ECEN 2370
Embedded Software Engineering Lab #5
LEUART driver – BLE NCP
Fall 2020

Revision 1.0: Revisions are highlighted in GREEN.

i. Due to the frequency of transmitting to the LEUART, 2.7 seconds, we do not want to go into Energy Mode 3 due to the time to turn on / stablize the LFXO oscillator is 1.0 seconds. Add a new energy mode block define in app.h
    1. #define SYSTEM_BLOCK_EM EM3
ii. Add a call to block_sleep_mode(SYSTEM_BLOCK_EM); to your app_peripheral_setup(void) function

Objective: To learn how to develop a UART/LEUART driver that supports the following design goals:

- Modular
- Encapsulated
- Interrupt Driven State Machines
    - Low Energy Operation
    - Supports Multiple Tasks

The UART peripheral is a simpler communication protocol than I2C. UART interfaces are normally point to point data links, so no protocol is required to address which device that the Pearl Gecko would like to communicate when it requests a read or write on the UART communication bus. The UART protocol also does not require the slave or end point to acknowledge receipt of data on a write. With a write to a UART end point not requiring any acknowledgement, data dependency, with the end point, a SW ladder chart with an external device is not required but it still can be very useful. This assignment will begin by developing a software flow chart between the Pearl Gecko UART driver and the UART peripheral. Before beginning any programming for this assignment, a Software Flow Chart should be developed. Proper planning of the Software will minimize development, testing, and debug.

An example of a hardware dependency that must be taken accounted for in the Software Flow Chart is where the UART TXDATA, transmit buffer, is not available if the TXBL interrupt line or status bit is DEASSERTed. Similarly, a string of data has not completed its transmission until all the characters have been sent to the TXDATA buffer, moved to the Transmit Shift Register, and then sent out over the UART TX line.

When would you need to Block Sleep Mode since the LEUART can only operate down to EM2 energy mode? When would you unblock sleep mode the active LEAURT operation?

With the LEUART peripheral running at a lower frequency than the Pearl Gecko CPU core, software synchronization will be required between your driver and the low frequency LEUART peripheral.

The LEUART0 will be connected to a BLE, Bluetooth Low Energy, NCP, Network Co-Processor. A Network Co-Processor is a very simple way of communicating via a radio. The NCP handles the software communication stack and the application on the Pearl Gecko just transmits the data to the NCP for transmission or receipt of data from it.

In this assignment, you will add functionality to Lab 4, so all of Lab 4 functionality will remain. Every time that the Si7021 takes a temperature measurement, it will send data to the BLE module to transmit the temperature to your phone. When the temperature rises above 80F, LED1 will turn-on indicating a high temperature and will turn-off when the temperature returns below 80F. The temperature will be sensed once every time the LETIMER0 underflows. Low-Energy software methodology will always be used to keep the micro-controller at the lowest possible energy mode.

In this assignment, we will further explore developing test strategies before implementing the assignment code.

The rubric for this assignment will also include running your project with optimization -02. For development, continue to use optimization -O0. The debugger will show a clearer relationship of a breakpoint to the line of C-code due to no rearrangement of the assembly code in relation to the C-code. With optimization enabled, the linear direct relationship of the assembly code to c-code begins to breakdown. If you are not using code that would be optimized out like the below for loop delay, your code should operate successfully with optimization enabled.

```
for (int j = 0; j < 1000; j++);
```

Key Learning Objectives for this Assignment:

- Universal Asynchronous Receive Transmit (UART) peripheral: The UART interface is one of the three most common digital communication buses to connect sensors and small actuators to a microcontroller. This bus protocol is very different than the I2C protocol from Lab 4 in that the UART interface is generally a point to point communication bus that does not require a master sending an address to specify which sensor to respond to the request. With the addition of an RS-232 driver which changes the voltage range from 0-3.3v to -12 to +12v, the UART via RS-232 is an effective communication protocol out of an embedded system using cabling.
- Test-Driven Development (TDD) Methodology: Developing a process that indicates errors in your project code will reduce the overall debug and thus development time. Having a unit test developed before developing the project code that checks small segments of code reduces the debug effort by focusing on a small number of lines of code. The Test-Driven Development Methodology provides two additional key benefits

to the developer and overall project. First, the Test-Driven Code is designed to validate system requirements and not a specific implementation. By testing the system requirements, once the developer's code passes the TDD test, the code developed will most likely pass the project's overall system verification and validation. The second key benefit, the TDD code should be designed to remain in the system while code is being developed. These unit test become regression tests that will indicate an error if another piece of code being developed results in a failure of a previously completed code segment.

Note: You will be using the completed Lab #4 project as a starting point for Lab #5.

Checkpoint: By lab time on Tuesday, October 27th

Due: Sunday, November 1st, 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:

2. Work with the instructing team to get your Lab #4 project fully functional.

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

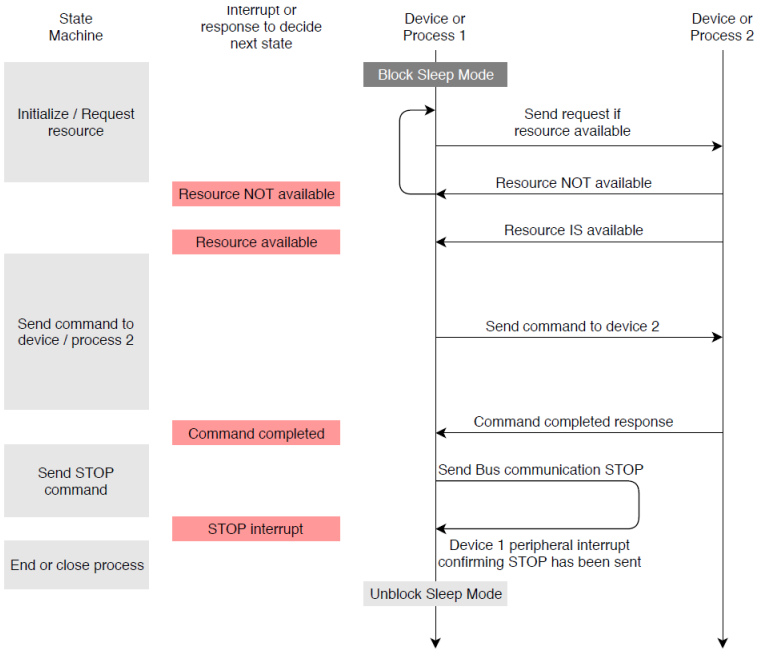
4. Software flow Chart:
    a. *Software flow charts are extremely helpful in the planning and organization of your software code.*
    b. Develop a Software Flow Chart for the state machine that will be used for the driver in leuart.c to transmit a string of characters to the LEUART peripheral
    c. *A Software Flowchart is required for this Lab. If you are using a Ladder Flow Chart, the left "side-rail" of the SW ladder will be your program, the Pearl Gecko, and the right "side-rail" of the ladder will be the Pearl Gecko's LEUART0 peripheral.*
    d. *The rungs of the ladder indicate data transfer from one device, "side-rail," to the other*
        i. *These rungs indicate dependencies between the devices*
        ii. *The other device cannot proceed until the dependency from the other device has been resolved*
    e. *If an iteration or loop is required, it should be indicated by a looping arrow*

f. *This flowchart must also include your calls to block sleep mode and unblock sleep mode function calls as well as any assumptions before the state machine begins and any re-initializing / resetting static variables at the end.*
g. *In summary, the Software Flow Chart should cover the following:*
    i. *What assumptions are made such as to initialize the LEUART peripheral*
    ii. *Starts with a call to leuart_start() to initiate a write of a string of characters from the Pearl Gecko to the BLE module*
    iii. *Check for availability to transmit data*
    iv. *Set Leuart is busy*
    v. *Transmit a character*
    vi. *….*
    vii. *….*
    viii. *….*
    ix. *Determine end of string transmitted*
    x. *Determine when data is finished transmitting to clear Leuart busy*
    xi. *When to set LEUART CB DONE event*
    xii. *When to unblock sleep mode*
    xiii. *Comment block on any items that must be re-initialized to prepare for next LEUART transmit request*
h. Important items to note for your LEUART flow chart regarding handling its interrupts
    i. The TXBL interrupt occurs anytime the LEUART TX_Buffer is empty. You will get LEUART0 TXBL interrupts continuously while the buffer is empty, so the TXBL interrupt must be disabled until you are ready to begin transmission of the string using the interrupt handler. This means that TXBL must be disabled by default, and only enabled once your state machine for LEAURT is ready to GO.
        1. With the TXBL needing to be enabled at a specific time in the sequence of the state machine, you must include enabling this interrupt in the proper place of your flowchart
    ii. Similarly, once you transmit the last byte of the string to the leuart's TXDATA buffer, if you do not disable the TXBL interrupt, you will continuously enter the TXBL interrupt handler even if you have no data to transmit which will prevent your code from going to sleep
        1. Once you have transmitted the last byte of data, the TXBL interrupt should be disabled
        2. You must include the disabling of the TXBL interrupt in your flow chart
    iii. Now, let's look at the TXC interrupt.
        1. At what state do you want to enable the TXC? Or, put another way, when do you care about the TXC interrupt?
        2. And, when do you not want to look at TXC?
        3. The enabling and disabling of the TXC interrupt must be included in your flow chart

i. It is required in the software flow chart to specify where you would call block_sleep_mode() to ensure that the LEUART0 will continue to operate while the CPU is asleep and unblock_sleep_mode() when the driver can release its block hold

j. In addition, for your software flow chart, you must add the state machine's states and the monitored interrupts to the flow chart like the example below

k. The SW ladder chart is a flow chart of a state machine



l. There are many software packages that you can use to create your Software Ladder Chart including Power Point. One tool to create charts can be found at:

i. https://www.draw.io/

5. NOTE: The instructing staff will not help with software coding questions until the Software Ladder Chart has successfully been completed.

6. Time to develop your project's framework

a. First, you will develop the internal Pearl Gecko physical connections required for the assignment

i. Establish the LEUART0 clock tree

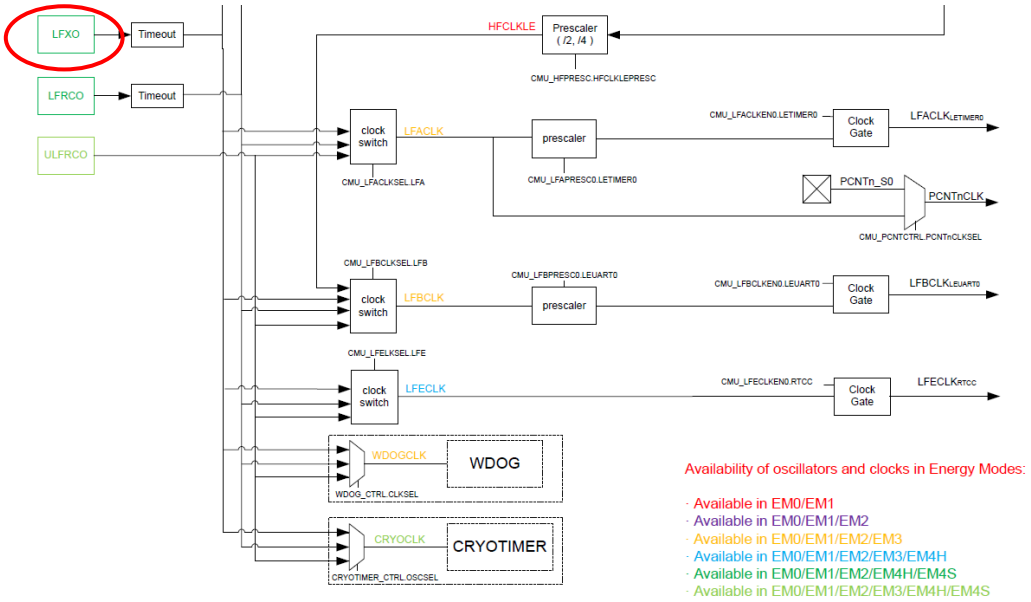
ii. Make the physical connections from the LEUART0 to the external pins of the Pearl Gecko

7. cmu_open() function:

a. For the LEUART0, we will use the LFXO oscillator to enable operation down into the Pearl Gecko Energy Mode 2 level.

i. Why would you use an external crystal, the LFXO, and not use the internal RC oscillator, LFRCO, if they both have the same nominal frequency? They both work down into EM2 energy mode as well.

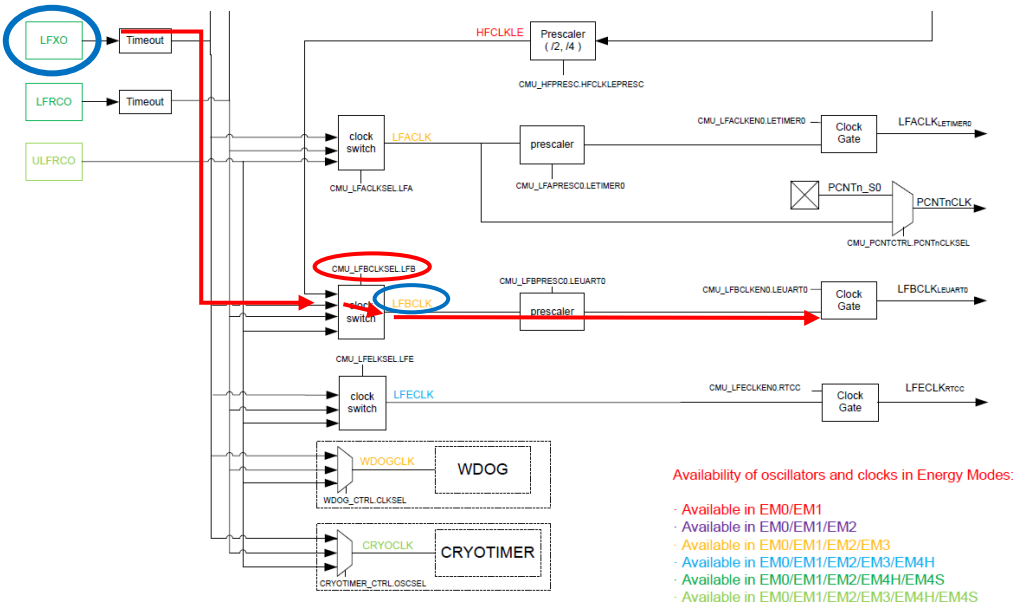
ii. With the Pearl Gecko being a Low Energy micro-controller, by default, the crystal oscillators are disabled. The first activity in establishing the clock tree is to enable, turn-on, the desired oscillator, LFXO.



iii. You can enable / turn-on the oscillator by using the following Silicon Labs em library routine. This must be done in your cmu_open() function.
   1. CMU_OscillatorEnable()

iv. In building the clock tree, the next step is to route the oscillator to the desired clock branch. This routing is done through selecting the proper input of the clock switch mux.

v. The setting of the clock switch branch mux select lines, CMU_LFACLKSEL_LFB, for the LEUART0 is performed using the following emlib routine
   1. CMU_ClockSelectSet()
   2. Using the Silicon Labs Hardware Abstraction Layer, HAL, document, what clock branch “definition” would you use to specify the LFBCLK branch?



8. gpio_open() function:
    a. The BLE module will be connected to the Pearl Gecko STK3402 starter kit via the expansion connector

    b. The BLE module has four pins that must make connection with the Pearl Gecko starter kit via the expansion connector:
        i. VCC – connect to the Pearl Gecko 5.0v power pin (5v) on the expansion connector. Using this power pin to power the BLE will separate it from the power plane that measures the current of the Pearl Gecko, VMCU (Voltage Micro-Controller Unit). Separating it from the Advance Energy Mode power plane enables better observation of the current consumed by your software code that you are developing for the Pearl Gecko micro-controller.
            1. *The HM-18 data sheet that includes the "AT" commands is the actual radio module specification which is located in the middle of the board level module that defines the radio module supply as:*

- Power: +1.9~3.7 VDC 50mA

2. *But, the complete DSD TECH module specification is 3.6-6.0v. The module has a Low Drop Out regulator that takes the input voltage and converts it down to 3.3v. This enables the module to work with Arduino systems that provide 5.0v. I have verified using an oscilloscope that the output of the DSD TECH module to the Pearl Gecko board is 3.3v which matches the voltage of VMCU, Voltage of Micro-Controller Unit, 3.3v.*

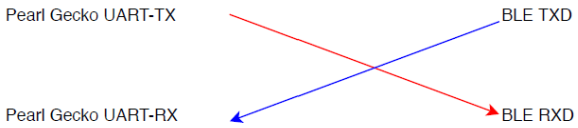
ii. GND – connect to GND, ground, pin

iii. TXD – should you connect it to the UART-RX or UART-TX pin?

1. For UARTs, the RX or TX designator is in reference to the device pins. For instance, the UART-TX is the Pearl Gecko's transmit while the TXD of the BLE is in reference to the BLE's transmit pin. For proper configuration, the Pearl Gecko UART-TX must be connected to the BLE receive.

iv. RXD – should you connect it to the UART-RX or UART-TX pin?

1. For UARTs, the RX or TX designator is in reference to the pin's device. For instance, the UART-RX is the Pearl Gecko's receive while the RXD of the BLE is in reference to the BLE's receive pin. For proper configuration, the BLE module transmit must be connected to the Pearl Gecko UART-RX.



c. *NOTE: When you look at the backside of the Pearl Gecko STK3402 start kit board, you will see all the pin definitions of the expansion connector. The row of pins closest to the edge of the printed circuit board are the top row of pins while the inner most pin definitions refer to the lower row of pins on the expansion connector when the board is faced up.*

d. You will need to obtain female-to-female jumper wires to connect the BLE module to the proper pins of the Pearl Gecko expansion connector

e. Similar to the Si7021, the BLE module via the expansion connector is routed to particular pins on the Pearl Gecko starter kit which results in hardware dependences. The first task is to define these hardware dependences. The definitions should be placed in your brd_config.h file. You will need to use the

DSD TECH HM-18 data sheet, Pearl Gecko's Data Sheet, Schematic, and/or Starter Kit User guide to define these dependencies. Where appropriate, you will want to use the Defined Statements / Enumeration found in the Pearl Gecko Hardware Abstraction Layer, HAL, online documentation.

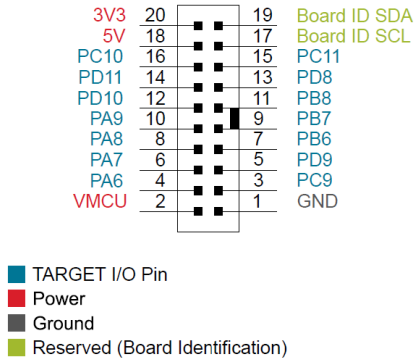


Figure 4.2. Expansion Header

Table 4.3. EXP Header Pinout

Pin	Connection	EXP Header function	Shared feature	Peripheral mapping
20	3V3	Board controller supply		
18	5V	Board USB voltage		
16	PC10	I2C_SDA	SENSOR_I2C_SDA	I2C0_SDA #15
14	P[redacted]	UART_RX		LEU0_RX[redacted]
12	P[redacted]	UART_TX		LEU0_TX

f. The above figure is from the Pearl Gecko STK3402 starter kit. You can open this document to determine the associated Pearl Gecko port and pin information for both the LEUART_RX and TX pins

g. Use the port and pin information to create definitions in brd_config.h associated with the LEUART_RX and LEUART_TX signals

h. In gpio_open() function, you will need to enable the proper configuration for the LEUART_RX and UART_TX signals

   i. LEUART_TX:

      1. TX refers to transmits and thus refers to an output signal. The UART pins can only be output or input pins as well as they are point to point connection. There is no requirement to support multiple devices on a communication bus which is in an AND configuration.

         a. For a point to point connection that does not require a pull-up resistor, the GPIO output structure must be a type PushPull

b. In gpio_open(), using two c-lines of code, set the LEUART-TX pin to the following:
    i. Strength – STRONG, Alternate – WEAK
        1. What Silicon Labs em library routine will you need to set drive strength? Is there an example in gpio_open()?
    ii. GPIO mode type = PushPull
        1. What Silicon Labs em library routine will you need to set the GPIO type mode? Is there an example in gpio_open()?
c. These c-lines of code should use the LEUART_TX port and pin definitions that you created in brd_config.h

ii. LEUART-RX:
1. RX refers to receive and thus refers to an input signal
    a. An input of a point to point connection that does not require a pull-up resistor has a GPIO structure of type Input
    b. Setting the strength of a signal refers only to output drive strength. With an input, there is no output of a signal so no requirement to set an output drive strength
    c. In gpio_open(), use a single c-line of code to configure LEUART-RX
        i. GPIO mode type = Input
    d. These c-lines of code should use the LEUART-RX port and pin definitions that you created in brd_config.h

9. The physical mapping of the Pearl Gecko has now been updated with establishing the LEUART0 clock tree and configuring the Pearl Gecko GPIO pins

10. Test Driven Development:
    a. *Before you begin developing the LEUART and BLE functions, you will first complete the development of a routine to test the UART peripheral. The goal in Test Driven Development is to develop a logical test of the desired function and not a test of the implementation. By writing a test to verify the end goal / requirements and not the actual implementation, it is analogous to having someone else proofread your report. The other person does not know what you have written or any previous conceptions, so it is likely that they will find mistakes that you would not. Also, since the test is designed to meet the end system requirements, it will verify that your code meets the basic program requirements of the function under test. This test should fail with the implementation not started or completed. With the test developed and verified that it checks and indicates a failure, then it is time to develop the program to pass the test.*

i. *When using EFM_ASSERT to verify that the clock tree of a peripheral has been configured correctly, it is an example of NOT being a logical test of the desired function, but a test of verification of its implementation.*

b. To begin, you will add a new event in app.h:
   i. #define BOOT_UP_CB xxxxxxxxxxx
      1. Remember that the basic scheduler implication on this project requires that each event is defined by a single unique bit
   ii. This will be an example of an event being set from a program and not an Interrupt Service Routine
      1. Add the BOOT_UP_CB event into the scheduler's static variable by calling add_scheduled_event(BOOT_UP_CB); routine from your void app_peripheral_setup(void) function
   iii. Now go back to your main.c while(1) loop. Add a check for the BOOT_UP_CB event, and if true, call the following function:
      1. void scheduled_boot_up_cb(void);
         a. Add this function as a function prototype in app.h and create the function in app.c
            i. As in similar routines, the first instruction in this event handler should remove the BOOT_UP_CB event from the scheduler's static variable.
               1. What scheduler function will you use to remove the BOOT_UP_CB from the scheduler's event variable?
            ii. Move the call to letimer_start(LETIMER0, true); from immediately before app_peripheral_setup() to the end of this scheduled_boot_up_cb() function
   iv. To ensure that the BOOT_UP_CB is set before entering the main.c while(1) loop, add an EFM_ASSERT verifying that the event_scheduled static variable has the BOOT_UP_CB bit set immediately before the while(1) loop statement in main.c. If it is zero / DEASSERTed, the Boot Up routine will not run which will result in not enabling the LETIMER0.
      1. EFM_ASSERT(get_scheduled_events() & BOOT_UP_CB);
      2. Is this EFM_ASSERT() test an example of Test Driven Development?

c. Put a breakpoint at the remove schedule event function call in the scheduled_boot_up_cb() function in app.c.

d. Run your program using the debugger:
   i. Does it stop at the breakpoint in scheduled_boot_up_cb() function just once, and does the program continue to work as in Lab 4?
      1. If yes, go to the next step
      2. If not, debug the code that you have added

11. void ble_test(char *mod_name){}

a. Download from the Lab 5 Canvas folder the HW_Delay.h/.c, leuart.h/.c and ble.h/.c files and copy them into their respective directories of your project. For this assignment, the prototype functions will be provided to you as well as a routine that can be used to program the name of you BLE module which will also be a logical test of your LEUART initialization routines, physical connection to the BLE module, and code development
    i. The LEUART0 code will not be application dependent, so your BLE module routines will be located in the ble.h and ble.c files.
    ii. In app.h, you must include the ble.h file. If you do not, you will obtain compile issues due to a .h or .c file in your source folder not "connected" to your project.
        1. #include "ble.h"
        2. ble_open() will use the defines in ble.h to initialize the STRUCT that will be sent to leuart_open(). Before you can successfully compile, you will need to complete all the definitions in ble.h and wherever possible, use a HAL enumeration. The information to determine these values will come from the HM-18 data sheet, online HAL documentation, micro-controller reference manual, or the micro-controller datasheet.

```
#define HM10_LEUART0            XXX
#define HM10_BAUDRATE           XXX
#defineHM10_DATABITS            XXX
#define HM10_ENABLE             XXX
#define HM10_PARITY             XXX
#define HM10_REFFREQ            XXX
#define HM10_STOPBITS           XXX

#define LEUART0_TX_ROUTE XXX
#define LEUART0_RX_ROUTE XXX
```

b. Go into the ble.c file that you imported into your project earlier in this lab and find the ble_test() function
c. The ble_test() routine will perform two functions:
    i. Verify that your LEUART has been configured properly for two-way communications, transmit and receive, and that there is proper connections from the LEUART0 peripheral to the BLE module
    ii. Program a name to the BLE module. All the BLE modules will have identical names out of the box, and it can be difficult to locate your module when all of modules are powered up during lab
    iii. For the name to be programmed into the DSD TECH HM-18, it must be reset. The test code will send the command to reset the DSD TECH HM-18 after the name has been sent to the BLE module. The BLE module reset time is undocumented. It appears that 2 seconds should be long enough for this reset to occur. The HW_Delay timer will be used for this 2 second timeout.

d. To successfully pass ble_test(), the following functions and integration of code into your project will be required:
    i. ble_open()
    ii. leuart_open()
e. Time to review the ble_test() function
    i. As you read the comments in the ble_test() function, you will notice that the comments will instruct you to complete lines of code to make this ble_test() function to operate correctly. Complete all the instructed lines of code to modify the default assignment of the data strings that will be sent to the BLE module to logically confirm communication from your ble_test() function through the LEUART peripheral to the DSD Tech HM-18 module and back.
        1. You will require the DSD TECH HM18 data sheet to complete this portion of the lab
    ii. For example, you must complete the following string default assignments indicated by the "" quotes. Since we are describing strings, the DSD TECH HM-18 command will be placed within the "" quotes.

```
// The break_str is used to tell the BLE module to end a Bluetooth connection
// such as with your phone.  The ok_str is the result sent from the BLE module
// to the micro-controller if there was not active BLE connection at the time
// the break command was sent to the BLE module.
// Replace the test_str "" with the "Test" command to disconnect/break a BLE connection
// Replace the ok_str "" with the result that will be returned from the BLE
//   module if there was no BLE connection

char          test_str[80] = "";
char          ok_str[80] = "";
```

    iii. There is a total of 3-commands that you will need to look up in the DSD Tech HM-18 data sheet and their expected responses
        1. Example: The testdisconnect string is "AT"
        2. Note: When the DSD TECH HM10 documentation looks for a ? such as AT+NAME?, it is looking for a user input string. If you look at the documentation, the ? is the value of the name that you are changing the DSD TECH module to be. If you scroll down the ble_test() function, you will find a strcat, string concatenation, function that will concatenate the test function input argument as the ? to the AT+NAMEstring.
    iv. There are also questions that you must answer in this function. These answers will be included in this assignment's rubric: Below is an example:

```
// This sequence of instructions is sending the break ble connection
// to the DSD HM10 module.
// Why is this command required if you want to change the name of the
// DSD HM10 module?
// ANSWER:
```

v. There is a total of 5-questions in this test function that you must answer

f. Add a function comment block for this test using the standard doxygen comment block that has been defined and used by this course.

g. Now, add the function call ble_test() in scheduled_boot_up_cb() in app.c.
   i. The argument that you pass to ble_test() will be the new name of your BLE module. Please select a name that is unique and be sensitive to others since your fellow students and the professor will be seeing your BLE module name on their phones as well.
   ii. ble_test() returns a value whether the test fails or passes. It could fail and be caught in an EFM_ASSERT during the ble_test() or return that it failed.
   iii. After your call to ble_test() in scheduled_boot_up_cb(), perform an EFM_ASSERT test on the returned value from ble_test(). This EFM_ASSERT test should jump to its while(1) loop if the ble_test() function return equals failed, false. Passes equates to true.

h. Now it is time to add the 2 second delay to allow the HM-18 to write its new module name into its non-volatile memory.
   i. After the call to ble_test() and the EFM_ASSERT test in the scheduled_boot_up_cb() routine, add a call to timer_delay() which is located in the HW_Delay.h/.c files. The input argument is the number of milli-second delay desired. Specify a 2 second delay.
   ii. You will need to add an include of the HW_Delay.h file into app.h

i. Try building your project.
   i. If it does not build, debug the reason that it is not building

j. If it builds, run the project using the debugger. Ensure that your phone is not connected to your BLE module.
   i. Does the ble_test() function work as you expect it would?
   ii. If yes, you are ready to go to the next step

12. NOTE: The instructing staff will not help with software coding questions of the remaining leuart.c and ble.c files until the ble_test() test compiles, indicates a failure, and appears completed including all answers and ble_test() function comment block.

13. Develop the remaining leuart.h/.c functions.

a. leuart.h
   i. In the leuart.h file provided, you will need to complete the definitions of the Energy Mode to block while receiving or transmitting data

```
#define LEUART_TX_EM          XXX
#define LEUART_RX_EM          XXX
```

b. leuart_open()
   i. First, you must determine what is required to complete the following structures defined in leuart.h

1. LEUART_OPEN_STRUCT

ii. The LEUART0 driver must be configured to be modular and encapsulated meaning that it is completely contained and transportable.

iii. Similar to Lab 4, you will need to complete the following structures in leuart.h:
    1. LEUART_OPEN_STRUCT: Used by an application module to open a LEUART peripheral

iv. LEUART_OPEN_STRUCT:
    1. The open STRUCT variables will come from two documents:
        a. Review the HAL documentation for LEUART_Init() function. All variables required for this function should be included in the LEUART_OPEN_STRUCT structure.
        b. Review the Pearl Gecko Reference Manual and Data Sheet for LEUART for all values required to route TX and RX from the LEUART peripheral to the Pearl Gecko external pin. These values / bits must be included in the LEUART_OPEN_STRUCT structure.
            i. What two things must be set for each TX and RX to properly route them from the LEUART peripheral to the external pins?
            ii. What registers and where are these bits located in each register?

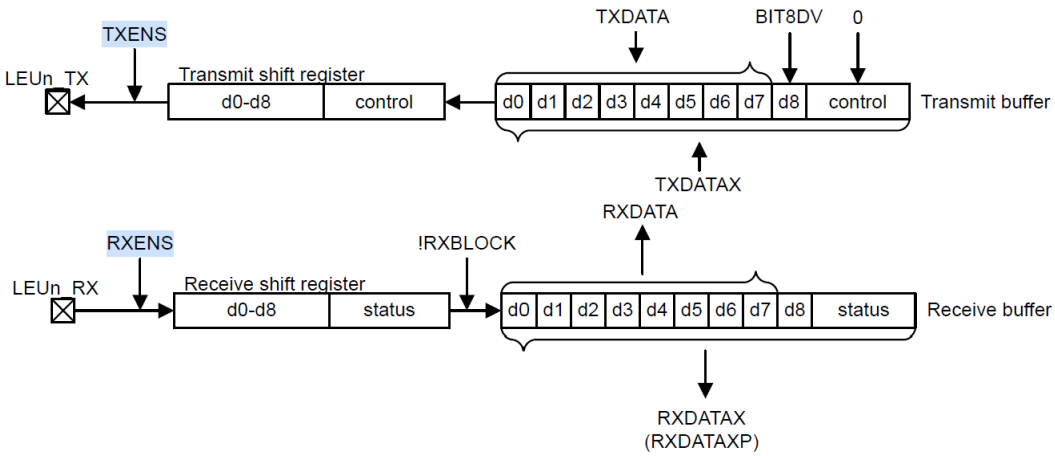
v. In the leuart_open() function, it must have one of its arguments as the LEUARTx peripheral and the other argument will be the configured LEUART_OPEN_STRUCT.

```
void leuart_open(LEUART_TypeDef *leuart, LEUART_OPEN_STRUCT *leuart_settings);
```

vi. The first code to be written in leuart_open() as in any open() driver function is to enable the peripheral clock
    1. You must also ensure that the clock tree to the LEUARTx is configured correctly in cmu.c
    2. Similar to the i2c_open() function, create a test right after your enabling of the clock to the LEUART0 peripheral to verify that the clock tree has been enabled and created correctly.
        a. You can check out Lab 4 assignment or your i2c_open() function on the implementation of this test
        b. The EFM_ASSERT() test must use a register that is controlled by the LEUART0 clock to be an effective test. For this assignment, perform a write and check with the LEUART0->STARTFRAME register.
        c. Per the LEUART0->SYNCBUSY register, you can verify that the STARTFRAME register is within the lower frequency, LEUART0 clock, domain

vii. *Note: The LEUART is being clocked off the LFXO 32.768 KHz oscillator which is at a different frequency than the CPU clock at 32 MHz. With the two devices at different frequencies, synchronization between the two devices is required when information is sent to registers located in the other clock domain.*

1. *For example, when you set the transmit and receive enable command via the LEUARTx->CMD register, you must check or stay in a "polling" while loop checking the LEUARTx->SYNCBUSY register for the synchronization to indicate the completion of the command transfer to the lower frequency LEUARTx peripheral*
    a. *As an example, let's look at an analogy. The SYNCBUSY indicating that the CMD has arrived into the Lower Frequency clock domain of the LEUART is similar to receiving a text from your favorite online retailer that your package has arrived in town and is out for delivery. You know the package has arrived in town, but you have not received it.*
2. *In taking a deeper look, the SYNCBUSY only checks that the signals have synchronized to the lower frequency domain, the LEUART peripheral*
3. *For key CMD signals that you require to actually have taken affect, you must verify that they have been executed by looking at the STATUS register*
    a. *For example, if you are turning on the receiver, RX, or transceiver, TX, the output of the CMD bits to enable these must propagate and enable the LEUART shift registers. Since the CMD register does not hold "state" of these enable "bits," these bits must be clocked into an internal LEUART register. Without having the schematics, it most likely requires one or two low frequency clock cycles*



b. *If the Pearl Gecko is running at 32 MHz and the LEUART is working at 32,768 KHz, the Pearl Gecko can execute*
    i. *Time period of one LEUART clock cycle*
        1. $t = \frac{1}{32{,}768} = 30.5uS$
    ii. *Number of Pearl Gecko clock cycles in one LEUART clock period*
        1. $cycles = 32{,}000{,}000 * 0.0000305 = 976$
c. *If the ARM Cortex-M4 processor can operate at 1 instruction per cycle, the Pearl Gecko could operate from 976 (1 LEUART cycle delay) to 1,952 (2 LEUART cycle delay) instructions before the LEUART could perform a transmit or receive after synchronization of the CMD register*
d. *So, the SYNCBUSY register informs you that the configuration information or CMD has been synchronized to the lower frequency peripheral, we need a register to inform us that the CMD has been performed.*
    i. *Going back to our analogy of SYNCBUSY informing us that the package is near, we need a method to confirm to the online retailer that the package has been received such as a signature confirming receipt of the package. We need to find a way to confirm that the CMD has been received and executed upon.*
e. *For the TXEN and RXEN signal, you then can use the LEUART STATUS register to provide the feedback that the TXEN and RXEN commands to the CMD register have been executed, taken affect. You use while (leuart->SYNCBUSY) poll to validate that the signals have arrived in the lower frequency domain, you must then poll the LEUARTx->STATUS register for these signals to be ASSERTed to ensure that the LEUART is ready for operation.*

viii. *You can determine what LEUART registers exist in the lower frequency domain by looking at the LEUART SYNCBUSY register definition found in the Pearl Gecko Reference Manual:*

17.5.18 LEUARTn_SYNCBUSY - Synchronization Busy Register

Offset	Bit Position																													
0x044	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset																									0	0	0	0	0	0	0	0
Access																									R	R	R	R	R	R	R	R
Name																									PULSECTRL	TXDATA	TXDATAX	SIGFRAME	STARTFRAME	CLKDIV	CMD	CTRL

ix. *In writing your functions, what register writes to the LEUART peripheral should you poll to wait for synchronization to complete?*

x. *A simple synchronization polling routine is the following:*
   1. *while(LEUARTx->SYNCBUSY);*
   2. *If any register that the SYNCBUSY monitors is in the process of synchronization, the value will be non-zero and result staying in the while loop to read LEUARTx->SYNCBUSY until it returns all zeros.*

xi. *You must be careful to use the SYNCBUSY check for your direct writes to the asynchronous registers as well as the Silicon Lab routines that do not perform a SYNCBUSY check at the end of their function.*
   1. *Example: From the LEUART_Enable() function, you can see below that the library routine does a SYNCBUSY check/stall before writing to the CMD register, but not one afterwards. If in your leuart_open() function you are to write to an asynchronous register or you may use the LEUART0 peripheral before a previous write to a CMD, CTRL, or library routine is synchronized, you must perform a while(leuart->SYNCBUSY) stall to ensure the desired function is programmed into the LEUART peripheral before you access the register or use the peripheral. It is good practice to perform a SYNCBUSY stall after each of these asynchronous register writes or library calls.*
      a. *By allowing the programmer to determine when to perform the SYNCBUSY check/stall after a library function call, Silicon Labs leaves it to the programmer to develop code that is more energy efficient. For example, if after you write to an asynchronous register you perform non-asynchronous c-code before you perform the SYNCBUSY check/stall, you are enabling parallelism by performing code operations while simultaneously synchronizing the registers.*

```
void LEUART_Enable(LEUART_TypeDef *leuart, LEUART_Enable_TypeDef enable)
{
  uint32_t tmp;
```

```
    /* Make sure that the module exists on the selected chip. */
    EFM_ASSERT(LEUART_REF_VALID(leuart));

    /* Disable as specified. */
    tmp   = ~((uint32_t)(enable));
    tmp  &= (_LEUART_CMD_RXEN_MASK | _LEUART_CMD_TXEN_MASK);
    tmp <<= 1;
    /* Enable as specified. */
    tmp |= (uint32_t)(enable);

    /* LF register about to be modified requires sync; busy check. */
    LEUART_Sync(leuart, LEUART_SYNCBUSY_CMD);

    leuart->CMD = tmp;
  }
```

2. *Example: Let's take a look at the Silicon LEUART_Init() function below. It performs a SYNCBUSY check/stall before the write to the CMD register, and it performs a second SYNCBUSY check/stall in the LEAURT_FreezeEnable(). The LEUART_FreezeEnable() function is used to allow writes to multiple asynchronous registers and then "unfreeze" them to be synchronized simultaneously to the low frequency clock domain. After the "unfreeze," there is no SYNCBUSY check/stall. If you write to one of these registers before synchronization is complete, you may over write the desired LEUART_Init() initialization or if you try to access the peripheral before the synchronization is complete, the peripheral may not operate as you expect.*

```
void LEUART_Init(LEUART_TypeDef *leuart, LEUART_Init_TypeDef const *init)
{
  /* Make sure the module exists on the selected chip. */
  EFM_ASSERT(LEUART_REF_VALID(leuart));

  /* LF register about to be modified requires sync; busy check. */
  LEUART_Sync(leuart, LEUART_SYNCBUSY_CMD);

  /* Ensure disabled while configuring. */
  leuart->CMD = LEUART_CMD_RXDIS | LEUART_CMD_TXDIS;

  /* Freeze registers to avoid stalling for the LF synchronization. */
  LEUART_FreezeEnable(leuart, true);

  /* Configure databits and stopbits. */
  leuart->CTRL = (leuart->CTRL & ~(_LEUART_CTRL_PARITY_MASK
                                   | _LEUART_CTRL_STOPBITS_MASK))
                 | (uint32_t)(init->databits)
                 | (uint32_t)(init->parity)
                 | (uint32_t)(init->stopbits);

  /* Configure the baudrate. */
  LEUART_BaudrateSet(leuart, init->refFreq, init->baudrate);
```

```
  /* Finally enable (as specified). */
  leuart->CMD = (uint32_t)init->enable;

  /* Unfreeze registers and pass new settings on to LEUART. */
  LEUART_FreezeEnable(leuart, false);
}

void LEUART_FreezeEnable(LEUART_TypeDef *leuart, bool enable)
{
  if (enable) {
    /*
     * Wait for any ongoing LF synchronization to complete to
     * protect against the rare case when a user
     * - modifies a register requiring LF sync
     * - then enables freeze before LF sync completed
     * - then modifies the same register again
     * since modifying a register while it is in sync progress should be
     * avoided.
     */
    while (leuart->SYNCBUSY != 0U) {
    }

    leuart->FREEZE = LEUART_FREEZE_REGFREEZE;
  } else {
    leuart->FREEZE = 0;
  }
}
```

xii. After the LEUART0 clock has been enabled and verified, the next step is to complete the LEUART_Init_TypeDef local STRUCT and call the LEUART_Init() function to initialize the peripheral

xiii. DO NOT ENABLE THE LEUART AT THIS TIME EVEN IF YOU ARE PASSING THE VALUE TO ENABLE THE LEUART TO leuart_open(). YOU HAVE NOT FINISHED INITIALIZING THE PERIPHERAL AND THE PERIPHERAL CANNOT BE INITIALIZED WHILE THE PERIPHERAL IS ENABLED.

1. You can use the analogy of shifting gears in a car. You can change gears while the car is stopped (not enabled), but it is recommended that you do NOT change from drive forward to reverse while the car is moving (enabled) at 20 miles per hour.
2. You will enable the LEUART per the requirements at the end of leuart_open() after you have initialized the peripheral completely.

xiv. Since the LEUART will be using external Pearl Gecko pins to communicate with the BLE module, after initializing the peripheral, you will need to route the LEUART TX and RX signals to the proper GPIO pins

xv. It is good practice in configuring a peripheral to ensure that it is in a known or starting state before you enable the peripheral. After you have completed setting up the LEUART0 and before you enable the LEUART0, you should perform a command to clear the RX and TX buffers

1. What register will you write to perform this clearing function?
2. What enumerations / definitions will you use to write these bits into the register

xvi. We will look at adding the initialization of the LEUART0 interrupts later in this lab.

xvii. With the LEUART now completely initialized, it is time to enable the LEUART using the em_library routine LEUART_Enable();

xviii. At the end of the leuart_open(), perform an EFM_ASSERT() to verify whether TX and RX have been enabled per LEUART_OPEN_STRUCT argument.

1. The EFM_ASSERT should be reading the value of TXENS and RXENS from the LEUART0->STATUS register
2. For the synchronization of TXENS and RXENS signal, you must also poll the LEUARTx->STATUS register for these signals to be ASSERTed if you are enabling them.

c. Before running the ble_test(), place a breakpoint at the end of your leuart_open() function and verify that the peripheral has be configured as expected

i. Is it at the correct baud rate?
ii. Are the pins routed correctly?
iii. Are the TX and RX enabled as program
iv. ed?
v. Etc.

d. The ble_test() function is a Test Driven Development unit test that validates that the LEUART0 has been initialized correctly to enable LEUART transmits and receive from an external UART device. It is not a test to validate your interrupt driven state machine. At this point, one more function is required before you can run ble_test() to validate proper configuration of the LEUART peripheral.

i. The ble routine, ble_open(), will need to be developed to initialize the LEUART_OPEN_STRUCT which will then be used to call the leuart_open() function.

1. This routine is very similar to what routine in the I2C lab?

ii. As with other routines that initialize the system, a call to ble_open() should be made from your set up function in app.c

```
void app_peripheral_setup(void)
```

iii. Due to the frequency of transmitting to the LEUART, 2.7 seconds, we do not want to go into Energy Mode 3 due to the time to turn on / stablize the LFXO oscillator is 1.0 seconds. Add a new energy mode block define in app.h

1. #define SYSTEM_BLOCK_EM EM3

iv. Add a call to block_sleep_mode(SYSTEM_BLOCK_EM); to your app_peripheral_setup(void) function

v. With ble_open() now developed and called to initialize its interface with the LEUART module, place a breakpoint after the ble_test() function call in the app.c function scheduled_boot_up_cb().

1. Compile your program and run your program in the debugger with your phone NOT connected or paired with your BLE module

2. If you have reached the breakpoint and ble_test() returned true or "1," your LEUART peripheral has been successfully configured up to this point of the assignment. If returned false or "0," you will need to debug your ble_open() and leuart_open() functions.

vi. *The following leuart functions are polling functions that are used by ble_test(). These functions access the peripheral information instead of ble_test() directly to maintain modular and encapsulation principles. These routines are provided to you in the leuart.h and leuart.c files that you copied into your project.*

```
uint32_t leuart_status(LEUART_TypeDef *leuart);
   • Returns the value of the LEUART status register
void leuart_cmd_write(LEUART_TypeDef *leuart, uint32_t cmd_update);
   • Writes the value of cmd_update to the LEUART CMD register
uint32_t leuart_ien_test_get(LEUART_TypeDef *leuart);
   • Returns the value in the LEUART IEN register
   • Disables all interrupts to the LEUART (IEN = 0)
void leuart_ien_reset(LEUART_TypeDef *leuart, uint32_t ien_reg);
   • Clears all pending interrupts in the LEUART IF register
   • Write the ien_reg into the LEUART IEN register.  It is not an OR
     operation with the IEN register.
void leuart_app_transmit_byte(LEUART_TypeDef *leuart, uint8_t data_out);
   • Sends a single byte to the LEUART TXDATA buffer
uint8_t leuart_app_receive_byte(LEUART_TypeDef *leuart);
   • Polls until a byte is available in the LEUART RXDATA buffer
```

vii. *The naming convention I chose mentions which register the function accesses is referred in the routine's name such as leuart_status() function returns the status of the LEUART->STATUS register*

viii. *What leuart register would leuart_app_transmit_byte() function access?*

ix. *Some functions require a check on whether a LEUART peripheral's register is available or whether data is ready to be returned. For example, if the transmit buffer is full, if you try to write data into the transmit buffer, you will overwrite the existing data and thus not transmit the desired sequence of characters*

1. *What function(s) must you verify that the register is empty before you write to it?*
    a. *What c-line of code would you use to verify whether a register is available/empty to accept data?*
2. *What function(s) must you verify that there is data ready before you read the register?*
    a. *What c-line of code would you use to verify whether data is available to read?*

x. *Each of these function that write data to a LEUART register that exists in the LEUART clock domain will require a SYNCBUSY check before the routine is returned*

1. *Which of these functions require a SYNCBUSY check?*

Checkpoint 1: You should target to reach this point before Lab on Tuesday, October 27th

e. The assignment will send a string to the LEAURT to be transmitted to the BLE module using interrupts to maintain the following goals:
    i. Maintain low energy operation
    ii. Enable other tasks access to the CPU if needed

f. LEUART0_IRQHandler():
    i. Similar to the I2C0_IRQHandler(), the interrupt handler will be used to determine when the LEUART0 is available to transmit and when the LEUART0 has completed transmission. Refer back to your Software Flow Chart to develop the LEUART0_IRQHandler() routine. For this lab, you will need to implement LEUART0 transmissions via the interrupt handler. The test routine uses polling to obtain read bytes from the LEUART peripheral and did not use the LEUART0 Interrupt Service Routine.
        1. What interrupt would you use to determine that you can send a byte to the LEUART0 for transmission?
        2. What interrupt would you use to determine that the LEUART0 has completed all transmissions that have been sent to the peripheral?
        3. Now go back and take a look at your leuart_open() function. What must you do to enable the interrupts to the CPU? Add these instructions at the end of leuart_open().
        4. What instruction should you perform immediately before enabling the interrupts in the LEUARTx->IEN register?
    ii. You will need to define a static State Machine STRUCT and what information is required by this static variable / structure to maintain your state machine. It must be defined in leuart.c to make it private.
    iii. As it is waiting to perform the next step in the state machine in the SW flow chart, it should exit the ISR and go back to sleep if no other event is required servicing
    iv. Since the LEUART driver is designed to be modular and encapsulated, any functions that the LEUART0_IRQHandler would need to implement its state machine must be located in leuart.c.
        1. Since these functions are meant solely to be accessed by other routines in leuart.c, they should be defined as static function protypes
    v. Upon completion of the string transmission, what interrupt should you use to unblock the sleep mode placed by the leuart_start() function since the LEUART0 can only work down into EM2 with the LFXO oscillator
        1. The interrupt handler should also set an event indicating that the LEUART0 transmission has been completed

g. leuart_start():
    i. As in the i2c driver, the leuart_start() function is used to initialize the state machine private variable that the driver will use to transmit a string of characters
    ii. With the LEUART0 configured with the LFXO oscillator, it can only operate down into EM2
        1. You must set a block sleep mode in this function of not going below EM2 sleep mode since you are about to start using the LEUART peripheral.
    iii. After all static variables required for your software flow chart and operation of transmitting a string are configured, it is time to initiate the LEUART transmission?
        1. How would you initiate the transmission of the string at this point?

h. It is time to make sure the code will compile and does not break any existing code. It should operate as Lab 4 and pass in ble_test() since all the code has for the test has been completed.
    i. If it does not, debug until it compiles and passes ble_test()

14. Develop the ble_write() function:

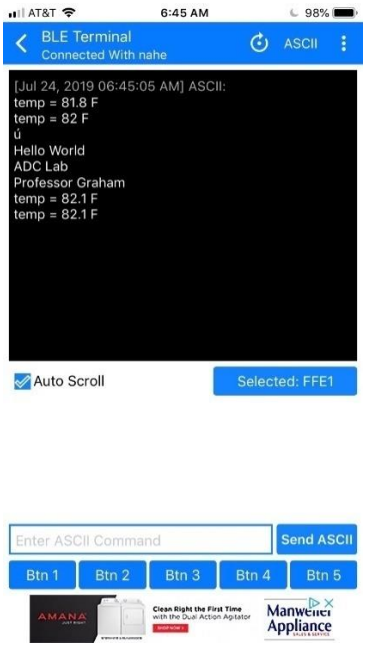
a. ble_write() is a routine to package a write request and send it to the leuart_start() function
    i. This routine is required since an application does not know what LEUARTx the BLE module is connected
    ii. The leuart_start() function is a general driver and must be told which LEUARTx peripheral to initialize and start the state machine for write transmission
    iii. This ble_write() function will need an input string whose last character of the string is a null,"", character
        1. Why must the last character of the string be a null,"", character?
        2. HINT: What library function returns the length of a string?
            a. How does this function know when a string ends?
    iv. In the ble_write() function, it must provide at a minimum the following information to leuart_start()
        1. LEUARTx peripheral
        2. Pointer to the start of the string
            a. With the requirement that this string exists between functions, it must be defined as a private / static string variable
        3. (optional) string length
            a. Why can this argument be optional?
        4. Event to call back when the BLE transmission is completed

15. Add an event handler for BLE_TX_DONE_CB

a. Go to app.h and add an appropriate complete #define for BLE_TX_DONE_CB which will be used to request an event to be serviced after the LEUART driver has completed the transmission of a string
b. Add an event handler for BLE_TX_DONE_CB in app.h/.c as well as an IF statement in the main.c while(1) loop to call this event handler upon the LEUART0 Interrupt Routine setting the event upon completion of a string transmission
c. At this time, in the callback handler in app.c, just add the instruction to clear or remove the event from the event_scheduled private variable

16. Time to test your BLE application:

a. Download your Bluetooth Module network co-processor, or similar, application to your phone and install it. You will be using this application to interface to your Pearl Gecko + Bluetooth module kits.
    i. For the iphone, I recommend the BLE Terminal – HM10. I found that the DSD Tech app had a bug that impacts Lab 6. Below is a screenshot from the BLE Terminal app.



    ii. Once you open your app, you will need to find your particular BLE module. Initially, all the modules will have the same name. After you have successfully passed your ble_test() test function, the name of your BLE module should be changed to the name that you passed as an argument to the ble_test() function.
    iii. Once you find your BLE module, click on our BLE module and it should open to a screen similar to the app on my iPhone.

b. ADD ble_write() to the scheduled_boot_up_cb() function

i. It is time to verify whether the BLE write and LEUART function correctly before adding code to the application to transmit the temperature to your phone
ii. After the ble_test() function call, its EFM_ASSERT test, and timer delay, send a string to your phone such as "\nHello World\n" using your ble_write() function
iii. The application and driver at this time is not designed to handle a second write attempt to access the LEUART0 before a previous transmission has been completed. We will add the ability to handle the situation of back to back writes in the next lab, Lab 6.
iv. Now, compile and run your project, make sure that your phone is not connected to your BLE module.
    a. If your program is passing the ble_test() function, are you seeing the new name as you scan for it using your phone app?
        i. If not, you may need to restart your phone to fully forget the previous name of your module as well as unplug, wait for 10-seconds, and plug back in your Pearl Gecko starter kit
    b. Now, you will explore using c-compiler directives to specify your code run compile options based on a #definition. The optionality based on a #define statement enables a single code base work across multiple circumstances. In case of the Silicon Labs' em_libraries, c-compiler directives are used to enable different sequence of code operation based on the micro-controller chosen for the project. Before your call to ble_test(), add the following c-compiler directive:
        i. #ifdef BLE_TEST_ENABLED
    c. After the timer_delay() and before the ble_write() of "\nHello World\n", add the end if c-compiler directive
        i. #endif
    d. Since you have not defined BLE_TEST_ENABLED, your code should be "greyed out" which indicates that it will not be compiled during the next build

```
void scheduled_boot_up_cb(void){
    remove_scheduled_event(BOOT_UP_CB);

#ifdef BLE_TEST_ENABLED
    EFM_ASSERT(ble_test("Prof, 4"));
    timer_delay(2000);  // delay is in milliseconds
#endif

    ble_write("\nHello World\n");

    letimer_start(LETIMER0, true);
}
```

e. Now, go to app.c and add the following definition
    i. #define BLE_TEST_ENABLED
f. Next, save all of your files and go back to your app.c file. Since BLE_TEST_ENABLED is now defined, your call to ble_test() should not be "greyed out" and thus will be compiled into your project

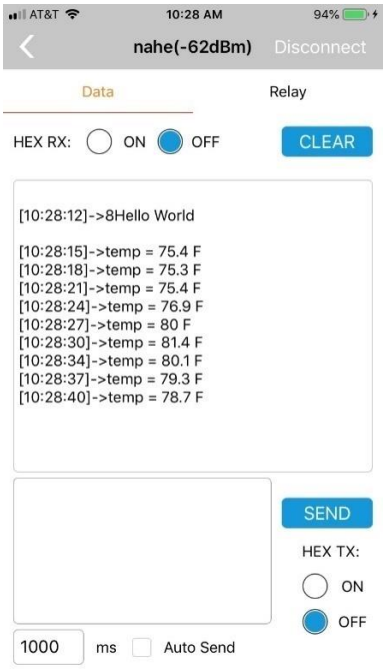
```
117 void scheduled_boot_up_evt(void){
118     remove_scheduled_event(BOOT_UP_EVT);
119
120 #ifdef BLE_TEST_ENABLED
121     EFM_ASSERT(ble_test("Prof"));
122     for(int i = 0; i <20000000; i++);
123 #endif
124
125     char test_str[80]="\nHello World\n";
```

g. These compiler directives will be included in the grading rubric
h. Go back to app.c and comment out the BLE_TEST_ENABLED definition. Once the ble_test() passes, you are not required to run it again
    i. Please note, the ble_test() will be ran by the graders
i. Compile and run your project. Once you are in the Energy Profiler, connect your phone app to your BLE module and reset your Pearl Gecko via the reset switch next to Pearl Gecko indicated by the yellow circle below.

j. After reset, you should see “Hello World” print to your phone app

17. NOTE: The instructing staff will not help with application coding questions until the “Hello World” test appears on the phone app.

18. Time to add your application code
    a. Once the Si7021 read data has been converted into degrees F and compared to 80F, your application code in scheduled_temp_done_cb() should write the temperature to the BLE module. The format of the data to your phone must be:
        i. Temp = 72.5 F
        ii. You will only print to your phone one decimal point of precision, and if the value is to an integer value, you do not need to write out the decimal point or the tenth of a degree digit. In this cases, your output could be:
            1. Temp = 74 F
        iii. The string that you send to the phone should look like the following:
            1. “Temp = 78.5 F\n”
            2. The \n indicates to the BLE Terminal app a new line character
    b. The application code can be as little as three or four lines of code.
        i. In my implementation, I create a temp_str array, 1 line of code, two lines of code to format my data, and the fourth line of code to send it to the ble module using my ble_write() function
        ii. You are allowed to use string library functions to format your data
    c. Data on you phone should look similar to my screen shot below:



d. Using the Energy Profiler, zoom in on an LETIMER interrupt to read the Si7021 temperature value. This interrupt should look similar to the below screenshot:

19. Doxygen comments
    a. The module and all functions must be included in your Doxygen comment report for this project, both global and private functions. This includes the following modules:
        i. app.c
        ii. ble.c
        iii. leuart.c
        iv. letimer.c
        v. si7021.c
        vi. i2c.c

Deliverables:

1. Each person must submit their Lab 5 software flow chart into the Lab 5 Program canvas assignment
2. Each person must submit a separate Doxygen zip file of their Lab 5 exported project into the Lab 5 Program canvas assignment
3. Each person exports their project as an archived filed. Upload the .zip file into the Lab 5 Program canvas assignment
4. Each person to provide their answers via Canvas/quiz/Lab5 Worksheet

Questions:

Complete the Lab 5 worksheet to complete this assignment.

Point breakdown:

- Lab 5 has a total of 40 points
- Exported project will be graded on:
    - Functionality
        - Questions answered in the ble_test() function
        - Correctly transmission of temperature through the BLE module to its phone app
        - LED 1 turns on when the temperature is read equal to or above 80F
        - LED 1 turns off when the temperature is read below 80F
        - Code correctly works while built using optimization -02
    - Meeting Energy / Current expectations
        - Energy Profiler results
    - LEUART driver is developed to be a generic and non-application specific
    - Application specific code located in app.h/.c
    - LEUART driver is interrupt driven
        - Verified via the Energy Profiler and code review
    - All functions include proper use of Header Guards

    - Proper commenting of functions as previously defined in earlier assignments
    - Silicon Labs IP statement in sleep_routines.c
    - 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 35 pts
    - 1 day late to 3 days late — max score is 30 pts
    - 3 days late to 5 days late — max score is 25 pts
    - After 5 days late — max score is 0 pts