READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

# 8ECEN 2370
# Embedded Software Engineering Lab #3
# Scheduler and Energy Modes
# Spring 2021

Objective: To learn the use of static / private (or local global) variables and how to access them externally. The concept of a scheduler will be introduced which will be used throughout the remaining course assignments. A scheduler utilizes the concept of a state machine and will make the code easier to develop, test, and debug. Scheduler events will be set via Interrupt Service Routines and the scheduler will be cycled through each time an interrupt occurs and will continue to cycle through until all events have been serviced.

This assignment will explore learning how to manage the lowest possible energy mode that the microcontroller can be in at any time based on the open and/or active peripherals. Managing a database of what the lowest possible energy mode state that the microcontroller cannot enter and maintain proper operation will be critical in future assignments.

Key Learning Objectives for this Assignment:

- Accessing State Information (Static / Private Variables) thru function calls: Most embedded applications require variables to hold state outside of functions. The Interrupt Service Routines (ISR) are functions that are called by an interrupt that has no input arguments or returns a value. To affect the application outside these ISRs, state information is held by private (static) variables. Global variables are not the preferred method in code development due to the ease of the variable being affected unknowingly. For example, in large projects, many developers are writing code for the application. If two developers created the same global variable name aand used it for different purposes, the entire program would not perform as expected.
- Interrupt Service Routines (ISRs): In embedded systems, asynchronous events occur that can disrupt the linear flow of a program. These events are Interrupts. Interrupt Service Routines should be small and quick functions that records the events and if necessary, schedule an event (or callback) to be serviced. In handling the interrupt, the ISR must reset the interrupt flags to enable a future interrupt to trigger and evaluate only the desired interrupts pending.
- Scheduler: A key function of an operating system is scheduling tasks. On your computer, your operating system is switching between tasks to perform functions such as handling internet data, receive an email, perform antivirus checks and many other operations. The scheduler is used to ensure that all tasks have an opportunity to have access to resources such as the CPU. In this course, the scheduler is very simple where in Real-Time Operating Systems, the scheduler is more complex by enabling tasks to be prioritized.
- Energy Modes: Many microcontrollers have different levels of operations to save energy by limiting functionality in these different modes. For a microcontroller, the

highest consuming peripheral is generally the CPU due to the frequency of operation and the amount of circuitry to implement the processor compared to the other peripherals integrated in the microcontroller. The lower energy modes will significantly save more energy by turning off the CPU.

- Managing Energy Modes: To take advantage of the different energy modes, the microcontroller must know what energy mode it can enter and still perform the active tasks. The process of managing the energy mode will be assigned to the peripheral driver and the associated sleep routines. The application developer should not be required to manage the energy mode since the application should be abstracted from the physical device implementation.

Note: You will be using the completed Lab #2 project as a starting point for Lab #3.

Pacing: Completed up to Sleep Mode / Energy Mode section before the 2nd week of lab.

Due: Sunday, February 21st, at 11:59pm

Making change to clarify documentation:

- *Italicized text will signify informational details of the assignment*
- Standard text will signify activities required to complete the assignment

Instructions:

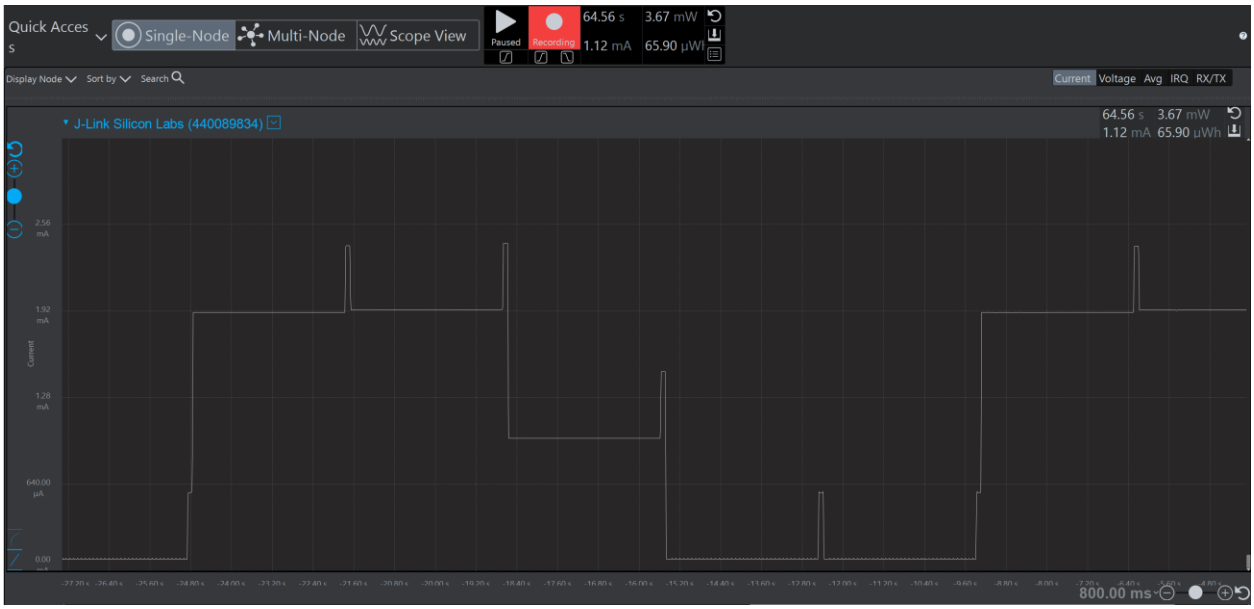
1. Work with the instructing team to get your Lab #2 project fully functional.
    a. The LETIMER period will remain at 1.8 seconds and the active on-time of 0.25 seconds
    b. The HFRCO high frequency clock will remain at 19.0 MHz

2. Change the name of the project by adding your two initials in front and the Lab name, Energy_Mode
    a. Ex. For Keith Graham, the name would change to KG_Enegy_Mode_Lab

3. *In this Lab, the LETIMER0 will continue to be used as a PWM to provide a heartbeat, <u>but it will now be used to generate an interrupt</u> which will be used to change the lowest energy mode that your Pearl Gecko will enter while asleep. When you complete the assignment, you will be able to see the different current consumptions that your Pearl Gecko can enter while in a sleep mode. First, you will enable the LETIMER0 interrupt and develop its Interrupt Service Routine. The next step will be to develop code to implement a simple scheduler. Once the scheduler has been verified to operate correctly, you will develop the routines to keep track of the lowest possible energy mode allowed by the system and the properly put the microcontroller into the correct energy mode while asleep. At the end of this assignment, you will have an Energy Profiler that looks like the below picture.*

# READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

4. *In developing a project, it is best to develop and verify small segments of code. By developing small segments of code, if the code is not working, you have a small area to concentrate your debugging. Most of the time, it will be associated with you new code.*




5. Read this entire document before you begin to develop any code

6. LETIMER0 upgrade to interrupt upon the LETIMER0 counter UNDERFLOWing:

   a. To make the LETIMER0 STRUCT to be versatile, the structure that is used to initialize the LETIMER0 will need to have variables enabling its three interrupts of COMP0, COMP1, and UF.
   b. The application will be defining to the LETIMER0 driver which interrupts to be enabled. The application will notify the driver through the function, app_letimer_pwm_open(), which interrupts to enable and the defined event to be called upon receiving that particular interrupt. This information will be passed to the letimer_pwm_open() function through the APP_LETIMER_PWM_TypeDef argument in the letimer_pwm_open() call.
   c. Add the following variables to the APP_LETIMER_PWM_TypeDef STRUCT found in LETIMER.h.

```
bool        comp0_irq_enable; // enable interrupt on comp0 interrupt
uint32_t    comp0_cb;
bool        comp1_irq_enable; // enable interrupt on comp1 interrupt
uint32_t    comp1_cb;
bool        uf_irq_enable;          // enable interrupt on uf interrupt
uint32_t    uf_cb;
```

      i. Note, cb stands for CallBacK

d. *Next, you will need to begin defining definitions for each individual CallBack for the scheduler to understand what CallBack (event) needs to be serviced. In this simple scheduler, you will define a single unique bit per possible CallBack (event). By defining a single unique bit to each CallBack (event), you are encoding the CallBack so that the scheduler can test a single bit to determine whether a particular CallBack (event) requires to be serviced. With 32-bits declared in an uint32_t variable, the system can use one static / private variable that can support up to 32 possible CallBacks (events) by adding an event with an OR operation and removing an event with a ~ AND operation. For example:*
   i. *Adding the event LETIMER0 COMP0:*
      1. *event_scheduler |= LETIMER_COMP0_CB;*
   ii. *Removing the event LETIMER0 COMP0:*
      1. *event_scheduler &= ~LETIMER_COMP_CB;*

e. You will be defining these CallBacks (events) in app.h since the CallBacks to be scheduled are application dependent. Add the following #define statements in app.h with the other #define statements. You will notice that each of these events has a single unique bit identifying them.

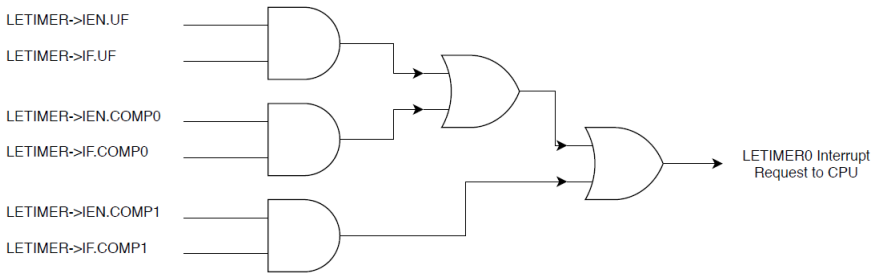
```
// Application scheduled events
#define LETIMER0_COMP0_CB           0x00000001  //0b0001
#define LETIMER0_COMP1_CB           0x00000002  //0b0010
#define LETIMER0_UF_CB              0x00000004  //0b0100
```

   i. *Looking at the binary representation for each CallBack, the comments, you can see that each CallBack is represented by a unique bit*

f. It is time to go into app_letimer_pwm_open() and add these 6 newly created elements of your APP_LETIMER_PWM_TypeDef STRUCT. For this assignment, you will only be using the LETIMER0 interrupt for Underflow
   i. Enable, defined as true, the interrupt for uf_irq_enable
   ii. Disable, defined as false, the interrupt enable for comp0_irq_enable and comp1_irq_enable
   iii. Initialize / configure each of the comp0_cb, comp1_cb, and uf_cb to the appropriate application scheduled event defined in app.h per the previous step

g. These variables will need to be set or defined with the other variables before you call the letimer_pwm_open() function to open/initialize the LETIMER0.

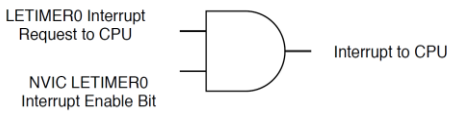
h. *For an interrupt to be generated out of the Pearl Gecko peripheral such as the Under Flow, UF, two conditions must be met. The particular interrupt must be enabled in the Interrupt Enabled, IEN, register, and the source of the interrupt had to occur and set the bit in the Interrupt Flag, IF, register. Below is logic diagram of the LETIMER interrupt circuitry:*




   i. *The first column of logic is AND gates. These gates act as the clock gates in the clock tree. If the associate Interrupt Enable Bit in the IEN register is DEASSERTED, 0, then the output of the AND gate will always be DEASSERTED, 0.*
   ii. *With the output of an AND gate DEASSERTED, 0, these AND gates cannot make the output of the next level of logic, the OR gates ASSERTED, 1, and thus not send a signal to the CPU requesting an LETIMER0 interrupt to be serviced.*
   iii. *Only if both inputs, the specific Interrupt Enable bit and the Interrupt Flag bit, being ASSERTED, 1, will the output of an AND gate become ASSERTED, 1. An AND gate output ASSERTED, 1, will propagate through the OR-gates to send the signal to the CPU requesting the LETIMER0 interrupt to be serviced.*
   iv. *For this lab, you will only be writing a 1, ASSERTED, to the Interrupt Enable, IEN, register for the Underflow, UF, interrupt conditions. With the corresponding bits DEASSERTED for the COMP0 and COMP1 interrupt enable bits, the only interrupt that can be generated in this lab will be the Underflow, UF, interrupt.*

i. Next, modify your letimer_pwm_open() function to enable all interrupts that are set via the APP_LETIMER_PWM_TypeDef input argument to be true and disabled if they the interrupt is defined as false. You are creating a generic driver that can be reused. <u>The rubric will include verifying that all interrupts have been included in the driver.</u>
   i. *Before enabling the interrupts, what is a good practice to do?*
      1. *Clear the Interrupt since the cause of the interrupt will still assert to true even if the interrupt is not enabled*
         a. *The interrupt can be cleared by writing to the Interrupt Flag Clear (IFC) register or reading the appropriate registers that require reading to clear the interrupt source*
      2. *<u>It will be included in this Lab's rubric</u>*
   ii. You can enable the peripheral interrupts using a write directly to the LETIMER Interrupt Enable (IEN) Register or by using the appropriate EM Library routine

iii. If there are interrupts to be set, after setting them, you must enable these interrupts to the Pearl Gecko ARM Cortex CPU by setting true the NVIC interrupt enable bit by calling the below function:
   1. NVIC_EnableIRQ(LETIMER0_IRQn);




j. The architecture for this project is that all Interrupts are serviced through CallBacks which are scheduled through the Scheduler. Interrupts are asynchronous events and thus do not have input arguments. With no input arguments, you will transfer desired information to the Interrupt Service Routine via Private (Static) variables. Later in this assignment you will use the LETIMER0 Interrupt Service Routine (ISR) to schedule the appropriate event for a specific interrupt. Create the following Private (static) variables in letimer.c:

```
//***********************
// private variables
//***********************
static uint32_t scheduled_comp0_cb;
static uint32_t scheduled_comp1_cb;
static uint32_t scheduled_uf_cb;
```

Use the information being passed to letimer_pwm_open() function via the input argument app_letimer_struct to initialize / configure the above Private (static) CallBack variables.

k. With the driver completed, it is time to develop the Interrupt Service Routine. You must add the following function prototype in your letimer.h file. It must be this exact name. The development tools, the Linker, will replace any default function with a user defined function if the names are identical. This development tool feature enables a default Interrupt Service Routine to handle interrupts that are not defined by the user. In the case of this interrupt, I believe the default handler is a while(1) loop.
   i. void LETIMER0_IRQHandler(void);

l. With the Interrupt Service Routine, ISR, function prototype defined, go to letimer.c and develop the LETIMER0_IRQHandler(void) function.
   i. Create the function in letimer.c
      1. Void LETIMER0_IRQHandler(void){

         }
   ii. It is good practice to read the source interrupts as soon as you get into the interrupt handler. Add the following code:
      1. Declare a local variable to store the source interrupts:
         a. Example: uint32_t int_flag;
         b. You are using an uint32_t as this variable due to a register is not a signed integer

2. By ANDing the interrupt source, IF register, with the interrupt enable register, IEN, your interrupt source variable will only contain interrupts of interest
    a. int_flag = LETIMER0->IF & LETIMER0->IEN;
3. Next, clear the interrupt flag register to enable these interrupts to occur again and result in another call to the Interrupt Service Routine request
    a. LETIMER0->IFC = int_flag;

iii. *Each possible interrupt should have its own IF statement to handle its particular interrupt, not an IF ELSE statement.*

1. *Why is it required that each possible interrupt have its own IF statement and not be grouped in an IF ELSE statement?*
2. *What silicon lab enumeration should be used to compare with the saved interrupt source variable, int_flag, for each IF statement?*
    a. *The Silicon Labs builds up of an enumeration of a register bit by the following:*
        i. *Peripheral name with no number at the end such as LETIMER*
        ii. *Append an underscore, _*
        iii. *Register initials such as IF for interrupt flag*
        iv. *Append an underscore, _*
        v. *Bit name such as COMP0*
        vi. *Example: LETIMER_IF_COMP0*
3. Next, within the Interrupt Service routine, for each IF statement, add the following:
    a. To add a check that you are clearing your interrupts within each interrupt source IF statement. Add an assertions statement such as:
    b. EFM_ASSERT(!(LETIMER0->IF & LETIMER_IF_UF));
4. Lack of handling each possible interrupt source will impact your grade. <u>The rubric will include verifying that all interrupts are being handled. You are developing a generic LETIMER0 ISR.</u>

m. Upon completion of the LETIMER0 ISR, it is time to build your code and run it via the debugger. Before you start to run your program in the debugger, place a breakpoint in the LETIMER0 ISR's UF interrupt IF statement at its EFM_ASSERT statement.

i. Is your code stopping at this breakpoint at every LETIMER0 UF interrupt?
    1. If yes, un-pause the debugger

ii. Is your code <u>NOT</u> being stopped at an EFM Assertion statement?
    1. If it is going into an EFM ASSERT, it is time to determine the module and the line item of the EFM_ASSERT and begin debugging

iii. Is your LED still blink as in the last assignment?

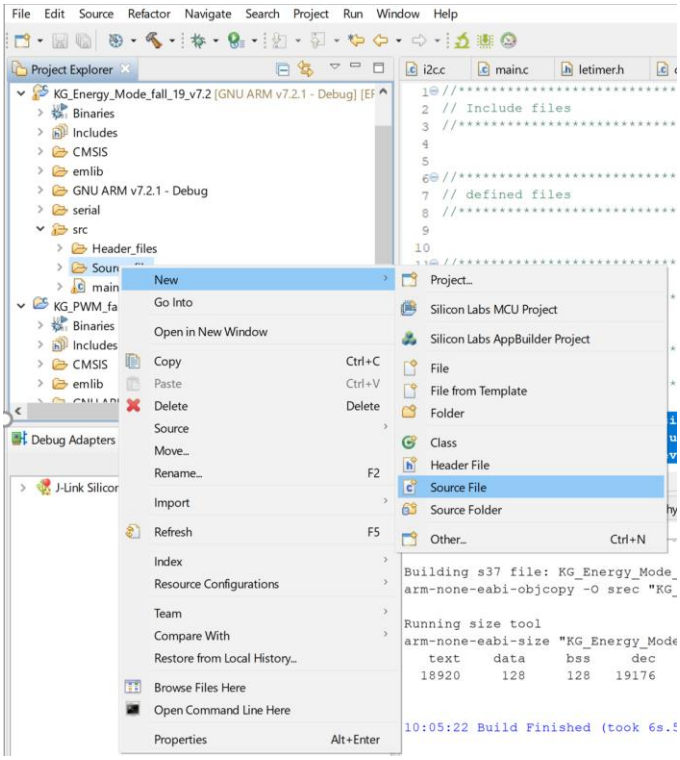
iv. If yes to all of the above, it is time to move to the next part of this Lab.

7. NOTE: The instructing staff will not help regarding the Scheduler or Sleep Mode / Energy Mode portions of this assignment until the LETIMER0 upgrade has been completed.

8. Adding support for a simple scheduler:
   a. Now it is time to begin working on developing a simple scheduler.
   b. Please download from Canvas Lab 3 folder the scheduler.h file. Please add it to your src/Header_files directory.
      i. You can add it by copying the scheduler.h folder from where it was downloaded into your computer
      ii. Select the src/Header_files directory
      iii. Paste the copied file
   c. The scheduler.h files will include all the infrastructure functions for your scheduler which includes the following:

```
void scheduler_open(void);
void add_scheduled_event(uint32_t event);
void remove_scheduled_event(uint32_t event);
uint32_t get_scheduled_events(void);
```

   d. *By adding these function prototypes in the .h file, any other .c file that includes this .h file can access these functions, thus making these functions global*
   e. *These functions allow an external function to scheduler.c to access its static / private (local global) variable though a function call such as get_scheduled_events();*
   f. Now, create a .c file by right clicking your src/Source_files folder, selecting New, and Source file




g. In the popup menu, make sure it is pointing to your src/Source_Files directory and type scheduler.c. Click Finish to complete the addition of the new .c file.

h. Local variables in a function lose state, their value, when the function exits. For the scheduler to maintain the state of pending events, state must be maintained until the event has been processed, so it must be maintained outside of a function. The variable to maintain the state will be a private (static) variable within the scheduler.c file and not directly accessible outside of scheduler.c. This will create better code for portability, reliability, and testing.

   i. Within the newly created scheduler.c file, create the static / private variable:

```
//*********************************
// private variables
//*********************************
static unsigned int event_scheduled;
```

i. In the newly created scheduler.c file, you will need to provide the include statement to it's .h file

   i. Before any #define and static / private variable declarations in scheduler.c, add the following include statements:

      1. #include "scheduler.h"

   ii. Before any #define and function prototypes in scheduler.h, add the following include statements:

      1. #include "em_assert.h"

2. #include “em_core.h”
    a. This include statement is required to access the following functions:
        i. CORE_DECLARE_IRQ_STATE;
        ii. CORE_ENTER_CRITICAL();
        iii. CORE_EXIT_CRITICAL();
3. #include “em_emu.h”
    a. This include statement is required to access the following functions:
        i. EMU_EnterEM1();
        ii. EMU_EnterEM2();
        iii. EMU_EnterEM3();

j. To access this private (static) variable outside of scheduler.c, the application code will need to access them via (global) function calls. These function calls will return the value and/or modify the value of the private (static) variable, event_scheduled. Develop these functions in scheduler.c.
    i. void scheduler_open(void) : Opens the scheduler functionality by resetting the static / private variable event_scheduled to 0.
    ii. void add_scheduled_event(uint32_t event) : ORs a new event, the input argument, into the existing state of the private (static) variable event_scheduled.
        1. *What one line of code will perform this ORing function?*
    iii. void remove_scheduled_event(uint32_t event) : Removes the event, the input argument, from the existing state of the private (static) variable event_scheduled.
        1. *What one line of code will negate or remove a bit(s) from a variable?*
    iv. uint32_t get_scheduled_events(void) : Returns the current state of the private (static) variable event_scheduled.

*k. event_scheduled is a private (static) variable or it could be described as a globe variable within the space of its .c file. Since interrupts are asynchronous, an interrupt could occur in the middle of updating this variable and create an incorrect result. Let’s look at the following example:*
    *i. An event is being cleared in event_scheduled using the following instruction:*
        1. *event_scheduled = event_scheduled & ~clear_event;*
        2. *event_scheduled initially equals 0b0001 and clear event = 0b0001*
    *ii. If after reading the value of event_scheduled in the above c-code and before the & of ~clear_event occurs, and interrupt intercedes and adds the event 0b0100 using the following instruction:*
        1. *event_scheduled = event_scheduled | add_event;*
    *iii. If the Interrupt Service Routine has priority over completing the clearing of the event which is in process, event scheduled will become the value of 0b0101, but when the Interrupt Service Routine returns to the c-line of*

*code to complete event_scheduled = event_scheduled & ~clear_event, the processor will use the event_scheduled read before the interrupt, 0b0001 and not 0b0101. After completing & operation, it will save back event_scheduled 0b0000 losing the event set by the Interrupt Service Routine, 0b0100*

iv. *To prevent a global or shared resource data coherency or timing issue, access private (static) variables should always be atomic especially if they can be altered by an Interrupt Service Routine or an external function. Atomic operations must complete and cannot be interrupted until completion. In the above example, if the clear operation was atomic, the clear would have been completed before the ISR attempt to add the event and thus the result after the ISR operation would be 0b0100. If the ISR operation occurred before the clear event, the initial value of scheduled_event would be 0b0101 and after the clear operation, 0b0100. Both paths through the code resulting in the same correct result.*

v. *To make an operation atomic, before an operation to the static / private or local global variable, interrupts should be disabled using the following instruction:*
    1. *CORE_DECLARE_IRQ_STATE;*
        a. *Declares a variable to save the current state of the global interrupt enable bit*
    2. *CORE_ENTER_CRITICAL;*
        a. *Stores the current state of the PRIMASK, global interrupt enable bit*
            i. *0 = enabled*
            ii. *1 = disabled*
        b. *Calls __disable_irq() which sets the PRIMASK bit to 1 to disable all global interrupts*

vi. *After the completion of accessing the global variable, the atomic operation has been completed, interrupts must be re-enabled to allow proper operation of the system*
    1. *CORE_EXIT_CRITICAL()*
        a. *Reads the value of the saved global interrupt bit that was saved in the variable declared by CORE_DECLARE_IRQ_STATE;*
        b. *If the saved value == 0, enabled, this function will then call __enable_irq() which will clear the bit in the PRIMASK register to enable global interrupts*
        c. *If the saved value == 1, disabled, no action will be taken since the CORE_ENTER_CRITICAL() already will have disabled global interrupts*

vii. *As a complete example, the clearing operation of the event_scheduled private (static) variable should be atomic using this sequence of instructions:*

1. *CORE_DECLARE_IRQ_STATE;*
2. *CORE_ENTER_CRITICAL();*
3. *event_scheduled = event_scheduled & ~clear_event;*
4. *CORE_EXIT_CRITICAL();*

l. Good programming practices do not rely on "Magic Numbers," but on defined constants. Constants/Defines should be in ALL CAPITAL LETTERS to easily distinguish them from variables.

m. With the private (static) event_scheduled variable established and the mechanism to read and modify its state externally, it is now time to develop the event handler.

n. What we do with a CallBack (event) is application dependent, so the CallBack (event) handler for the LETIMER0 event for UF should be placed in app.h/.c and not in our driver files LETIMER0.h/.c.

   i. Create the below function in both app.h and app.c
      1. void scheduled_letimer0_uf_cb (void);
   ii. For now, in this function in app.c, add just one line of code to remove the LETIMER0_UF_CB from the scheduler's private (static) variable, event_scheduled.
      1. What function from scheduler.h should you call to execute this clearing or removing an event from event_scheduled?
      2. What should the function call input argument be?
   *iii. In the event handler, we clear / remove the CallBacks (events) due to that the CallBack (event) is now being processed or serviced. By clearing the event, we are making it available to be called the next time the event is triggered.*

o. The interrupt handlers will be the main source of setting a CallBack (event) to be serviced, but not the only place. In some state machines, the completion processing a state may trigger a following CallBack (event) to occur. The combination of the scheduler and event handlers forces a desired sequence of CallBacks (events) that can only be done in a specified order. By specifying the CallBacks (events) through the scheduler, it prevents timing or data concurrency issues.

   i. Go back to the LETIMER0_IRQHandler(void), and inside the IF statement for each interrupt, call the appropriate scheduler function to set the appropriate event to be serviced for its interrupt in the private (static) event_scheduled variable.
      1. In letimer_pwm_open(), you configured the private (static) variables that define the event for each of the possible interrupts
      2. Use the appropriate private (static) variable for each of the interrupts to set the event_scheduled variable through the add_scheduled_event() function
   ii. Upon exiting the interrupt handler, you program will go back to the scheduler and process all desired events.

p. At this point, CallBacks (events) are now scheduled through the Interrupt Service Routine and CallBacks (events) are processed or serviced through CallBacks (event) handlers called from reading / decoding the CallBacks (events) from event_scheduled in the scheduler.c module.

q. *The scheduler works in conjunction with the main.c's while (1) loop. The order of operations will work in the following order if the Pearl Gecko is asleep:*
   i. *An interrupt will wake up the processor by going into its Interrupt Service Routine, ISR*
   ii. *Upon exiting the ISR, it will go to the next instruction after the EMU_EnterEM2(true) line of code in main.c.*
   iii. *The line of code after EMU_EnterEM2(true) is where you will be adding code to check whether an event needs to be serviced.*
   iv. *Upon checking and returning from all events, the while(1) will loop back and the system will re-enter sleep mode using EMU_EnterEM2(true);*

r. After the call to enter sleep, EMU_EnterEM2(true), add an If statement to check whether the LETIMER0_UF_CB event has been set in the private (static) variable event_scheduled. If the event has been set, the If statement should call the event function in app.c to service the interrupt/CallBack/event.
   i. *What is the function to service the LETIMER0_UF_CB event?*

s. Now let's add some EFM_ASSERT statements to test out the scheduler. Go to your LETIMER UF event handler that you wrote in app.c.
   i. Add the following assert statement as the first line of code in this function before calling the clearing event function in scheduler.c. This assert will verify whether the event handler was entered due to the proper event.
      1. EFM_ASSERT(get_scheduled_events() & LETIMER0_UF_CB);

t. Before we complete this section of the assignment, you will add two more events and event handlers. These events are LETIMER0_COMP0_CB and LETIMER0_COMP1_CB.
   i. In the scheduler section of main.c while(1), add two more events to be checked. One for each of these two events and call their associated event handlers.
   ii. *Should these checks for events be using separate IF Statements or an IF/Else statement?*
   iii. Similar to the event handler for LETIMER0_UF_CB, these two event handlers should be in app.h/.c due to being application specific.
   iv. At this time, the application does not use the LETIMER0 COMP0 and COMP1 interrupts. Inside these event handlers, add the following code:
      1. Clear or Remove the associated event
      2. EFM_ASSERT(false);
         a. The "false" statement will always return a 0 and thus the Assert will fail and your code will enter a while(1) loop.
         b. If your code enters one of these EMF_ASSERTs, you can surmise that you have not properly initialized the

LETIMER0 interrupts via initializing the data structure or the code to open the LETIMER0.

u. Last item before you try out the scheduler. You defined and created a function to reset the static / private variable event_scheduled. It is good practice to initialize your variables.
   i. Add a call to scheduler_open() to initialize the state of the scheduler.
   ii. What is a proper location to add this function call in app.c?

v. Before you run the program, place a breakpoint in the LETIMER0 UF CallBack (event) service routine on the c-line of code used to clear or remove the LETIMER0_UF_CB from the static / private variable event_scheduled in app.c.

w. Now, run your program in debug mode, does the program repeatedly break at the line of c-code to clear the event in the LETIMER0 UF event service routine?
   i. If no, pause the program and determine if one of the EFM_ASSERTs has occurred.
   ii. If yes, its time to go to the third part of Lab 3, adding sleep mode functions

9. NOTE: The instructing staff will not help regarding the Sleep Mode / Energy Mode portions of this assignment until the LETIMER0 upgrade and Scheduler code sections have successfully been completed.

<<< GOAL TO HAVE COMPLETED UP TO HERE BEFORE THE START OF THE 2nd WEEK OF LAB >>>

10. Sleep Mode / Energy Mode:
    a. These functions are based from Silicon Labs' emlib library sleep.c function. Instead of using the Silicon Labs routine, you will create your own version of them.
    b. Create sleep_routines.h file in your src/Header_files folder. Use the same process as used to create a .c file, but instead of source file, select Header File.
    c. You may want to copy the format used in another .h file as a guideline for this new sleep_routines.h file. Add the following .h includes in your sleep_routines.h file.
       i. The sleep_routines.h should include all the include statements that are directly associated to implement sleep_routines.c
          1. #include "em_emu.h"
             a. Enables access to the calls to the different sleep routines such as EMU_EnterEM1(); or EMU_EnterEM2();
             *b. Please note that these routines begin with EMU and thus you are required to include em_emu.h include statement. EMU stands for Energy Management Unit.*
          2. #include "em_int.h"
          3. #include "em_core.h"
             a. This include statement is required to access the following functions:

i. CORE_DECLARE_IRQ_STATE;
ii. CORE_ENTER_CRITICAL();
iii. CORE_EXIT_CRITICAL();

b. Core interrupt handling API

4. #include “em_assert.h”
   a. Enables the use of the EFM_ASSERT() statements

ii. In this sleep_routines.h file, you must define all the function prototypes required to support maintaining the state of the lowest possible sleep mode based on active Pearl Gecko peripherals.
   1. void sleep_open(void) : Initialize the sleep_routines static / private variable, lowest_energy_mode[];
   2. void sleep_block_mode(uint32_t EM) : Utilized by a peripheral to prevent the Pearl Gecko going into that sleep mode while the peripheral is active.
   3. void sleep_unblock_mode(uint32_t EM) : Utilized to release the processor from going into a sleep mode with a peripheral that is no longer active.
   4. void enter_sleep(void) : Function to enter sleep
   5. uint32_t current_block_energy_mode(void) : Function that returns which energy mode that the current system cannot enter.

d. Good programming practices do not rely on “Magic Numbers,” but on defined constants. Constants should be in ALL CAPITAL LETTERS to easily identify them from variables. In sleep_routines.h, define the following constants:

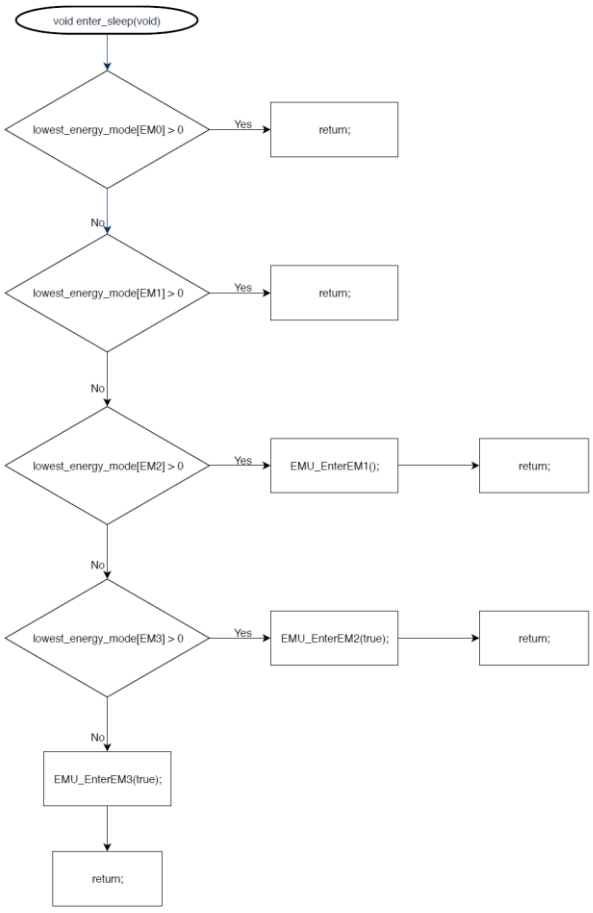
```
#define     EM0                 0
#define     EM1                 1
#define     EM2                 2
#define     EM3                 3
#define     EM4                 4
#define MAX_ENERGY_MODES        5
```

e. Now, go and create sleep_routines.c in your Source_Files folder.
f. Similar to sleep_routines.h include statements are required in sleep_routines.c, you will need to include all of the .h files required by the sleep routines. These include statements are:
   i. #include “sleep_routines.h”
   ii. Note that you are only including one .h file since the reference to sleep_routine.h associates the following .h files with the sleep_routines.c file:
      1. em_emu.h
      2. em_int.h
      3. em_assert.h
      4. em_core.h
g. In sleep_routines.c, you must define a static / private variable, an array, to maintain state of which state that the Pearl Gecko cannot go into based on its

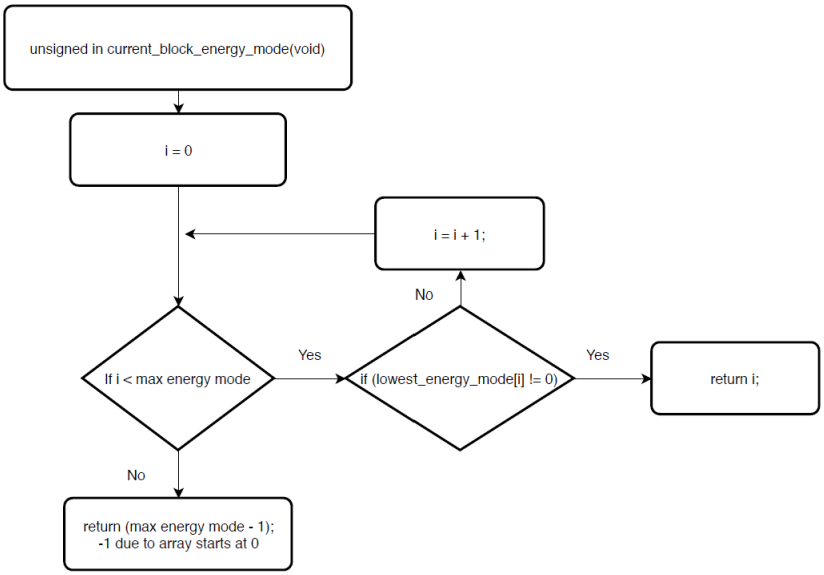
current active peripherals. This array will be type int, integer, to help determine whether there were more unblock function calls to block function calls. Further details will be discussed shortly.

```
//***********************************************
// private variables
//***********************************************
static int lowest_energy_mode[MAX_ENERGY_MODES];
```

h. With sleep_routines.h created and the static / private variable to hold state, it is now time to go and develop the routines defined as prototype functions in sleep_routine.c.
   i. For the enter_sleep() function, below is a flow chart of its functionality




   ii. Below is a flow chart for the current_block_energy_mode() function




iii. Refer to the lecture slides if required to help develop the energy mode sleep functions

i. To help debug now or in the future, add the following EFM_ASSERT statements.
   i. A release of a sleep mode should only occur once for every set a block sleep mode. With one release for every set, the state of any sleep mode should never go below 0. At the end of the function sleep_unblock_mode(), add the following EFM ASSERT
      1. EFM_ASSERT (lowest_energy_mode[EM] >= 0);
   ii. Similarly, we will be using a small amount of peripherals. A sleep mode state should never become too large. If a peripheral is setting more block mode sets than unblocks, one of the sleep mode state variables could continue to increase and never become 0 again due to the uneven number of blocks versus unblocks for the given peripheral. Add the following EFM ASSERT at the end of the function sleep_block_mode():
      1. EFM_ASSERT (lowest_energy_mode[EM] < 5);

j. The IP used to create these functions came from Silicon Labs. You must give credit to the developer of the IP. The sleep.c functions that these routines IP came from has the following statement which must be included in your sleep_routines.c file.
   i. Include the below IP statement in your doxygen file comment block

# READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

```
/***************************************************************************
 * @file sleep.c
 ***************************************************************************
 * @section License
 * <b>(C) Copyright 2015 Silicon Labs, http://www.silabs.com</b>
 ***************************************************************************
 *
 * Permission is granted to anyone to use this software for any purpose,
 * including commercial applications, and to alter it and redistribute it
 * freely, subject to the following restrictions:
 *
 * 1. The origin of this software must not be misrepresented; you must not
 *    claim that you wrote the original software.
 * 2. Altered source versions must be plainly marked as such, and must not be
 *    misrepresented as being the original software.
 * 3. This notice may not be removed or altered from any source distribution.
                                                                *
 * DISCLAIMER OF WARRANTY/LIMITATION OF REMEDIES: Silicon Labs has no
 * obligation to support this Software. Silicon Labs is providing the
 * Software "AS IS", with no express or implied warranties of any kind,
 * including, but not limited to, any implied warranties of merchantability
 * or fitness for any particular purpose or warranties against infringement
 * of any proprietary rights of a third party.
 *
 * Silicon Labs will not be liable for any consequential, incidental, or
 * special damages, or any other relief, or for any claim by any third party,
 * arising from your use of this Software.
 *
 ***************************************************************************/
```

k. Similarly, to the private (static) variable, event_scheduled, in scheduler.c, the private (static) array, lowest_energy_mode[], in sleep_routines.c must be protected due to external events such as Interrupt Service Routines can modify its value by making accesses to this array atomic.
    i. For the functions sleep_block_mode() and sleep_unblock_mode(), before updating the private (static) array, disable all interrupts before and re-enable afterwards

l. For the enter_sleep() function, we will also need to make the function atomic to protect the access and use of the lowest_energy_mode[] array.
    i. Add the highlighted yellow lines of code in your enter_sleep() function

```
void enter_sleep(void) {
    CORE_DECLARE_IRQ_STATE;
    CORE_ENTER_CRITICAL();
    .
    .
    .
    (Desired Function Code)
    .
    .
    .
    CORE_EXIT_CRITICAL();
```

    ii. Below is a description how these CORE CRITICAL functions work within this enter_sleep() routine.
        1. Between the CORE_ENTER_CRITICAL() and CORE_EXIT_CRITICAL() functions, interrupts will not interrupt the execution of the code but will wake up the processor if an interrupt occurs. It will allow the instructions between the call to sleep such as EMU_EnterEM2() and CORE_EXIT_CRITICAL() to execute before jumping to the Interrupt Service Routine. The interrupt will occur upon completion of the CORE_EXIT_CRITICAL() function which re-enables the interrupts.
        2. If an interrupt occurs after CORE_ENTER_CRITICAL() and before the sleep call such as EMU_ENTEREM2(), the CPU will automatically wake up due to the pending interrupt.

m. You defined and created a function to reset the static / private variable lowest_energy_mode[]. It is good practice to initialize your variables.
    i. Add a call to sleep_open() to initialize the state of the lowest energy mode state variable.
    ii. What is a proper location to add this function call in app.c?

n. With the sleep routines developed, the proper practice of calling sleep_block_mode() and sleep_unblock_mode() must be established for all of the peripherals. In this assignment, only one peripheral is defined, LETIMER0.
    i. It is the driver's responsibility to set the energy mode that the microcontroller can go and not the application. The application is defining what the LETIMER0 does, but not its interaction with the microcontroller's hardware
        1. With the responsibility residing with the driver accessing the sleep_routines.c functions and its defined statements, you must provide the sleep_routines.h include statement in letimer.h
        2. For readability and the use of no magic numbers, in letimer.h, create a definition of the energy mode that LETIMER0 cannot, block, enter while using the ULFRCO oscillator

```
#define LETIMER_EM    EM4 // Using the ULFRCO, block from entering Energy Mode 4
```

        3. EM4 is a definition defined in the sleep_routines.h file earlier in this lab
    ii. Every time that a peripheral is enabled or turned-on, a call to sleep_block_mode() must be called to prevent from going into an energy mode that the microcontroller cannot enter while this peripheral is active. In this lab, there are two locations that the LETIMER0 can be enabled.
        1. letimer_pwm_open()
        2. letimer_start()
    iii. In letimer_pwm_open(), your driver open function should check whether LETIMER has been enabled at the very end of the function. If it is enabled and running, then a call should be made to establish the energy mode that the microcontroller cannot enter while the LETIMER is active
        1. sleep_block_mode(LETIMER_EM);
        2. You can verify whether the LETIMER is enabled/running by checking the RUNNING bit in its STATUS register
    iv. Similar to the driver open function, if a call to letimer_start() to enable or turn-on the LETIMER, a call should be made to establish the energy mode that the microcontroller cannot enter while the LETIMER is active only if the LETIMER was not enabled at the start of this function
        1. This IF statement has another condition than whether you are enabling the LETIMER. A call to set the energy mode to block should only be made once, the time that the peripheral changes from inactive to active state. In letimer_start(), only if the LETIMER is not RUNNING and enabled should a call to the block mode sleep routine be called
            a. sleep_block_mode(LETIMER_EM);

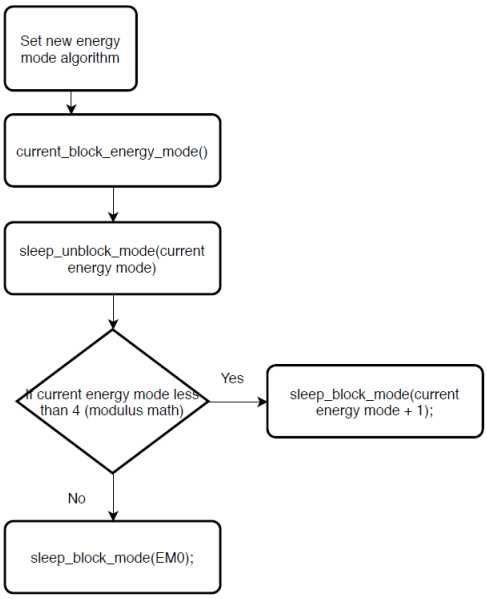
v. Inversely to when the peripheral is enabled or becomes active that a call to block the energy mode is made, when a peripheral is disabled or becomes inactive, a call to unblock the energy mode must be made
    1. We are going to assume for now that the driver was disabled before a call to open the driver was made, so only the case of the LETIMER becoming active will occur in the driver
    2. In letimer_start(), an IF statement must check whether a call to this function is disabling the LETIMER. Similar to only calling the block sleep mode when a peripheral is going from inactive to active state, the unblock sleep call should only be called once when the LETIMER is going from active to inactive state. This requires the IF statement to include checking the status of RUNNING and being disabled to remove its hold on the energy mode to block
        a. sleep_unblock_mode(LETIMER_EM);
    3. Let's add one more piece of functionality to the letimer_start() function. The call to the LETIMER_Enable() software library writes to the LETIMER CMD register to turn-on or turn-off the LETIMER. Before exiting the two if statements, let's guarantee that the LETIMER_Enable() operation completes by adding a stall while statement until the LETIMER has synchronized the CMD register write.
        a. while(letimer->SYNCBUSY);
        b. To save time and energy, you only want to stall if a change is occurring in LETIMER0 state, becoming enabled or becoming disabled

vi. A driver should handle the case if it is called to open even if the driver is already open. The definition of how the driver handles this situation must be defined by the developer. The driver open function may just modify the current peripheral with the new open call or it may return an error. In our case, we will have the driver modify the current operation of the peripheral.
    1. It is best practice to disable a peripheral before reprogramming it. You have modified letimer_start() function to properly turn-off the LETIMER as well as release its block on the energy mode that the system can go into while asleep.
    2. Add a call to disable the LETIMER after you have enabled the LETIMER0 clock and before the EFM_ASSERT() to verify the clock has been enabled
        a. letimer_start(letimer,false);

o. With the sleep routines developed and adding the proper LETIMER calls to block and unblock sleep modes, it is time to try them out. For this assignment, every time that you enter the LETIMER0 UF event service routine, you will cycle

through the energy modes defining what the lowest energy mode that your sleep routine will enter.

i. Read the current lowest energy mode setting
ii. Unblock/remove this current energy mode setting
iii. If the current energy mode block is less than 4, call the function to block/set the energy mode the current energy block mode plus one
iv. Else, if the current energy mode block is 4, set the energy mode to block to EM0
v. Below is a Flow Chart of the above written description




vi. Is the Flow Chart or written description more helpful?

p. One more thing to do before we try the program out. We must modify the main.c while (1) loop to support the sleep function with the scheduler. If we are looping back to the EMU_EnterEM2(true); function, two changes must be made.
   i. If an event is still pending, you cannot enter sleep
   ii. And, instead of EMU_EnterEM2(true), you must call your sleep routine
   iii. Change EMU_EnterEM2(true) to …
      1. if (!get_scheduled_events()) enter_sleep();

q. The return from the get_scheduled_events() function is the value of a static variable that can be changed via an asynchronous event such as an interrupt from a Pearl Gecko peripheral. Since the effective value returned could change once the function exits, the expression that utilizes the return value must be protected by making the expression atomic.
   i. Protect, make atomic, the if statement to evaluate whether there are no events scheduled and if yes, call the enter_sleep() function.

r. Now run the program in Energy Profiler. It should look similar to the below snapshot. <u>The time and current values in the snapshot are not representative of Lab 2.</u>


s. Why does it appear that Pearl Gecko is sleeping in EM0 for two adjacent periods?

11. Each function in scheduler.c, sleep_routines.c, app.c, cmu.c, gpio.c and letimer.c, should have a proper doxygen file and function comment block fully describes what the function performs, the input arguments, and outputs (return). Below is an example from a Silicon Labs embedded library function.

```
/***************************************************************************//**
 * @brief
 *   Calibrate the clock.
 *
 * @details
 *   Run a calibration for HFCLK against a selectable reference clock.
 *   See the reference manual, CMU chapter, for more details.
 *
 * @note
 *   This function will not return until the calibration measurement is completed.
 *
 * @param[in] HFCycles
 *   The number of HFCLK cycles to run the calibration. Increasing this number
 *   increases precision but the calibration will take more time.
 *
 * @param[in] ref
 *   The reference clock used to compare HFCLK.
 *
 * @return
 *   The number of ticks the reference clock after HFCycles ticks on the HF
 *   clock.
 ******************************************************************************/
uint32_t CMU_Calibrate(uint32_t HFCycles, CMU_Osc_TypeDef reference)
{
```

READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

# READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

Deliverables:

1. Each person exports their project as an archived filed. Upload the archived project .zip file into the Lab 3 Canvas Exported Project assignment
2. Each person generates their Lab 3 doxygen html folder, zips the folder, and uploads the zipped doxygen folder into the Lab 3 Canvas Exported Project assignment
3. Each person to provide their answers via Canvas/quiz/Lab3 Worksheet

Questions:

You must complete the Lab 3 worksheet to complete this assignment.

Point breakdown:

- Lab 3 has a total of 35 points
- Exported project will be graded on:
    - Functionality
        - Program enters all possible energy modes EM0 thru EM3
        - Period of switching energy modes = LETIMER period
        - Program sequences thru the proper sequence of energy mods
    - Proper developed scheduler routines
    - Proper developed sleep routines
    - Correct operation of the LETIMER0 interrupt handler
    - Program cycles through Energy Modes 0 – 3
        - Verified via the Energy Profiler
    - Proper commenting of functions
        - cmu.c, gpio.c, app.c, letimer.c, scheduler.c, sleep_routines.c
    - Silicon Labs IP statement in sleep_routines.c and sleep_routines.h
    - No use of magic numbers
    - Best coding practices
- Partial credit will be given on code that is not completely functional

Late Penalty deduction:

- Exported program
    - Due day + 1 day — max score is 30 pts
    - 1 day late to 3 days late — max score is 25 pts
    - 3 days late to 5 days late — max score is 20 pts
    - After 5 days late — max score is 0 pts