

# READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

## ECEN 2370 Embedded Software Engineering I2C driver – Si7021 Temp Sensor Fall 2020 Revision 1.0 – Highlighted in GREEN Revision 2.0 – Highlighted in BLUE

**Objective:** To learn how to develop an I2C driver to read the temperature from the Si7021 temperature sensor. This driver will follow the following structure:

- Modular
- Encapsulated
- Interrupt Driven State Machine

Before beginning implementing the Interrupt Driven State Machine for this assignment, a Software Ladder (or sequential) Flow Chart must be developed. Proper planning of the Software will minimize development, testing, and debug.

For this assignment, upon the LETIMER UF interrupt, every 2.7s, the Pearl Gecko will request a temperature read from the Silicon Labs Si7021 Temperature/Humidity sensor which is integrated into the Pearl Gecko STK3402 Starter Kit. To interface to the Si7021, you will develop an I2C driver which will be interrupt driven. To achieve this I2C read, a software state machine will need to be developed which will transition from one state to the next state upon interrupts. Between these interrupts, the Pearl Gecko should go into the lowest energy state possible to maintain proper operation.

When the temperature rises above 80F, LED1 will turn-on indicating a high temperature and will turn-off when the temperature returns below 80F.

Key Learning Objectives for this Assignment:

- **Developing Software Flow Charts:** Planning and architecting either a hardware or software project is an important concept to learn. Successful preparation before implementing code will result in better code that is easier to debug resulting in overall less time on a project. Like the use of non-Magic numbers in coding, the flow chart should use clear wording to describe the states, the events that define transition of states, and any important information that will be required to translate the flow chart to program code.
- **Converting the Software Flow Chart to an Interrupt Driven State Machine:** After planning a software state machine through its software flow chart, it must be broken into the required functions to be implemented in software. A flow chart that includes

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

defined states and the events that define state transitions will greatly reduce the effort to convert a flow chart to its equivalent software code.

- **I2C (Inter-Integrated Circuit):** I2C is one of the most prevalent digital communication buses for embedded system that sensors and actuators use to communicate with the microcontroller. The bus protocol supports multiple devices on the same bus through a device addressing scheme as well as supporting multi-bus masters. The I2C bus is a serial bus requiring only 2-signals/GPIO pins by the microcontroller enabling its use with small and inexpensive microcontrollers. The versatility of the I2C protocol supporting multiple devices and use by a wide range of microcontrollers from low to medium to high-end systems makes it an important bus protocol to learn and understand.
- **Interrupt Driven State Machines:** Interrupts enable a microcontroller to support multi-tasking such as reading a temperature from an I2C sensor and writing data to a Bluetooth Network Processor concurrently. It appears to be concurrently, but actually interleaved. If a communication bus is waiting for a response, the microcontroller can move to another task and then return to the first task when the appropriate interrupt is received. Developing an Interrupt Driven State Machine is critical to enable multi-tasking of embedded system.

**Note:** You will be using the completed Lab #3 project as a starting point for Lab #4.

### Pacing:

- By Monday, October 5<sup>th</sup>, at 11:59pm
  - Just inside si7021.c, approximately page 15
- By Monday, October 12<sup>th</sup>, at 11:59pm
  - Up to si7021.c – Part 2, approximately page 29

### Due Dates:

- Interrupt Driven State Machine: Saturday, October 10<sup>th</sup>, 2020 at 11:59pm
- Entire project: Sunday, October 18<sup>th</sup>, 2020 at 11:59pm

### Note:

- Code will be tested and graded while built with -O2 optimizer
- Recommend that you get the project fully functional in -O0 before testing in -O2

### Making change to clarify documentation:

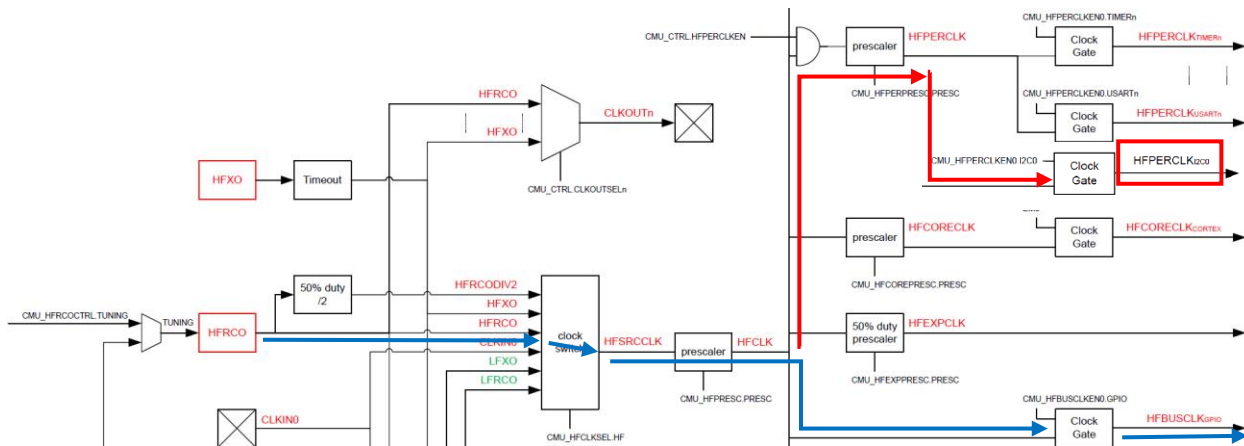
- *Italicized text will signify informational details of the assignment*
- Standard text will signify activities required to complete the assignment

### Instructions:

1. Work with the instructing team to get your Lab #3 project fully functional.

READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

2. Change the name of the project by adding your two initials in front and the Lab name, I2C\_Lab
  - a. Ex. For Keith Graham, the name would change to KG\_I2C\_Lab
3. Before we begin to develop any code for the I2C peripheral or Si7021 device, we first must configure the Pearl Gecko hardware for the I2C peripheral clock tree and the GPIO pins that will be used to connect the Pearl Gecko to the Si7021.
  - a. **cmu\_open() function:**
    - i. Let's take a look at the I2C clock tree first.



- ii. *The Pearl Gecko EFM32PB12 Reference Manual is missing the I2C peripheral in the high frequency clock tree. I have added the I2C block per the Pearl Gecko EFM32PB12 Reference Manual.*
- iii. *In a previous lab, the high frequency oscillator was routed to the GPIO peripheral, indicated by the blue arrows*
  - 1. *The enabling of the HFRCO oscillator and setting the select lines to place the HFRCO onto the HFCLK bus was done in main.c*

```
/* Switch HFCLK to HFRCO and disable HFXO */
CMU_HFRCOBandSet(MCU_HFXO_FREQ); // Defined in brd_config.h
CMU_OscillatorEnable(cmuOsc_HFRCO, true, true);
CMU_ClockSelectSet(cmuClock_HF, cmuSelect_HFRCO);
CMU_OscillatorEnable(cmuOsc_HFXO, false, false);
```

- iv. *Is the clock branch already established and enabled in `cmu_open()`?*
  - 1. *If yes, no clock tree changes are required*
    - a. *Remember, you will be enabling the clock, `CMU_HFPERCLKEN0.I2C0`, signal described in the clock tree diagram in your I2C open function*
  - 2. *If no, perform what is required to enable the proper oscillator and switch it onto the correct clock branch that provides the clock signal to the I2C peripheral*

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

### b. `gpio_open()` function:

- i. The Si7021 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 Si7021 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. Please add the below `#define` statements in `brd_config.h`.

```
#define SI7021_SCL_PORT      XX
#define SI7021_SCL_PIN      XX
#define SI7021_SDA_PORT      XX
#define SI7021_SDA_PIN      XX
#define SI7021_SENSOR_EN_PORT XX
#define SI7021_SENSOR_EN_PIN XX
```

- ii. *These definitions are similar to the definitions used to define the LED0 and LED1 pins in Lab 1.*
- iii. The next step of setting the hardware dependencies / configuration of the Pearl Gecko is to define the GPIO ports, pins, and properly set the GPIO pin modes for the Pearl Gecko interface to the Si7021.
  1. To locate what pins are connected / routed from the Pearl Gecko to the Si7021, open up the Pear Gecko STK3402 Start Kit document and go to figure 6.4. I have blackened out several key pieces of information so that you will need to find this documentation in the Pearl Gecko STK3402 Start Kit document.

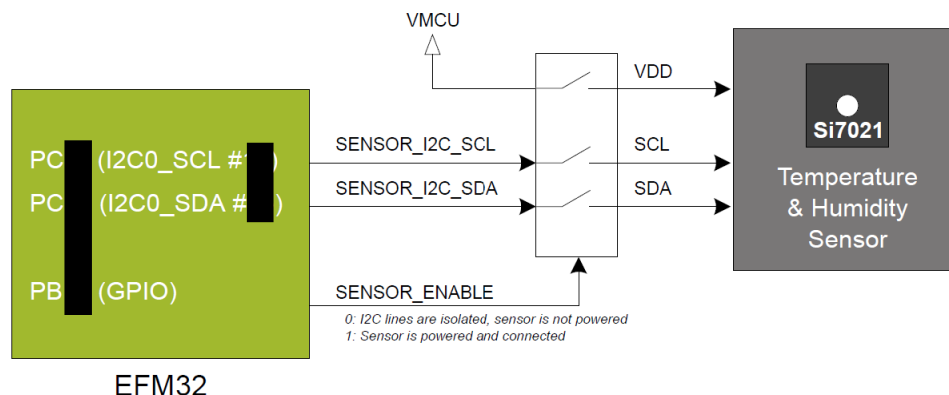


Figure 6.4. Si7021 Relative Humidity and Temperature Sensor

2. In the Pearl Gecko STK3402 Starter Kit document, go to figure 6.4. Complete the below definitions based on the information to the

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

left of each signal in figure 6.4 for the below #define statements in `brd_config.h`.

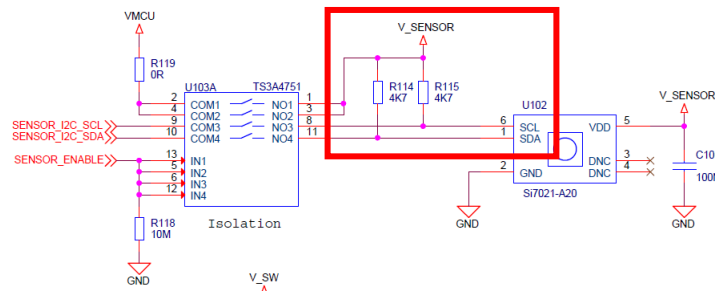
```
#define SI7021_SCL_PORT      XX
#define SI7021_SCL_PIN      XX
#define SI7021_SDA_PORT     XX
#define SI7021_SDA_PIN      XX
#define SI7021_SENSOR_EN_PORT XX
#define SI7021_SENSOR_EN_PIN XX
```

3. *As a reminder, port in the Pearl Gecko are described as `gpioPortX`.  $X$  = the letter value of the port. You will need to go to the Pearl Gecko STK3402 Start Kit document to determine the I/O pin numbers and ports that have been blackened out.*
- iv. From Figure 6.4, you will notice that there are three control signals that go from the Pearl Gecko to the Si7021. The `SENSOR_ENABLE` bit is used to select via an external device to the Pearl Gecko whether the Pearl Gecko pins are connected to the Si7021.
  1. In Figure 6.4, under `SENSOR_ENABLE`, it provides the information on whether `SENSOR_ENABLE` must be ASSERTED, 1, or DEASSERTED, 0, to connect the two devices
  2. What should `SENSOR_ENABLE` be set to connect these two devices, ASSERTED or DEASSERTED?
  3. The output pin from the Pearl Gecko must drive `SENSOR_ENABLE` either to a 0 or a 1 to provide a proper signal to the external switch that is isolating the Pearl Gecko from the Si7021. A signal that must be a 0 or a 1 comes from a PushPull output configuration.
    - a. In `gpio_open()`, set `GPIO_DriveStrengthSet` of the `SENSOR_ENABLE` to `StrengthWeakAlternateWeak`
    - b. Also, configure the `GPIO_PinModeSet` to a PushPull configuration where the default configuration is connecting the Pearl Gecko to the Si7021 through the isolation buffer.
    - c. You can use the Pearl Gecko HAL documentation to complete the set up of these two emlib functions using the proper enumerations
- v. *The next two pins, `SENSOR_I2C_SCL` and `SENSOR_I2C_SDA` are the I2C pins. In I2C, both pins can be used for an input or an output signal. To prevent an issue of over driving a device's Electro-Static Discharge (ESD) diode or input circuit, discussed or will be discussed in class, no device is allowed to drive the signal HIGH. Since no device can drive the signal high, to make the signal high, the GPIO pin will tri-state when the desired signal is to be high and allow an external pull-up resistor to take the*

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

signal to a HIGH or 1 value if no other device is pulling the signal to LOW, 0. Below shows the pull-up resistors from the Pearl Gecko STK3402 schematic.

Relative Humidity & Temperature Sensor



- vi. In this configuration, the bus acts as an AND logic gate. If all the connected signals to the output are HIGH, 1, then the signal is HIGH, 1. For an open drain output, a HIGH places the output into a tri-state mode which enables the pull-up to pull the bus HIGH.. If any of the connected outputs are LOW, 0, that output will drive the signal to LOW, 0, for the entire bus.

### AND Gate Truth Table

TABLE 3-2  
Truth table for a 2-input AND gate.

INPUTS		OUTPUT
A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

1 = HIGH, 0 = LOW



Figure 3-14 Boolean expressions for AND gates with two, three, and four inputs.

8

- vii. In `gpio_open()`, there is no need to set the output strength for the SCL and SDA lines since the Pearl Gecko will never be driving these signals HIGH, it will always allow the external pullup resistors to pull the signal HIGH, 1. To mimic this AND functionality, the Pearl Gecko GPIO output structure must be set to WiredAND.

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

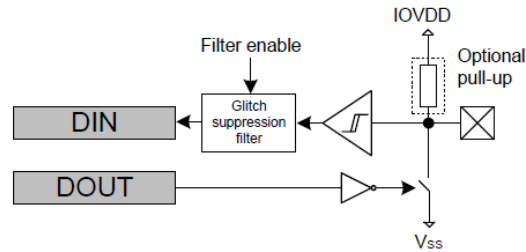


Figure 31.4. Open-drain

viii. The WiredAND output structure is performed by using an Open-drain GPIO structure as in the above figure from the Pearl Gecko Reference manual. From reviewing the block diagram, there is no output structure to push the output high, only an output, DOUT, that can enable a switch to pull the signal LOW, 0. With the use of external pullup resistors, you do not want to use the configuration that includes internal pullups, WiredANDPullUp.

- a. Configure the [GPIO\\_PinModeSet](#) to a WiredAND configuration where the default value is true, 1
  - i. You will use default of 1 or High to enable any other device if required to drive the signal LOW to initiate conversation
- b. You can use the Pearl Gecko HAL documentation to complete the setup of these two emlib functions using the proper enumerations
- c. The CMU and GPIO configuration of the Pearl Gecko has now been established for this LAB

#### 4. [I2c\\_open\(\)](#) function:

- a. The I2C driver must be configured to be modular and encapsulated meaning that it is completely contained and transportable.
- b. Create `i2c.h` and `i2c.c` files in the appropriate directories. You may want to copy the framework from another `.h/.c` file as an initial outline.
- c. In `i2c.h`, you will need to create two STRUCTs:
  - i. [I2C\\_OPEN\\_STRUCT](#): Used by a device such as the Si7021 module to open an i2c peripheral
  - ii. [I2C\\_STATE\\_MACHINE](#): Defines the I2C operation and keeps state of the I2C state machine
- d. The [i2c\\_open\(\)](#) function will require two input arguments. The argument types are as follows:
  - i. [I2C\\_TypeDef](#): Providing the base address of which I2C peripheral to be configured
  - ii. [I2C\\_OPEN\\_STRUCT](#): The STRUCT that the device code will use to define or configure the Pearl Gecko I2C peripheral per the device requirements

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- iii. *NOTE: The compiler and/or linker do not always differentiate between capital and lowercase letters. In your `i2c_open()` function prototype, ensure that your variables that you declare as your input arguments have a completely different name than your TypeDef STRUCTs; `I2C_OPEN_STRUCT`.*
  - 1. *Incorrect usage: `I2C_OPEN_STRUCT *i2c_open_struct`*
  - 2. *Correct usage: `I2C_OPEN_STRUCT *i2c_setup`*
- e. `I2C_OPEN_STRUCT`:
  - i. The open structure variables will come from several documents. You will now begin to create the `I2C_OPEN_STRUCT` TypeDef definition in `i2c.h`.
    - 1. Review the HAL documentation for `I2C_Init()` function. All variables required for this function's `I2C_Init_TypeDef` should be included in the `I2C_OPEN_STRUCT` structure.
    - 2. Review the Pearl Gecko Reference Manual and Data Sheet for I2C for all values required to route SCL and SDA from the I2C peripheral to the Pearl Gecko external pin. These values / bits must be included in the `I2C_OPEN_STRUCT` structure. Refer to the LETIMER0 implementation as an example. As in a cake "recipe," this STRUCT must define multiple things to initialize (bake) such as the I2C peripheral's `I2C_Init()` library function and other pieces of information such as routing to external I/O pins.
      - a. What two things must be set for each SCL and SDA signal to properly route them from the I2C peripheral to the external pins?
      - b. What registers and where are these bits located in each register?
        - i. Are there library enumerations that will place the desired bit configuration for the desired setting in the correct bit position of the register?
    - 3. Holdoff from adding any interrupts into this TypeDef definition until you learn more about interrupts later in this assignment.
  - f. The I2C driver should include a function for each of the following operations. Define the functions in the function prototype section of `i2c.h` or `i2c.c` where appropriate and create the functions in `i2c.c`:
    - i. `i2c_open()`; (global function)
    - ii. `i2c_bus_reset()`; (private function)
  - g. To support all the different possible i2c peripherals of the Pearl Gecko, there will be two `i2c_open()` function input arguments which include the following:
    - i. `I2C_TypeDef`
      - 1. Similar to the LETIMER\_TypeDef, this STRUCT will be used to send the base address of the desired I2Cx peripheral, such as I2C0, I2C1, etc
    - ii. `I2C_OPEN_STRUCT`



## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

h. Use the values of these two arguments to initialize the i2c peripheral defined by the I2C\_TypeDef argument in `i2c_open()`. Your code must support opening either the I2C0 or I2C1 peripheral, Best Coding Practice. **Your grade will depend on Best Coding Practices.**

- i. At the start of `I2C_open()`, the function must enable the clock to the appropriate I2C peripheral
  1. By default, in this low energy micro-controller, the peripheral's clock is disabled. If you do not enable the clock to the I2C peripheral, your writes to the I2C peripheral registers will not occur, but the code will appear to work.
    - a. Since there are multiple I2C peripherals in the Pearl Gecko, your I2C driver open should have an if statement to enable the clock to the appropriate peripheral based on the `i2c_open()` I2C\_TypeDef input argument.
  2. We are going to proactively develop a test to verify proper I2Cx clock operation. In your code, you will be writing to a register directly or indirectly. An indirect example is calling the `I2C_Init()` function. Before your first write to an I2Cx register in your initialization of the I2C peripheral, add the lines of code below. I2Cx is pseudo code that references your input argument found in `i2c_open()` that specifies whether you are using I2C0 or I2C1 peripheral. It is the input argument defined as I2C\_TypeDef.

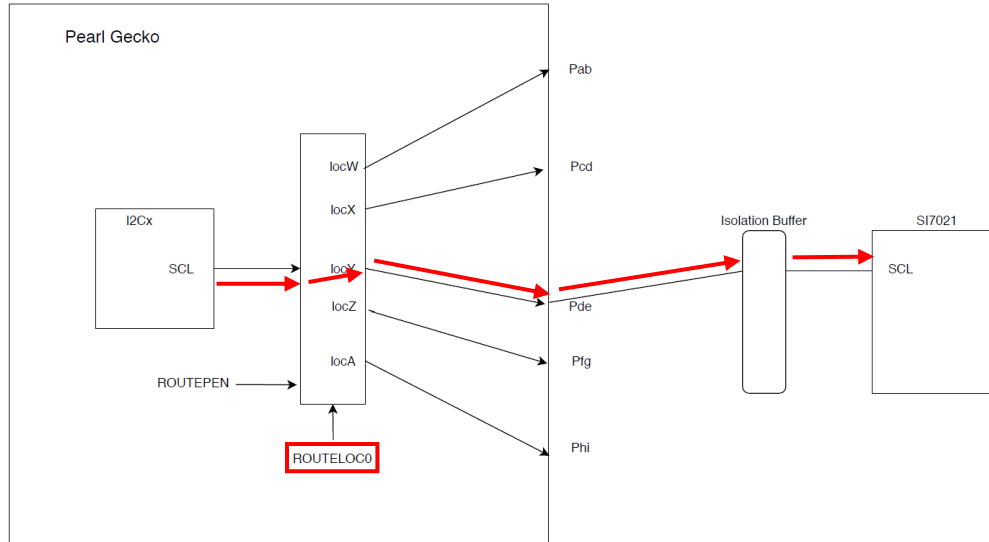
```
if ((I2Cx->IF & 0x01) == 0) {  
    I2Cx->IFS = 0x01;  
    EFM_ASSERT(I2Cx->IF & 0x01);  
    I2Cx->IFC = 0x01;  
} else {  
    I2Cx->IFC = 0x01;  
    EFM_ASSERT(!(I2Cx->IF & 0x01));  
}
```

I2Cx is the peripheral that you are opening/initializing

3. What is this proactively developed test verifying/validating?
4. **DEBUG HINT:** *If after the peripheral open function is completed and you notice that the registers in the debugger have not been modified, the most likely causes of error are:*
  - a. *The clock to the peripheral is not ENABLED*
  - b. *The clock tree to this peripheral is improperly configured*
- ii. After you have added code to validated whether the I2C clock is operational, the next task is to initialize the I2C peripheral using the `I2C_Init()` em\_lib routine. This function will require that you create a local `I2C_Init_TypeDef` STRUCT that you will initialize via the input argument of type `I2C_OPEN_STRUCT` of the `i2c_open()` function prototype found in `i2c.h`.

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- iii. After you have initialized the I2C peripheral, you will need to route the I2C SCL and SDA signals from the internal I2C peripheral to appropriate pins of the Pearl Gecko.
  1. *Going back to the steps to configure the `gpio_open()` function, you have already determined the Pearl Gecko port and pin numbers, Pxy, for both the I2C SCL and SDA signals*



The I2Cx ROUTEPEN register enables the output from the internal routing de-multiplexer

The I2Cx ROUTELOC0 sets the select lines to the routing de-multiplexer

2. *You will need to route both signals, but we will go through the SCL signal as an example.*
3. *With the SCL port and pin information known, open up the Pearl Gecko EFM32PG12 data sheet and search for the Pxy pin until you get to a table of alternate pin functions similar to the screenshot below. Your Pxy pin should be under the GPIO Name column. Please note, the example below may or may not be the proper Pxy for this project.*

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

GPIO Name	Pin Alternate Functionality / Description			
	Analog	Timers	Communication	Other
PC6	BUSBY BUSAX	TIM0_CC0 #11 TIM0_CC1 #10 TIM0_CC2 #9 TIM0_CDTI0 #8 TIM0_CDTI1 #7 TIM0_CDTI2 #6 TIM1_CC0 #11 TIM1_CC1 #10 TIM1_CC2 #9 TIM1_CC3 #8 WTIM0_CC0 #26 WTIM0_CC1 #24 WTIM0_CC2 #22 WTIM0_CDTI0 #18 WTIM0_CDTI1 #16 WTIM0_CDTI2 #14 WTIM1_CC0 #10 WTIM1_CC1 #8 WTIM1_CC2 #6 WTIM1_CC3 #4 LE-TIM0_OUT0 #11 LE-TIM0_OUT1 #10 PCNT0_S0IN #11 PCNT0_S1IN #10	US0_TX #11 US0_RX #10 US0_CLK #9 US0_CS #8 US0_CTS #7 US0_RTS #6 US1_TX #11 US1_RX #10 US1_CLK #9 US1_CS #8 US1_CTS #7 US1_RTS #6 LEU0_TX #11 LEU0_RX #10 I2C0_SDA #11 I2C0_SCL #10	CMU_CLK0 #2 CMU_CLKI0 #2 PRS_CH0 #8 PRS_CH9 #11 PRS_CH10 #0 PRS_CH11 #5 ACMP0_O #11 ACMP1_O #11 ETM_TCLK #3

4. Now search under the desired alternate functionality under the Communication column cross reference with your Pxy number. For this example with Pxy = PC6, under communication column you will find I2C0\_SCL #10. The #10 indicates the location number to program in the SCLLOC field of the I2C0->ROUTELOC0 register.
5. NOTE: It is at this time while looking for the alternate routing location number that you can determine whether you will be using I2C0 or I2C1. Some alternative pins allow either I2C0 or I2C1 to route to it, so it will be up to the programmer, you, to decide which one to use. Save the information of which I2Cx peripheral that you will be using to implement the Si7021 I2C temperature read.

### 15.5.19 I2Cn\_ROUTELOC0 - I/O Routing Location Register

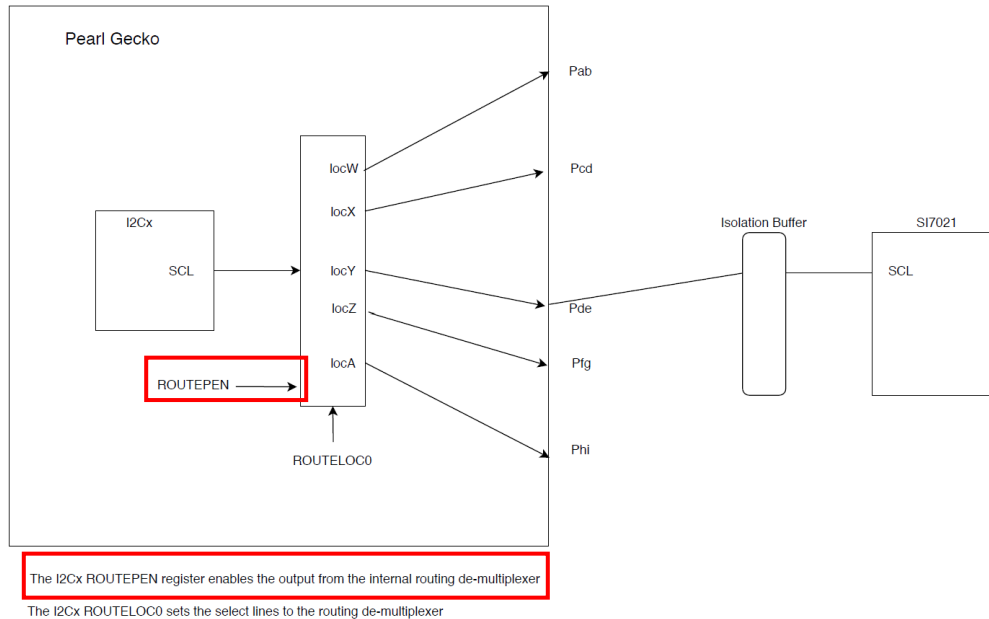
Offset	Bit Position																															
0x048	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset																			0x00													
Access																			RW													
Name																			SCLLOC							SDALOC						

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

Bit	Name	Reset	Access	Description
31:14	Reserved	To ensure compatibility with future devices, always write bits to 0. More information in <a href="#">1.2 Conventions</a>		
13:8	SCLLOC	0x00	RW	<b>I/O Location</b> Decides the location of the I <sup>2</sup> C SCL pin.
	Value	Mode	Description	
	0	LOC0	Location 0	
	1	LOC1	Location 1	
	2	LOC2	Location 2	
	3	LOC3	Location 3	
	4	LOC4	Location 4	
	5	LOC5	Location 5	
	6	LOC6	Location 6	
	7	LOC7	Location 7	
	8	LOC8	Location 8	
	9	LOC9	Location 9	
	10	LOC10	Location 10	
	11	LOC11	Location 11	

6. What is the ROUTELOCO value for SCL for this project?
  - a. What enumeration do you use to place the value in the correct location of the register?
  - b. For a field and not a single bit in a register, it's enumeration is built up as follows:
    - i. PERIPHERAL\_REGISTER\_FIELDNAME\_VALUE
    - ii. Example above:
      1. I2C\_ROUTELOC0\_SCLLOC\_LOC10
7. What is the ROUTELOCO value for SDA for this project?
  - a. What enumeration do you use to place the value in the correction location of the register?
- iv. After you route the signals to the external Pearl Gecko pins, you must enable the output of the demultiplexer to output internally routed signals onto the external pins via the I2Cx->ROUTE PEN register or the enable. Remember you must do it for both the SCL and SDA signals and they must be placed in the proper register field locations.
  1. **HINT: Remember the power of Boolean Multiplication.**

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN



- i. We will hold off from enabling interrupts until we have a plan or a flow chart on how to implement the I2C interrupt driven state machine.
- j. We will stop here in developing `i2c_open()` and come back to it later in this assignment.

### 5. `i2c_bus_reset()` function:

- a. Next, you must develop a routine to reset the I2C state machines of the Pearl Gecko I2C peripheral as well as reset the I2C state machines of the external I2C devices such as the Si7021
- b. The input argument to this function should just be the `I2C_TypeDef`
  - i. `I2C_TypeDef` specifying which I2C internal peripheral
  - ii. By providing this input argument, it enables this single reset function to reset any of the I2C busses connected to the micro-controller.
- c. It is best practice to have a reset solution that is independent of the high frequency clock or performance of the CPU. The solution should also adjust to the I2C bus frequency defined upon the configuration of the I2C peripheral.
  - i. The above implies that we will use the I2C peripheral itself to reset the I2C bus
- d. Reading the I2C chapter of the Reference Manual, it states the following regarding resetting the bus that defines 2 critical steps:
  - i. A bus reset can be performed by setting the START and STOP commands in I2Cn\_CMD while the transmit buffer is empty. A START condition will then be transmitted, immediately followed by a STOP condition
- e. Before we can leave the reset function, we must ensure that the reset occurred. You can verify that the above has completed by checking the MSTOP, Master Stop, bit in the interrupt flag bit set in the IF register. The MSTOP bit is set after the completion of the STOP Command.

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- i. A stall using the `while(!(i2c->IF & I2C_FLAG_MSTOP));` will perform this hold
  - ii. The MSTOP bit may have already been set in the IF register, so the IF register must be cleared before the reset operation.
- f. Below is the sequence of a proper reset function
  - i. Save the state of the I2C->IEN register
  - ii. Disable all interrupts using the I2C->IEN register
    - 1. This will prevent the Interrupt Service Routine from being called while performing a reset
  - iii. Clear the Interrupt Flag (IF) register via writing to the Interrupt Flag Clear register
    - 1. Ensure the interrupts are cleared before we stall waiting for an interrupt to occur
  - iv. Clear the I2C transmit buffer
  - v. Perform the simultaneous setting of the START and STOP bits in the I2C command register
  - vi. Stall until the STOP has been completed by checking the MSTOP bit in the Interrupt Flag register
  - vii. Clear all interrupts that may have been generated by the START/STOP reset commands by writing to the Interrupt Flag Clear register
  - viii. Put the state back of the Interrupt Enable register by using the saved state at the start of this routine
  - ix. Lastly, reset the micro-controller I2C peripheral state machine itself which is accomplished by writing the ABORT bit in the I2C CMD register

### 6. Si7021.c:

- a. The Si7021 is an I2C temperature and humidity sensor which is integrated on the Pearl Gecko starter kit. The physical implementation of the Si7021 on the Pearl Gecko starter kit creates dependencies for the I2C driver, so you will now be focusing on the Si7021 before completing the I2C driver.
- b. Create your Si7021.h/.c files in the appropriate folders of your project. You may want to copy the framework from another .h/.c file as an initial outline.
- c. As a reminder, the I2C routines will be independent of the specific or application implementation of the Si7021 on the Pearl Gecko Starter Kit.
- d. It is time to develop your `si7021_i2c_open()` function.
  - i. You will need to complete the configuration of a local STRUCT of type `I2C_OPEN_STRUCT`
    - 1. This STRUCT contains the information to complete the set-up of the I2C external devices such as the I2C bus speed
    - 2. To help you initialize all the elements in `I2C_OPEN_STRUCT`, refer to the `i2c_init()` function in the Hardware Abstraction Level, HAL, documentation for `I2C_Init_TypeDef` input argument and Si7021 data sheet

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- a. You are required to run the I2C bus at the maximum frequency to minimize the energy consumed during the I2C bus communication
- b. The I2C maximum frequency will be determined by the I2C device with the lower of the maximum I2C frequency specifications
  - i. You will need to determine the maximum I2C of both the Pearl Gecko and the Si7021
  - ii. In the `I2C_Init_TypeDef`, there is a variable in the structure for frequency. To find the I2C frequency definitions in the HAL documentation, while in the HAL document's I2C chapter, search for `I2C_FREQ`, I2C frequency. You will find three definitions. Which is the most appropriate based on your research on the lowest I2C clock speed for all I2C devices on the bus? At this stage, the Pearl Gecko and the Si7021. Use this definition to configure your I2C bus frequency in your `I2C_OPEN_STRUCT`.
- c. To provide enough time for the pull-up to change the I2C bus from a LOW to a HIGH, as the I2C bus frequency increases, the Pearl Gecko must begin to release the I2C signal earlier in the clock period, changing the ratio of LOW to HIGH. To determine the CLK Ratio, you will also use the Pearl Gecko HAL documentation. Search and go to the information on the `#define` statement used to set the desired I2C clock frequency. In this documentation, it will specify the clock frequency ratio required to meet the clock setup and hold times for a particular I2C bus frequency.
  - i. With the required clock ratio now defined, go to the HAL documentation for the `I2C_Init_TypeDef` and search the appropriate `TypeDef` for the clock ratios. Use the `#define` clock ratio that matches the required I2C clock ratio for the I2C bus frequency that you are configuring the bus.
- e. With the Si7021 and I2C routines completed to open the Si7021, it is time to initialize the I2C peripheral via the Si7021 open function. Where should this Si7021 open function call be made?
  - i. HINT: Where are the other peripheral opens occur?
  - ii. Above, we discussed one arguments to pass to the `i2c_open()` function, the local `STRUCTs` of type `I2C_OPEN_STRUCT`
  - iii. The second argument should be the I2C peripheral that you have chosen based on I2C routing information that you have chosen earlier

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- f. It is time to compile your code and see if it operates as a completed Lab 3 assignment. You have not written the code to read from the Si7021 yet or even to initialize the Pearl Gecko I2C peripheral.
7. Use the Debugger Register view to validate that you have configured in I2C peripheral correctly.
  - a. Are these correctly set in the I2C registers?
    - i. I2C bus speed?
    - ii. The I2C symmetry setting?
    - iii. Routing of the internal SCL and SDA lines to the external GPIO?
    - iv. Output enabling the internal SCL and SDA lines from the I2C module?
  - b. Are the SCL and SDA lines correctly set in the GPIO registers?

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

8. **NOTE:** The instructing staff will not help either the development of the Si7021 or I2C reads functions until you have completed a successful initialization and verification of the I2C peripheral via the S0721 open function.
9. **I2C Start and I2C Interrupt Service Routine function:**
  - a. It is Best Coding Practices to develop efficient code. In the case of a Low Energy micro-controller, it most likely refers to an interrupt driven routine. You will be developing an I2C read operation that is interrupt based. Now it is time to develop your I2C software ladder flow chart. Every arrow going from the Si7021 “side rail” to the Pearl Gecko “side-rail,” you will want the Pearl Gecko to be in its lowest energy mode to wait for the Si7021 to respond back to the Pearl Gecko via the I2C peripheral to generate the proper interrupt.
  - b. **Software Ladder Chart:**
    - i. Software Ladder charts are extremely helpful when a series of events are required and there are dependencies between the two devices.
    - ii. Develop a Software Ladder Chart for reading the temperature from the Si7021 using the Si7021 “no-hold” read command, Command Code 0xF3.

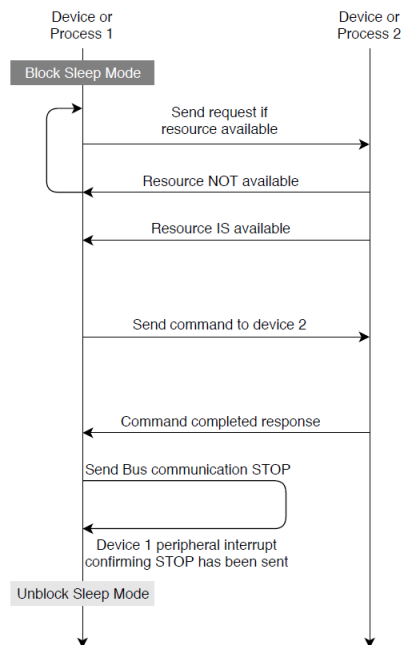
Command Description	Command Code
Measure Relative Humidity, Hold Master Mode	0xE5
Measure Relative Humidity, No Hold Master Mode	0xF5
Measure Temperature, Hold Master Mode	0xE3
Measure Temperature, No Hold Master Mode	0xF3
Read Temperature Value from Previous RH Measurement	0xE0

1. Go to the Si7021 datasheet and find the diagram that represents the sequence of operations to read the Si7021 temperature using the “no-hold” read command
2. The Si7021 datasheet diagram can easily be converted to your Software Ladder Chart.



## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

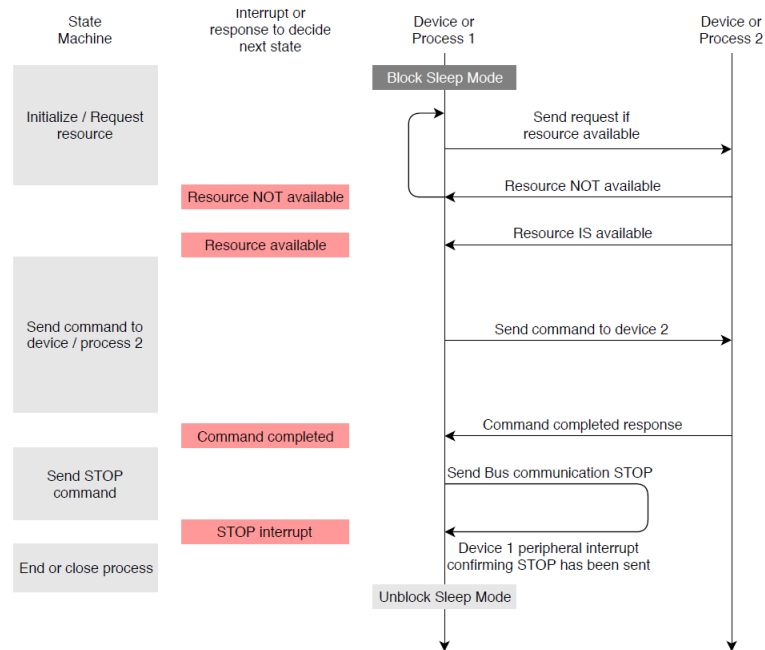
- a. Refer to your pre-lab video converting an I2C timing (communication) diagram into a Software Flow Chart
- b. You are not required to implement the optional Checksum byte read or compare
- iii. The left “side-rail” of the SW ladder will be your program, the Pearl Gecko, and the right “side-rail” of the ladder will be the Si7021
- iv. *The rungs of the ladder indicate data transfer from one device, “side-rail,” to the other*
  1. *These rungs indicate dependencies between the devices*
  2. *The other device cannot proceed until the data dependency from the other device has been resolved / receive*
  3. *When there is dependency that is stalling a device from proceeding, indicate on the Software Ladder Flow Chart what the device is waiting for to proceed such when the Pearl Gecko is waiting for a response from the Si7021, what signal / interrupt will indicate to go to the next step or state of your flow chart such as an ACK.*
- c. *If an iteration or loop is required, it should be indicated by a looping arrow*
- d. *Below is an example of a SW ladder flow chart*



- e. *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/>
- f. *You can develop or identify your states by using your Software Ladder Flowchart. If we go back to the flowchart example, you can identify the states by where you*

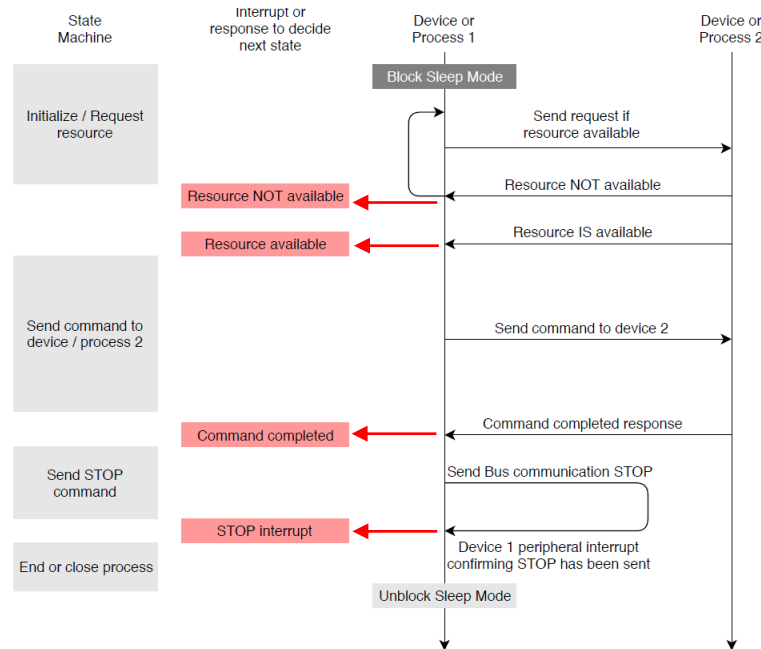
## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

*expect the interrupts to occur. In an interrupt driven state machine, the states will transition due to the reception of an interrupt. At each interrupt, a decision will be made to go to the next state or to repeat the current state. Since the decision point are the interrupts, the interrupts define the boundaries of the states. Please see the below updated Software Ladder Flowchart example.*



- g. Now, go back to your flow chart and add the states of the state machine defined by the flow chart using names that clearly define what the states represent or are performing. States labeled as state 1 or 2 are like Magic Numbers and should not be used.
- h. *With the internal signals routed, it is time to enable the interrupts that will be required to implement your Si7021 software ladder flow chart. For this I2C peripheral, what interrupts will you need to enable for the State Machine that you will be implementing?*
  - i. *HINT: Take a look at your I2C Software Ladder Flow Chart*

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN



ii. In looking at a flow chart that will be used to implement an interrupt driven state machine, each arrow that comes from the right is an interrupt, and thus each of these interrupts must be enabled when you open the driver. With the state machine completely implemented within the driver, you do not need to pass this information from the Si7021 to configure the `i2c_open()` function. They should be hard-coded enabled via the I2C open driver. In our example above, the interrupts to enable would be:

1. `IEN = Resource_NOT_Available`
2. `IEN |= Resource_available`
3. `IEN |= Command_completed`
4. `IEN |= Stop`

iii. In your I2C flowchart, some interrupts are very apparent. For example, an arrow labeled as ACK or NACK. Other arrows indicate information that you will need to investigate the interrupt source. In your flowchart, you should have an arrow from the Si7021 to the Pearl Gecko that specifies MS byte and LS byte. These arrows indicate the transfer of data. To determine this interrupt, go to the Pearl Gecko Reference Manual and look up the I2C Interrupt Flag Register. Read all the interrupts.

1. Which interrupt indicates that data is ready to be read from the I2C RX buffer?

iv. Go to your I2C flowchart, and like the example above, update your flow chart to include the interrupts associate to the left of each arrow that touches the Pearl Gecko rail from the right. You will use this updated flow chart later to develop your state machine.

- i. What is a best practice to do before enabling these interrupts?
  - i. Clear them by using the Interrupt Flag Clear, IFC, register

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- j. *After enabling the interrupts at the peripheral level, you will need to enable the interrupts at the CPU level by calling the appropriate `NVIC_EnableIRQ()`*
  - i. *With the goal for this open function to be generic between opening either I2C0 or I2C1, you will need an IF statement using the input argument of `I2C_TypeDef` to determine which `NVIC_EnableIRQ()` to enable. Similar to the IF statements at the top of the driver open function that were used to enable the clock to the appropriate peripheral based on the `i2c_open()` `I2C_TypeDef` input argument.*
  - ii. *A good way to find the documentation that you require is to find a similar reference already in your project or an example project. From a previous assignment, we used `LETIMER0_IRQn` as the enumeration for the ARM Nested Vector Interrupt Command, NVIC, operation to enable interrupts to the ARM CPU of the Pearl Gecko. Go to the Pearl Gecko HAL documentation and search for `LETIMER0_IRQn` to locate the table of all the enumerations used to enable the Pearl Gecko vector interrupts. Search for one that looks appropriate for the I2C peripheral that you are using, I2C0 or I2C1.*
  - iii. Making this driver independent of the peripheral will be included in this lab assignment's Rubric.

10. NOTE: The instructing staff will not help either the development of the I2C state machine or I2C start functions until the Software Ladder Chart has successfully been completed and reviewed.

- a. For an interrupt driven I2C function, at a minimum, the following functions will need to be defined and developed in your `i2c.h/.c` files:
  - i. `void I2Cx_IRQHandler(void)`: Interrupt Service Routine, ISR, Handler for I2Cx peripheral
    - 1. where x represents 0 if using I2C0 and 1 if using I2C1
  - ii. `I2C Start function`: Initializes a private STRUCT in `i2c` that will keep state of the progress of the I2C operation. This state information will be stored in a static STRUCT in `i2c.c` of type `I2C_STATE_MACHINE`. In your case, the read of the Si7021 read temperature register.
    - 1. This will not be called by the Interrupt Service Routine, but it will initiate the I2C read access of your flow chart resulting in sending out the START bit plus Device Address w/ the Write bit which will take your state machine to the first point of waiting for an Interrupt.
    - 2. The input arguments for this function will be the information required to initialize your `I2C_STATE_MACHINE_STRUCT` to initiate
- b. In developing a module, not all functions should be made global by including their functional prototypes in the associated `.h` file but should be private to the

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

module like the private variables. For example, if another module accessed the functions involved with the I2C state machine operation, it most likely would result in the state machine failing. Add the functional prototypes for the below functions in your .c file. Since other files do not include .c files, these functions will become private. Add their functional prototypes below your private variable declarations in i2c.c. Since these functions (function prototypes) are private, they should be declared as static and above any functions that are developed in the .c module.

- i. **I2C “interrupt state machine” function:** There should be a private function to service each of the interrupts required to support the state machine defined in the software flowchart.
- c. The I2C driver will need to have a private STRUCT to control the state of the I2C read operation. This structure is defined in i2c.c to keep it private. This state machine will be kept solely within the i2c.c scope instead of using the scheduler. After each interrupt is received, the Interrupt Service Routine, will call the appropriate I2C static (private) function to progress the state machine directly.
  - i. *Why would this driver not use the scheduler to handle its sequencing?*
  - ii. *The project is to design a modular and encapsulated driver. Encapsulation means the driver is self-contained and not dependent on user or application code.*
  - iii. *Why does the Interrupt Service Routine call functions and not embed the code directly in the ISR?*
    1. *If you did embed the code directly in the ISR, what benefits would arise from it?*
- d. Now create in I2C.h the structure that we referred to earlier, **I2C\_STATE\_MACHINE**.
  - i. The **I2C\_STATE\_MACHINE STRUCT** will comprise of two set of elements
    1. An element specifying the current state of the state machine such as whether in the above software flow chart example is in state “Initialize / Request Resource” or “Send Command to device / process 2”
      - a. *The definition of each state must be unique and does not require a coding scheme as in the event scheduler variable where each event is defined by a unique bit. To define unique states, each state just requires a unique number. In defining your states, you could use #defines and manually ensure that each state is defined by a unique number or you could allow the c-code enum structure to do it for you. **Manually** generally refers to additional effort and is more error prone compared to automatically.*
      - b. *In C, enumerations are created by explicit definitions (the **enum** keyword by itself does not cause allocation of storage) which use the **enum** keyword and are reminiscent of struct and union definitions: ... C also allows the*

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

programmer to choose the values of the enumeration constants explicitly, even without type. (Wikipedia)

- c. While defining an enum, if you do not specify a specific value to an enumeration, the enum operator will sequentially assign values **automatically**. If the first variable is not assigned a specific number, it will begin assigning values at 0.
- d. Let's go back to our example flow chart. For each state, you will require one instance in your enum representing each of the states in your flowchart. This enum should be defined in `i2c.h` since the state machine resides completely in the `i2c` module. Below is a pseudo code example.

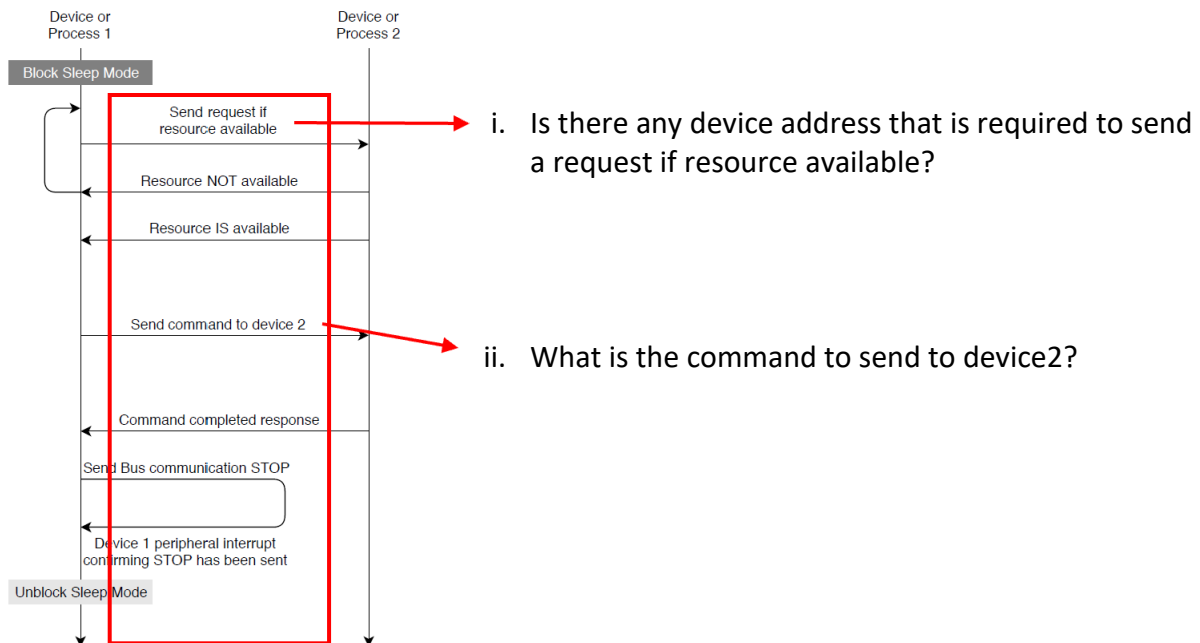


- e. In the above example, the states in the enumeration will have the following values:
    - i. Initialize / Request Resource = 0
    - ii. Send Command to device / process 2 = 1
    - iii. Send Stop Command = 2
    - iv. End or Close Process = 3
  - f. Using enums with automatic assignments is less effort and less error prone, thus using enums to define the state machines' states is best practices.
  - g. Using enum will be included in this lab's rubric.
2. The second set of elements is all the additional information required by the state machine to process an event when an interrupt occurs. For example, the driver is designed to support both I2Cx peripherals, so the **STATE MACHINE STRUCT** must keep track whether it is using I2C0 or I2C1. Another example is that

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

multiple devices could be on the I2C bus, so the I2C device address of the I2C device that is being accessed must be included in the [STATE MACHINE STRUCT](#).

- ii. How to determine what other elements will be required by this second set of elements in the [STATE MACHINE STRUCT](#) to be saved between function calls?
  1. To determine what other elements are required by your interrupt driven state machine, you can look at each rung of your software ladder chart.
    - a. What is required to be saved between interrupts to enable each rung?
      - i. For example, in your I2C ladder flow chart, what is required in the [STATE MACHINE STRUCT](#) after you have received the ACK responding to the Pearl Gecko sending the Start bit and Slave Address to the Si7021 to process to the next rung of the ladder?
      - ii. What is required to be used by the other rungs of the software flowchart?
    - b. Go back to your flowchart for help. Below is an example of using the rungs of our example flow chart to determine what is required by our [STATE MACHINE STRUCT](#)



2. Remember, the code in i2c.c is a generic driver. The code must not assume anything of the board implementation or application. What additional information is required to be saved in the private

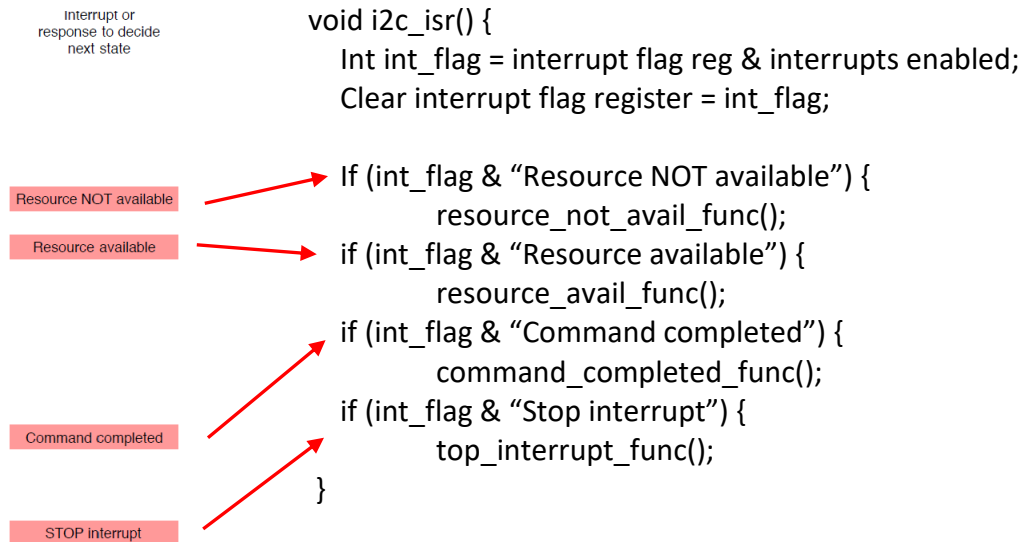
## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- `I2C_STATE_MACHINE_STRUCT` to preserve the information required for the interrupt driven state machine?
- Which Pearl Gecko I2C peripheral?
  - I2C device address?
  - I2C register address being requested?
  - Write or Read?
  - Pointer of where to store a read result or get the write data?
  - How many bytes to transfer?
- The vagueness of the `STATE_MACHINE_STRUCT` described above is for you to learn how to develop the required STRUCT / information that will be required as well as some elements will be required by all implementations while other elements will be unique to your implementation
- e. To initiate a read of the Si7021, your application code will perform a function call to your `I2C Start Function`.
- What input arguments will be required to initiate a read request?
    - The information to configure your STRUCT, `I2C_STATE_MACHINE`, in `i2c.c`
  - NOTE: In addition to all the information that will be required to initialize the I2C state machine, a pointer must be passed from the Si7021 to the I2C driver as an argument for the I2C to return the results of the Si7021 read.*
    - The Si7021 read function will request the read from the I2C and will exit before the read is completed. The variable pointer for the resulting read data must be a private variable inside Si7021.c. We will discuss this more later.*
- f. To verify that the I2C peripheral is ready before you begin an operation, insert the following `EFM_ASSERT` statement as the first instruction in your I2C start function:
- `EFM_ASSERT((I2Cx->STATE & _I2C_STATE_STATE_MASK) == I2C_STATE_STATE_IDLE);` // X = the I2C peripheral #
  - This `EFM_ASSERT` will “trigger” if your I2C peripheral has not completed its previous I2C operation
- g. The I2C peripheral can only work down to EM1 since it is a high frequency peripheral. You must have a call to `sleep_block_mode()` before you begin the I2C bus operations that will block the Pearl Gecko from going to sleep below EM1. The second instruction in your I2C start function should be:
- `sleep_mode_block(XX);`
  - Is setting XX = 2 OK?*
    - No. 2 is a magic number. It is best coding practices not to use magic numbers. In `i2c.h`, there should be a definition defining what energy mode to block entry.*
    - `#define I2C_EM_BLOCK EMx` // x = first mode it cannot enter



## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

3. The EMx should be define statements found in your sleep routine .h file
  - a. If you will be using EMx #defines from your sleep routine .h file, what must you include in your i2c.h file?
- h. After preventing the Pearl Gecko from going to sleep in an energy mode below its “active state”, you must initialize your private structure that will be used by the state machine.
- i. After initializing the private variable, you perform the I2C instructions up to the first arrow from the Pearl Gecko “side-rail” of your Software Ladder Flow Chart to the Si7021 “side-rail.”
- j. This function is now complete. As it exits, your program should return to the application function that called your [I2C Start Function](#) and finally return back to the main.c loop, and if no events are scheduled, the Pearl Gecko will go to sleep until the Si7021 performs its arrow from the Si7021 “side-rail” to the Pearl Gecko “side-rail” of your Software Ladder Flow Chart.
- k. Now, let's look at what is required within your I2C Interrupt Service Routine
  - i. As in all Interrupt Handlers, it is best practice to save the interrupts of interest and then clear all these interrupts
  - ii. To determine what interrupts should be evaluated in your ISR, let's go back to the example software flowchart where you specified the different interrupts of interest. Below is pseudo code from our software flowchart example:



### 11. The private functions to process the interrupt driven state machine

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- a. Inside the `I2Cx_IRQHandler()` function, for example, your IF statement for an ACK interrupt should call your `I2C ACK Function`
- b. Does your `I2C ACK Function` require any input arguments?
- c. *The `I2C ACK Function` in the I2C driver should respond to the ACK based on the current state of the state machine in the private I2C state machine structure to perform all the I2C instructions from the Si7021 “side-rail” arrow until you reach the Pearl Gecko “side-rail” arrow to the Si7021.*
  - i. *Note: Inside your `I2C ACK Function`, you will need to update your I2C private State Machine structure to indicate the new state of your state machine after servicing the call to it.*
  - ii. *One common method is to implement a state machine using switch / case statements inside your ACK and NACK functions. Below is an example using pseudo code:*

```
switch (current_state)
case state_1:
    perform an action
    current_state = next_state
    break
case state_3:
    perform an action
    current_state = some other next_state
    break
```

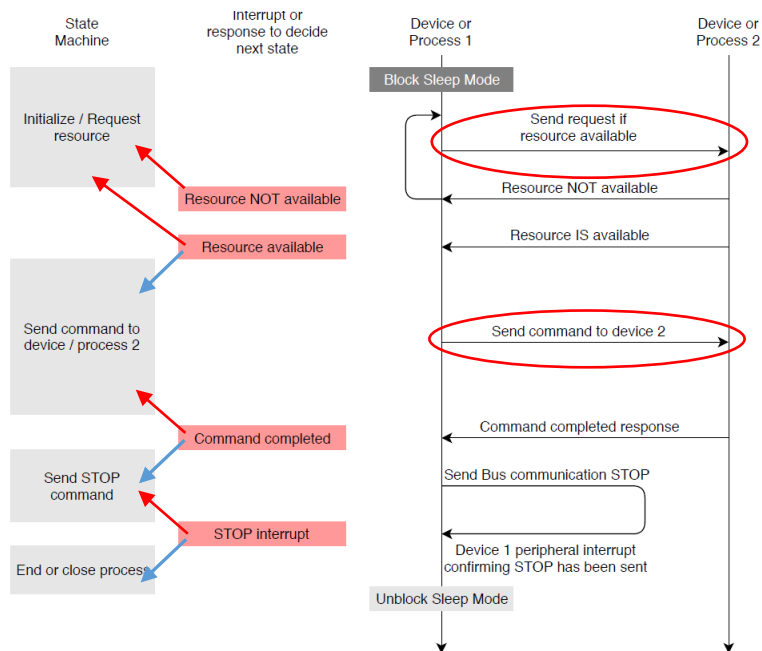
- iii. *It is best coding practices for all possible transitions to define what should happen in each state. Based on the transition input, in the case of an interrupt driven state machine the interrupt, each state should be defined. Based on the interrupt, it will progress the state machine, have no effect, or indicate an illegal transition based on the current state.*
  1. *This best coding practice will be included in the lab rubric*
- iv. *For best coding practices in developing code and finding any coding errors as soon as possible, an `EFM_ASSERT(false)` function call should be made on any illegal cases of your state machine. For example, if you receive an interrupt in a state where the interrupt such as a NACK is not expected, you should perform an `EFM_ASSERT(false)` for the system to halt and indicate an error for debugging as early as possible. The earlier your code can indicate an error; it will minimize the time to debug.*
- v. *Similar to indicating interrupts received for an illegal state, if you use a switch / case statements, it is best coding practices to include a default condition which calls `EFM_ASSERT(false)` if entered. The default case will catch any undefined or possible illegal passes through the switch / case statement of the state machine.*

```
switch (current_state)
```

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

```
case state_1:
    perform an action
    current_state = next_state
    break
case state_2:
    EFM_ASSERT(false);
    break
case state_3:
    perform an action
    current_state = some other next_state
    break
default:
    EFM_ASSERT(false);
    Break
```

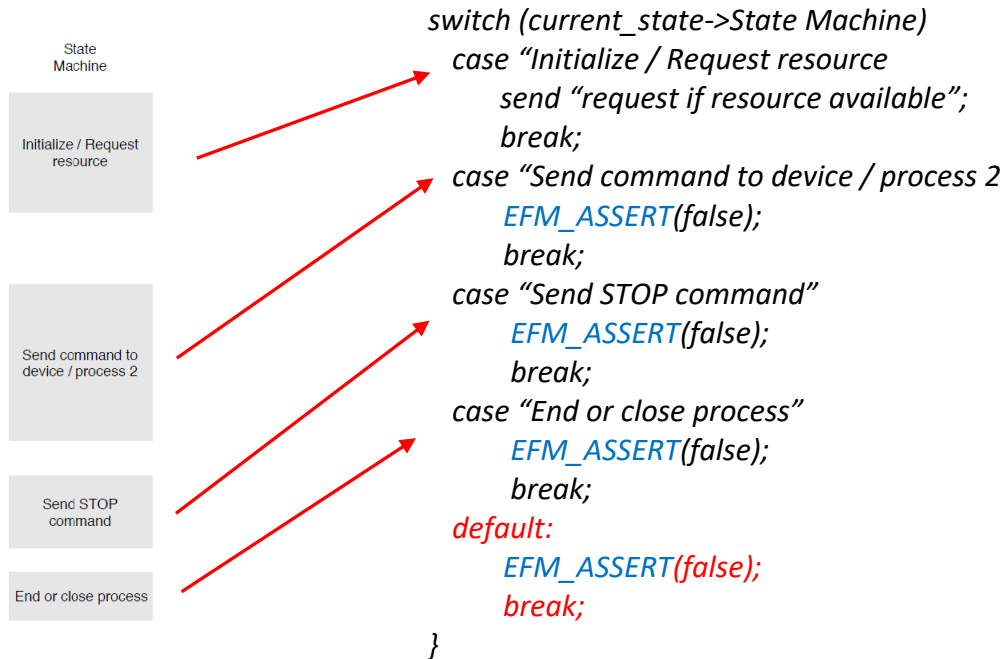
vi. Let's go back to our updated example flow chart



- a. In this example, we have an interrupt called *Resource\_NOT\_available*. Our interrupt service routine for our above example calls a function *resource\_not\_avail\_func()*; upon receiving this interrupt. Based on the above best coding practices, this function may look like the following example pseudo code:

```
void resource_not_avail_func (void) {
```

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN



### b. Points to note:

- i. The flow chart shows that *Resource\_NOT\_available* interrupt must resend the request for resource availability
  - ii. Since the arrow loops back up within the current state, no update to next state is required
  - iii. The default statement will indicate whether this function was entered by any illegal state, and by the example flow chart, these transitions are not allowed and thus `EFM_ASSERT(false)` would indicate an error.
- c. Now, look at the function called when the *Resource\_available* interrupt occurs:

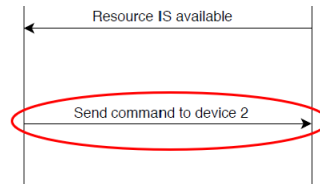
```
void resource_avail_func() {
    switch (current_state->state)
    {
        case Request_Resource
            send "command to device 2"
            current_state->state = Send_CMD_Device_2
            break
        ...
        ...
        ...
        default
            EFM_ASSERT(false)
            break
    }
}
```

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

}

d. Points to note:

- i. The c-code to be developed within the case statements come from your flow chart



- ii. In this example, *Resource\_available* interrupt must execute all rungs from the left to the right rails until the next interrupt, arrow from the right to the left
    1. Send "command to device 2"
  - iii. The flow chart's blue arrow for this interrupt indicates the next state of the state machine
    1. *current\_state->state = Send\_CMD\_Device 2*
- vii. Upon completion of the *I2C ACK Function*, it will return to *I2Cx\_IRQHandler()* to complete the call to this Interrupt Service Routine
  - viii. By performing the response required by your state machine defined in *I2C\_STATE\_MACHINE*, you should have completed the next arrow from the left "side-rail," Pearl Gecko, to your right "side-rail," the Si7021.
  - ix. Now it is time to go back to sleep until the next interrupt from the Si7021
- d. You will build similar functions for all the interrupts required by your software flowchart.
- e. When the I2C operation is completed, you must now *unblock\_sleep\_mode()* the I2C blocked energy mode. The Pearl Gecko should now be able to go back to sleep down into EM3 if no additional pending I2C operations
- i. What interrupt will you use to verify that the I2C peripheral is completed?
    1. Is this interrupt in your flowchart?
  - ii. The state machine must ensure that both the I2C hardware peripheral and the I2C state machine have completed before unblocking its energy mode hold. If the energy mode hold is released before the hardware peripheral has completed, the microcontroller may go to sleep below the I2C's peripheral "active" state and not complete the current I2C bus request.
- f. In addition to unblocking the sleep mode when the I2C operation is completed, it must indicate to the system that the operation is completed by sending a signal to the scheduler using:
- i. *add\_scheduled\_event();*

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- ii. Getting back to the idea that this I2C is a generic driver, it does not know what the event definition for I2C operation complete will be. This I2C operation complete event is an application definition. The complete event information must be passed to the I2C driver when it is opened, `i2c_open()` and saved as a private variable within `i2c.c` to be used for each I2Cx operation when completed. The defined value for this event should be defined in `app.h` with the other `#defined` events.

- iii. From the previous labs, you should have the following events in `app.h`

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

- iv. For this lab, we will use the external device in the event to be serviced instead of the Pearl Gecko such as `SI7021_READ_CB`.

- 1. What value should you define for your Si7021 temp done event?

12. Now compile and try running your code. It should still operate as Lab 3 since we have not written or called the `Si7021_read()` function. It is always good practice to check whether you have broken any code that should not have been broken.

13. Before we go to the next step in the assignment, we can check that the I2C Interrupt handler is properly calling each of your I2C state machine functions. Perform the following and check whether that proper state machine function was entered and processed in the correct case statement in your switch function.


- a. Place a breakpoint at the start of each of your I2C interrupt drive state machine functions

- b. Run your program in the debugger


- c. Let the program run long enough to initialize your I2C peripheral

- d. Pause the state machine and via the register view, go into your I2C registers and set the I2C interrupt flag for ACK using the Interrupt Flag Set (IFS) register

- i. Does the program break within the proper I2C state machine function?

- ii. Using the Step-Into, , within the debugger, does the switch statement process this interrupt as expected?

- e. If not, debug your interrupt handler and state machine switch statement

- f. If yes, perform a software reset, , within the debugger and try the next interrupt that your state machine requires such as NACK. Continue until you have tested all the interrupts.

<<< GOAL TO HAVE COMPLETED UP TO HERE BEFORE THE START OF THE 3rd WEEK OF LAB >>>

### 14. Si7021 – part 2:

- a. Two more functions will need to be written for the Si7021.
  - i. A function that the application code can request the temperature to be read from the Si7021
  - ii. A function to return the temperature read from the Si7021 in degrees Fahrenheit, F

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- b. First, you will use a private variable in Si7021.c that was created to transfer the result from the I2C driver routine back to the Si7021.c routines
  - c. The Si7021 read function should be short. It should just contain the call to I2C start function with the proper arguments set to read the Si7021 temperature value using the NO\_HOLD command address and the pointer to the Si7021 private result variable
    - i. You will pass the Call Back information to be loaded into the I2C state machine STRUCT via this Si7021. Imagine that you may want to have different callbacks for I2C writes and I2C reads. The I2C start function should support both read and write operations.
      - 1. The Si7021\_READ\_CB should be an input argument to the call si7021\_read() function
    - ii. An input argument to i2c\_start() of the operation callback should be added to the i2c\_start() function prototype and function
  - d. The Si7021 temperature conversion functions will read the value of the private variable containing the result of the Si7021 read. The Si7021 data sheet contains the algorithm to convert the Si7021 result into degrees Celsius, C, which will then need to be converted into degrees Fahrenheit, F. This function will RETURN the degrees in Fahrenheit, F. The return from this function should be a float since the requirement is to have the temperature to the tenth of a degree F.
    - i. Note: The call to the temperature function will only occur upon the completion of the Si7021 temp done event which will generate the conclusion of the correct sequence of requesting the I2C read operation, completing of the read, and then converting to degrees F.
    - ii. Please refer to the Si7021 datasheet for the algorithm to convert the value read from the Si7021 into degrees C. You will then need to convert the temperature into degrees F.
15. Now compile and try running your code. It should still operate as Lab 3 since we have not called the Si7021\_read() function. It is always good practice to check whether you have broken any code that should not have been broken.

### 16. Application Code

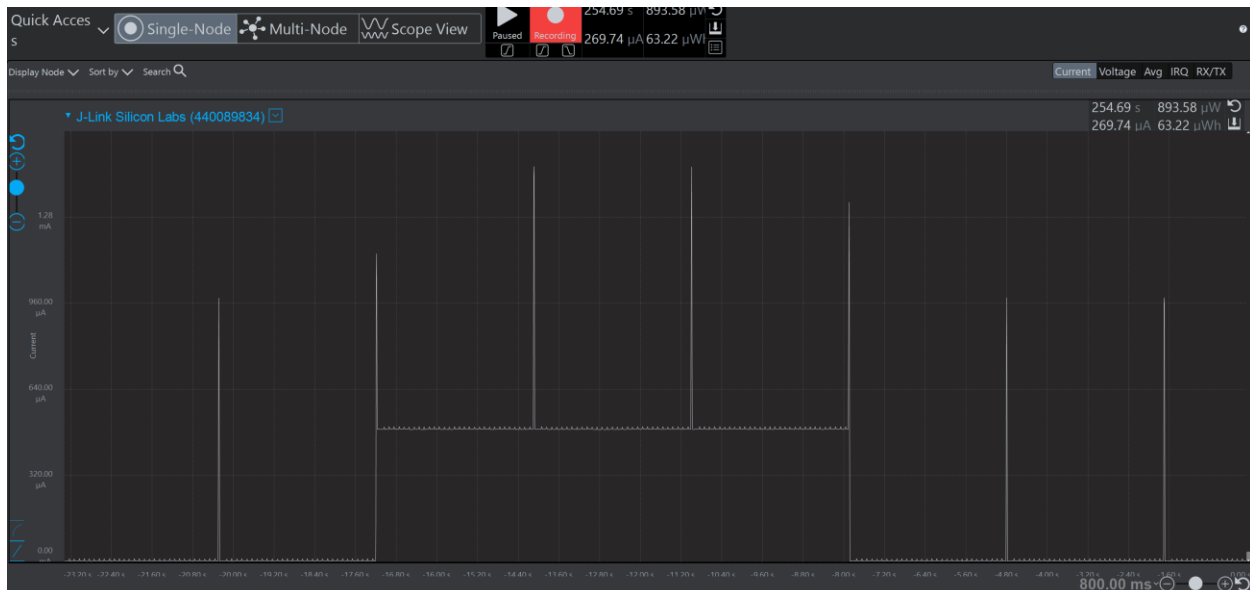
- a. The infrastructure for Lab 4 has now been completed. It is time to modify the application code for the assignment.
- b. First, disable the PWM LEDs from blinking.
  - i. The LETIMER PWM to the GPIO pins can be disabled in the app.c routine that opens the letimer pwm driver by setting the output enables to false.
- c. Modify in app.c the scheduled\_letimer0\_uf\_evt() function.
  - i. Delete all the code in this function that was used to step through the different Energy Modes for the previous lab, Lab 3.
  - ii. Now add a call to your Si7021 routine to read the temperature.
- d. Add a function in app.h/app.c to service the Si7021 temp done event
  - i. The Si7021 temp done event signals the completion of a Si7021 temperature read.

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- ii. The basic sequence of handling the `scheduled_letimer0_uf_evt()` should be followed to service the `Si7021 temp done event`
- iii. With the temperature returned, compare the result to 80.0 degree F
  1. If the value is greater than or equal to 80.0, turn-on / Assert the GPIO pin to LED1
  2. If the value is less than 80.0, turn-off DeAssert the GPIO pin to LED1
- e. The next to last thing to do is to add an event in the scheduler in `main.c`'s `while(1)` loop for the service the `Si7021 temp done event`. If this IF statement is "true," indicating the event must be serviced, it will call the `Si7021 temp done event` service routine.
- f. The final code to add before you try your project is to open the I2Cx peripheral. Go to your `app_peripheral_setup()` function in `app.c` to add a call to your `si7021_i2c_open()` function.

### 17. Now it is time to test your program.

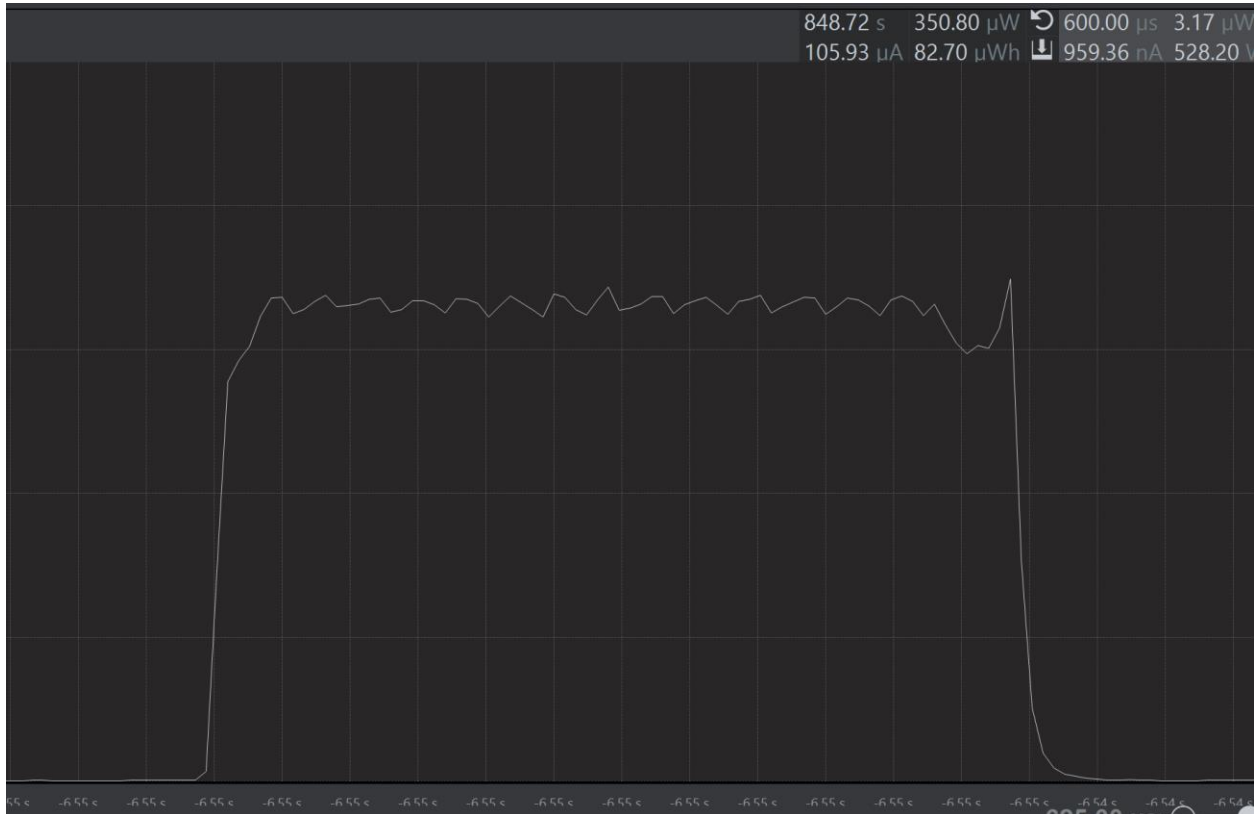
- a. You can verify correct temperature reads by putting a breakpoint in your temperature conversion to degrees F routine
  - i. Use the debugger to view your private variable that contains the results of the Si7021 read. The value of the private variable in Si7021 of the result should be in the range of **23,000 to 30,000**
  - ii. Use the debugger to view your converted temperature. Depending on the temperature of the room that you are in, I would expect the temperature in degrees F will be in the range of 65 to 82F
  - iii. If both of these values look good, try running your program with the Energy Profiler
- b. In the Energy Profiler view, your application should look similar to the screen shot below:





## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- i. The step-up in base current is due to me putting my finger on the Si7021 which took the temperature above 80F and turned-on LED1
- ii. The step-down in base current is due to me taking my finger off the Si7021 and the temperature fell below 80F and turned-off LED1
- c. If the PWM functionality has been turned-off and the LED0 is no longer blinking, why are there periodic current spikes?
- d. Below is a zoomed in picture of one of the current spikes.



- e. Why is the current not stable / flat between the Pearl Gecko waking up and going back to sleep?

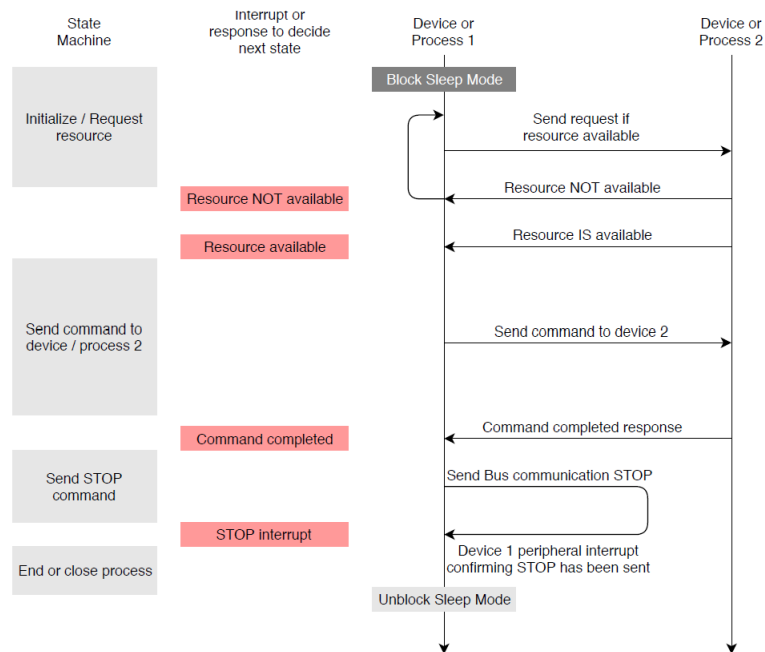
### 18. Doxygen documentation:

- a. A proper doxygen file and function comment block should be developed for all modules including the new modules:
  - i. i2c.c
  - ii. si7021.c
  - iii. app.c (new functions)
- b. Create the exported project's html doxygen html documentation folder.

Deliverables:

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

1. Each person must submit their Lab 4 software ladder chart into the Lab 4 Exported Project canvas folder
  - a. The flow chart should include all the information to implement an I2C Interrupt Driven State Machine. Like the example used in this assignment.



2. Each person must submit their Lab 4 zipped doxygen html folder into the Lab 4 Exported Project canvas folder
3. Each person exports their project as an archived file. Upload the .zip file into the Lab 4 Exported Project canvas folder
  - a. The exported project must have its HFRCO frequency set in main.c to 32 MHz
4. Each person to provide their answers via Canvas/quiz/Lab4 Worksheet

Questions:

Complete the Lab 4 worksheet to complete this assignment.

Point breakdown:

- Software flowchart has a total of 10 points
  - Rubric is located within the Lab 4 Flowchart Assignment
  - The flowcharts must be done individually, and they may be checked for originality
    - Flowcharts that are deemed not to be original will result in a 0 for this portion of the assignment for all flowcharts that are viewed as similar
- Lab 4 has a total of 40 points

## READ ENTIRE LAB ASSIGNMENT BEFORE YOU BEGIN

- Exported project will be graded on:
  - Code fully functional while built using -O2 optimizer
  - Functionality
    - LED 1 turns on when the temperature is read greater than 80F
    - LED 1 turns off when the temperature is read below 80F
  - Meeting Energy / Current expectations
    - Energy Profiler results
  - I2C driver is developed to be a generic and non-application specific
  - Application specific code located in app.h/.c
  - I2C driver is interrupt driven
    - Verified via the Energy Profiler and code review
  - Proper commenting of functions
  - 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