, so be patient. Which of your times (integer or float) are correct, and why is one of these incorrect?
5. Terminate your Debug session and make one small change to your code. Initialize both your floating point variable and long integer variable keeping track of time to start at 3000 seconds. So start the float variable to 3000.0 and the long variable to 3000000. Debug and run this code. What become very obvious only after 10 to 20 seconds of time elapse when comparing the float and long variable.

Exercise 2: Creating Swis 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. 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`.

The goal of this section is to add two Swi objects to the application you started in exercise 1. As stated in lecture, Swis are usually used to off-load processing from a hardware interrupt (HWI), other Clocks or other 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 to experiment with priority levels of Swi objects.

1. Now we are going to add Swis 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.

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

2. 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. 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 SYS/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`. Also give this Swi Interrupt Priority 0 to start. There is a "Scratch" table on the last page of the lab that you may want to fill out to help you keep track of the different threads you use during this lab.
3. Add a second Swi. Call this Swi something like `Swi_count`. Have this Swi be associated with the function, say `timecountfunc`. Start this Swi out with the Interrupt Priority 1.
4. In your main C file you need to define the Swis you created in SYS/BIOS. After the #include statements add the line `extern const Swi_Handle Swi_loaded;` and `extern const Swi_Handle Swi_count;` If you called your Swis something different change to the names you chose.
5. Also your main C file we need to change a few things. Sorry this may be a bit confusing so ask your TA if you get confused. First copy the Clock function you created earlier, ie `timecountClock,` and use it to create the function `timecountfunc` just mentioned above. In other words `timecountfunc` is going to do everything `timecountClock` was doing. Then delete all the lines of code in your timecountClock function getting it ready for the code specified in a few steps.
6. Create the C function `timeloadfunc` that the Swi object `Swi_loaded` has been configured to call. i.e. `void timeloadfunc(void) {`. Have this function simply call the given function `timeload`. Pass a value of 10ms to this function.
7. Now inside the Clock's function (`timecountClock`) which is being called every 1ms, increment by one a new global long integer as yet another time count variable. Post the Swi `Swi_count` every time into this function. Post `Swi_loaded` every twentieth time (20ms) into this function. 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)`.
8. So in summary, what you have created is a Clock function that is called every 1ms. It posts a Swi every 1ms that has priority 1 and posts another Swi every 20ms that has priority 0. The Swi with priority 1 just keeps

track of time and print to the LCD every 200ms. The Swi with priority 0 simulates a large amount of code by loading itself for 10ms.

9. 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 time keeping SWI has the highest priority.
10. Now go back to SYS/BIOS configuration tool and change the priorities of your Swis. Switch `Swi_loaded` to priority 1 and `Swi_count` to priority 0.
11. 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**.
12. To get ready for the next exercise, reset the priorities back so that `Swi_count` has the highest priority.

Exercise 3: Add a Task to Process user Input

In this section you are going to add a Task process to your application. The task's job is to decide what to do when the state of the four switches have changed. You can think of this as a low priority user interface for the DSP. Tasks are usually used for these lower priority operations.

1. Trade programming duties with your partner.
2. Add a semaphore to your SYS/BIOS configuration. Give it a name like "**Semaphore_switchesChanged**". You will be using this semaphore in your code, so just like the Swi in the previous exercise, you need to tell the compiler that this semaphore exists externally. Towards the top of your source file add:
   ```
   extern const Semaphore_Handle Semaphore_switchesChanged;
   ```
3. In the function associated with Swi `Swi_count`, add code to check every 100ms if the switches have changed. You will want to copy your ReadSwitches function from Lab 2. To do this you will have to keep track of what the switch state was the previous 100ms sample to check if they have changed. If the switches have changed, first save the switch state to a global integer variable and then post your created semaphore with the function `Semaphore_post`. Just as with `Swi_post`, `Semaphore_post` needs to be passed the semaphore handle.
4. Now create a Task to wait for that semaphore signal. A Task also has two names associated with it. The SYS/BIOS name of the Task and the function that the Task object launches ONCE. That is a key point of a Task. It only calls this function once so you need to write the Task function in a way that does not allow the function to exit. You can think of this function as a separate process or thread. For our purposes when creating Task functions we will always use an infinite loop to remain in the Task.
5. In SYS/BIOS create a Task object and give it a name. In your main C file write its Task function. Make sure in the SYS/BIOS configuration to set the Task object's "function" to this function's name. Also in the SYS/BIOS configuration make sure to change the Stack Size to 2048.
6. The Task function should first have an infinite loop as discussed above. Inside the infinite loop the task should call the `Semaphore_pend` function to wait forever for the semaphore:
   ```
   Semaphore_pend(Semaphore_switchesChanged, BIOS_WAIT_FOREVER);
   ```

7. With BIOS_WAIT_FOREVER passed to `Semaphore_pend` it will not exit until another process has called the `Semaphore_post` function. Exiting from `Semaphore_pend` indicates that the semaphore has been posted and processing should continue. For our simple application, the processing is going to be printing text to the character LCD. After processing the data communicated to the Task the code should loop back up to the `Semaphore_pend` function and wait for the next semaphore to arrive.
8. To get practice with the "switch" statement, add a switch statement after the `Semaphore_pend` call (and still inside the infinite loop) to check what new state was communicated to the task through the global integer variable storing the switch state.
9. For each possible state, 0-15, print a different line of text to the LCD.
10. Build your project and Demo it to your TA.

Lab Check Off:

1. Demo your Clock application that prints time in seconds to the LCD using both the integer time variable and the float time variable. Let the application run until a difference is noticed. **Why** is there a difference?
2. Demo your application with a Swi object added. Explain how the priority scheduler works.
3. Demo your application with a Task added. This task is "asleep" until the user changes the state of the switches but when awoken prints a corresponding line of text to the LCD.

Scratch Table to help you keep track of your application's different threads.

Type of Thread (HWI, SWI, Task,Clock,Timer)	**Handle (Name you assigned in the .cfg file)**	**Function Name (Function you associated with this thread in .cfg file)**	**Possible other information like Priority for a SWI**