ECEN 5823 Spring 2025
Assignment 4 - Si7021 and Load Power Management Part 2

Objective: Modifying the previous homework assignment to use a state machine, interrupt driven LETIMER0 wait routine, and interrupt driven I2C operations. Compare power consumption against the previous implementation. Incorporate a timestamp into your LOG_***() messages.

Note: This assignment will begin with the completed Assignment #3 - Load Power Management Part 1 assignment.

Instructions:

1. Start by creating your assignment repository using the assignment link at https://classroom.github.com/a/4tGkeglZ. You will not need to clone this from the GitHub repo you just created into a new local directory since you will be starting with your code from the previous assignment. Instead, follow these instructions and run these commands with git bash within your assignment directory. Ensure you are setup to use SSH keys before proceeding, see the instructions from github at https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/ and be sure to use Git Bash For Windows or Cygwin if using a Windows environment. (Cygwin .ssh directory will be different than Git Bash.)

   a. On your local PC, duplicate the folder created in Assignment #3 to a new folder, name it something like ecen5823-assignment4-<username> where <username> is your GitHub username.
   b. Change your current directory to this new folder ecen5823-assignment4-<username> and execute the following git commands.
   c. git remote remove origin
   d. git remote add origin <url>
      i. Where <url> is the URL for the repository created with the link above
      ii. This adds the new submission repository created in the link above as your new origin.
   e. git push origin master
   f. Import ecen5823-assignment4-<username> into Simplicity Studio
      i. Remember to perform a **Clean…** immediately after you import the project into Simplicity Studio.
2. Modify **timerWaitUs**(uint32_t us_wait) to support non blocking waits (interrupt driven) for the wait states needed by the Si7021 I2C routines.
   - **Your code should configure interrupts for the timer in a way that ensures an interrupt occurs to wake up the MCU as configured by timerWaitUs(), and that an event is created by your scheduler code, and that event is passed to the**

state machine described below. While waiting for the LETIMER0 interrupt, your program should sleep to EM3. Reminder, you may find it useful to maintain 2 versions of your wait routine. Example function names could be: timerWaitUs_polled() **and** timerWaitUs_irq()**.**

3. Implement the **letimerMilliseconds()** commented out in log.c to return a count of milliseconds since power up. See file log.c. Add this function to irq.c. One way to do this would be to use the values read from a running counter triggered by LETIMER0 UF interrupts, incrementing in milliseconds on each UF IRQ. Your logging routines should now include a log timestamp to help you analyze when events occurred in time. **For the code you submit, letimerMilliseconds() must be based on LETIMER0 interrupts.**
    - You may ask, timestamps based on 3 second intervals doesn't shed much insight on when things happen? True. You can add additional resolution by incorporating COMP1 interrupts as well as one option. For debugging purposes only, I direct your attention to "Systick Timer" in the CMSIS-CORE library. This is a timer that lives inside of the M4 CPU and can be configured to generate an IRQ every millisecond. This will provide much finer resolution (granularity) than the LETIMER0 based timestamp. If you choose to explore this option, include a #define that causes Systicks to be used for **letimerMilliseconds()** for debugging, and when not defined, your code shall use LETIMER0 for **letimerMilliseconds()**.
4. Modify I2C routines and si7021 temperature read and low power management functions to replace any inline waits with a **state machine** implementation, using your new **timerWaitUs_irq(),** interrupt-based routine, supporting non blocking waits. Also develop and deploy interrupt driven I2C transfers.
    - **Your I2C code should not contain any while() loops or inline waits within your I2C temp sensor routines, and should instead be sleeping during wait periods via calls to sl_power_manager_sleep() in main.c.**
    - **Your design should sleep to EM1 while performing I2C transfers, and EM3 at all other times when not running in EM0. Use sl_power_manager_add_em_requirement() and sl_power_manager_remove_em_requirement() to limit the MCU to EM1 during the I2C transfers.**
    - **Your design should use interrupts for I2C transfers and should not call I2CSPM_Transfer() or similar function as you did with the previous assignment. In my code, I did leave in the call to I2CSPM_Init() to setup the I2C (you can do the same) and then used the interrupt based I2C API calls in EMLIB.**
    - **Your design should use your scheduler and interrupt generated events to drive the behavior of your program. Your scheduler shall support 3 events that can all be present at the time:**
        i. **LETIMER0 UF**
        ii. **LETIMER0 COMP1**
        iii. **I2C Transfer Complete**
    - Place your temperature sensing **state machine** code in scheduler.c/.h

5. For the code you submit for grading, **the only LOG_***() call should be 1 call to print the temperature measurement.** No other log messages should be displayed in the terminal window except for API function calls that return errors. **Also, no calls to CMSIS Systick routines or __BKPT(0) are allowed. Your letimerMilliseconds() implementation must use LETIMER0 interrupts.**

Questions:
Answers to the Assignment4-I2CLoadPowerManagementPart2.md file in the questions folder, included in your submission.

Deliverables:

- Submission via **github classroom** and your repository setup via the classroom link at above. Use the following git command to submit your assignment:
  git push origin master
- Verify your ecen5823-assignment4-<username> repository contains your latest/ intended code and answers to the questions. It is your responsibility to push the correct version of your project/code. Use a web browser to verify the correct version of your project/code has been uploaded to GitHub.
- In **Canvas**, submit the URL to your github repository and in **GitHub** create a tag of your GitHub submission prior to the due date with the text of A4-final. The date and time of the tag creation is what we use to determine whether your submission was on time or late. These 2 steps will complete your assignment submission.

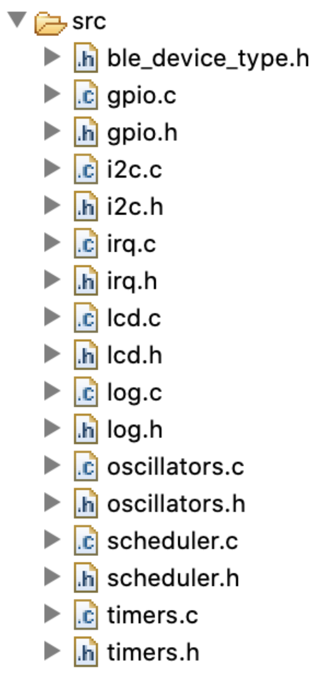
Approach/Guidance:

1. Update your LETIMER0_IRQHandler to count milliseconds. Yes, this is 3000ms increments, not terribly high-resolution, but it is energy efficient. Then update **loggerGetTimestamp()** in log.c to return this "time". Connect a terminal program and validate your **loggerGetTimestamp()** is returning non-zero time stamp values.
2. Leverage your work from assignment2 using capability of the LETIMER0 to use interrupts in the **timerWaitUs()** function instead of the simple polled scheme as in assignment3, sleeping to EM3 while waiting for the requested delay to expire. Get this working first, write some test code to verify that your interrupt driven **timerWaitUs()** is working correctly. You could name your **timerWaitUs()** functions **timerWaitUs_polled()** and **timerWaitUs_irq()** for example, as mentioned above, to have both of these available for development. Develop unit test code for your interrupt-based timer. Like in A3, unit test **10ms, 100ms and 1 sec** delays using LEDs and the Energy Profiler. 90% of the I2C transfer failures that occur in A4 are due to errors in the implementation of the

timerWaitUs_irq() function. Suggestion for this unit testing: Build a new testing state machine that transitions back and forth between 2 states based on the **timerWaitUs_irq()** COMP1 interrupt, and use the LEDs and the Energy Profiler to verify your design. In this assignment we allow the MCU to sleep to EM3. In the next assignment, we have limit the MCU to EM2. So verify both of your timerWaitUs functions (polling version and interrupt version) operate correctly in EM0-EM2 (using LFXO) and EM3 (using ULFRCO) selected by the value of the #define LOWEST_ENERGY_MODE.

3. Update your I2C code to implement an interrupt driven solution which sleeps to EM1 while waiting for the I2C transfers to complete. Update your scheduler to return events from the I2C transactions.
    - Be very careful with the usage of LOG_***() calls as these impact the speed of execution of your code. **For the I2C interrupt service routine (ISR), do not call any LOG_***() functions or you will miss your interrupt events and experience malfunctions in the execution of your code**. If you need to debug what is happening in your I2C ISR, build a high-speed event logger. You can then set a break condition (or call __BKPT(0) - this is an inline assembly instruction that will trigger the debugger) so that you can examine the values in your high-speed event log data structure.
4. Craft your state machine to use the events created by the interrupt driven waitTimerUs_irq() and I2C routines.

Files in your src/ directory should look like:



```
src
  ble_device_type.h
  gpio.c
  gpio.h
  i2c.c
  i2c.h
  irq.c
  irq.h
  lcd.c
  lcd.h
  log.c
  log.h
  oscillators.c
  oscillators.h
  scheduler.c
  scheduler.h
  timers.c
  timers.h
```