Lab 8 - PSoC 4 BLE IoT Sensor-Based System Design

School of Engineering

Electrical and Computer Engineering Department

ECEG 721-61 Embedded Systems

Jamie Quinn & Lucia Rosado-Fournier

December 9th, 2020

1.1 Objective

There are several objectives to this lab. The main objective is to measure a simulated heart rate and send the data over Bluetooth Low Energy (BLE). The next objective is to optimize the design for low power consumption. The last objective is to gain a better understanding of the usage of BLE with a secondary device.

1.2 Introduction

The heart rate is simulated by measuring an analog signal input. The signal is sent at a repeated interval, like a heart rate. The heart rate is measured by identifying these peaks in the signal that define the interval.

The BLE component of this lab will need to generate a notification every second to update the heart rate value to the client device. The BLE pioneer kit will act as a GATT server and send this signal over to the client.

Lastly, for the low power consumption portion of the lab, different modes will be implemented onto the device, with a watchdog timer (WDT) to act as the source of wakeup between the modes. The active mode handles the measurement of the signal and sending that information to the client device every second. The device will enter hibernate mode after being disconnected or timed out. It can then be woken up when the button SW2 is pressed.

1.3 Component Requirements

1.3.1 Hardware Components

1. BLE Pioneer Kit
2. USB Cable
3. 2 Jumper wires
4. Apple or Android device

1.3.2 Software Components

1. PSoc Creator version 3.1 or higher
2. CySmart 1.0
3. CySmart on the Apple or Android Device

1.4 Software

The code for this lab is broken into four main .c files. Starting with BleProcessing.c, this file contains the code to implement the BLE capability for the PSoC 4 boards. Next is HeartRateProcessing.c, where the heart rate measurement is implemented. Afterwards is the WatchdogTimer.c file that defines the functionality of the watchdog timer. Lastly, is the main.c file, the top level file of this lab.

The main.c file first initializes the system, calling upon the functions created in the previously mentioned files. It first turns off all LEDs and then enables interrupts. Afterwards the different components like the ADC, opamp, BLE, Heart Rate Service, and the WDT. Afterwards, it enters a forever loop where the heart rate is scanned and a notification is sent every second to the connected device. It then enters deep sleep mode so it can wait for the wakeup interrupt from the WDT. It also checks if the device is disconnected to enter hibernate mode and then waits for a button press of SW2.

Code

The code is broken into four different files and each is quite long. For brevity purposes, only the main function is below, but the rest of the code can be found on the GitHub repository.

```
int main()
{
       static uint32 previousTimestamp = 0;
       static uint32 currentTimestamp = 0;
       CYBLE_LP_MODE_T bleMode;
       uint8 interruptStatus;

       /* Initialize all blocks of the system */
    InitializeSystem();

       /* Run forever */
       for(;;)
       {
       /* Wake up ADC from low power mode */
       ADC_Wakeup();

       /* Analog Front End.
       * Detects the input signal and measures Heart Rate
       */
       ProcessHeartRateSignal();

       /* Put ADC in low power mode */
       ADC_Sleep();
```

```
/* Measure the current system timestamp from watchdog timer */
currentTimestamp = WatchdogTimer_GetTimestamp();

/* Send Heart Rate notification over BLE every second.
* Check if the current timestamp minus previous exceeds 1000 ms.
*/
if((currentTimestamp - previousTimestamp) >= 1000)
{
      /* Call API defined in BleProcessing.c to send
      * notification over BLE.
      */
      SendHeartRateOverBLE();

      /* Update the previous timestamp with the current timestamp. */
      previousTimestamp = currentTimestamp;
}

/* Try to stay in low power mode until the next watchdog interrupt */
while(WatchdogTimer_GetTimestamp() == currentTimestamp)
{
      /* Process any pending BLE events */
      CyBle_ProcessEvents();

      /* The idea of low power operation is to first request the BLE
      * block go to Deep Sleep, and then check whether it actually
      * entered Deep Sleep. This is important because the BLE block
      * runs asynchronous to the rest of the application and thus
      * could be busy/idle independent of the application state.
      *
      * Once the BLE block is in Deep Sleep, only then the system
      * can enter Deep Sleep. This is important to maintain the BLE
      * connection alive while being in Deep Sleep.
      */


      /* Request the BLE block to enter Deep Sleep */
      bleMode = CyBle_EnterLPM(CYBLE_BLESS_DEEPSLEEP);


      /* Check if the BLE block entered Deep Sleep and if so, then the
      * system can enter Deep Sleep. This is done inside a Critical
      * Section (where global interrupts are disabled) to avoid a
      * race condition between application main (that wants to go to
      * Deep Sleep) and other interrupts (which keep the device from
      * going to Deep Sleep).
      */
      interruptStatus = CyEnterCriticalSection();

      /* Check if the BLE block entered Deep Sleep */
      if(CYBLE_BLESS_DEEPSLEEP == bleMode)
      {
      /* Check the current state of BLE - System can enter Deep Sleep
      * only when the BLE block is starting the ECO (during
```

```
        * pre-processing for a new connection event) or when it is
        * idle.
        */
        if((CyBle_GetBleSsState() == CYBLE_BLESS_STATE_ECO_ON) ||
               (CyBle_GetBleSsState() == CYBLE_BLESS_STATE_DEEPSLEEP))
        {
               CySysPmDeepSleep();
        }
        }
        /* The else condition signifies that the BLE block cannot enter
        * Deep Sleep and is in Active mode.
        */
        else
        {
        /* At this point, the CPU can enter Sleep, but Deep Sleep is not
        * allowed.
        * There is one exception - at a connection event, when the BLE
        * Rx/Tx has just finished, and the post processing for the
        * connection event is ongoing, the CPU cannot go to sleep.
        * The CPU should wait in Active mode until the post processing
        * is complete while continuously polling the BLE low power
        * entry. As soon as post processing is complete, the BLE block
        * would enter Deep Sleep (because of the polling) and the
        * system Deep Sleep would then be entered. Deep Sleep is the
        * preferred low power mode since it takes much lesser current.
        */
        if(CyBle_GetBleSsState() != CYBLE_BLESS_STATE_EVENT_CLOSE)
        {
               CySysPmSleep();
        }
        }

        /* Exit Critical section - Global interrupts are enabled again */
        CyExitCriticalSection(interruptStatus);
}

/* Hibernate entry point - Hibernate is entered upon a BLE disconnect
* event or advertisement timeout. Wakeup happens via SW2 switch press,
* upon which the execution starts from the first line of code.
* The I/O state, RAM and UDBs are retained during Hibernate.
*/
if(enterHibernateFlag)
{
        /* Stop the BLE component */
        CyBle_Stop();

        /* Enable the Hibernate wakeup functionality */
        SW2_Switch_ClearInterrupt();
        Wakeup_ISR_Start();

        #if (RGB_LED_IN_PROJECT)
        /* Turn off Green and Blue LEDs to indicate Hibernate */
        Led_Advertising_Green_Write(1);
        Led_Connected_Blue_Write(1);
```

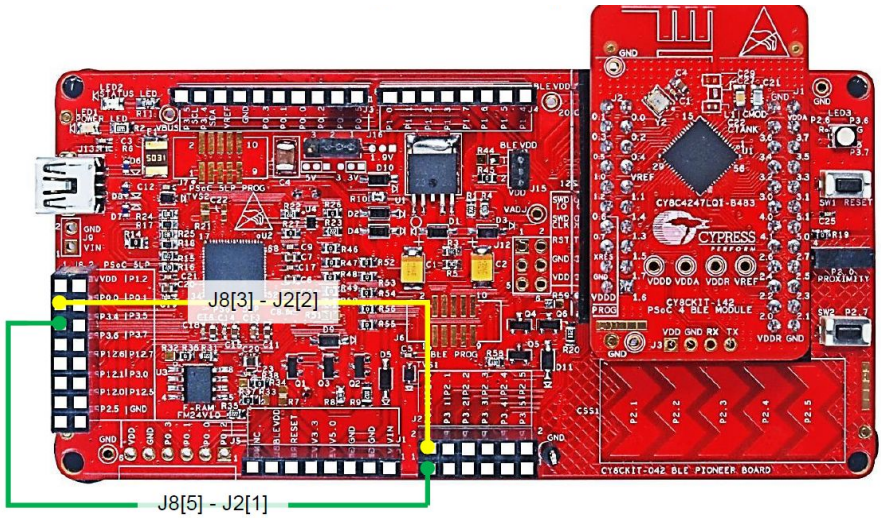
```
                /* Change the GPIO state to High-Z */
                Led_Advertising_Green_SetDriveMode(Led_Advertising_Green_DM_ALG_HIZ);
                Led_Connected_Blue_SetDriveMode(Led_Connected_Blue_DM_ALG_HIZ);
                #endif  /* #if (RGB_LED_IN_PROJECT) */

                /* Enter hibernate mode */
                CySysPmHibernate();
        }
        }
}
```

1.5 Procedure

All software settings were configured with the code given, but for instructions on how to configure, the full documentation is on the GitHub repository.

1. If needed, change the device to match the board.
2. If needed, update the components by clicking Project -> Update Components and then going through the window that pops up.
3. Build the program to generate the API needed for the components.
4. While unplugged from the PC, connect the two jumper cables as shown below. Connect pin P0.0 to P2.0 to simulate the heart rate. Connect P3.4 to P3.0 check the different values.



5. Turn on bluetooth and open the CySmart application on the Apple or Android Device.
6. Under Nearby Devices, connect to the BLE Lab 3 device, which is the board.
7. Once connected, tab the Heart Rate service and observe.

8. To observe the number changing, press SW2.
9. Disconnect the device to enter hibernate mode and press SW2 to wake it up.

1.6 Observation

As expected, when first programming the board, the LED turned green to indicate that the device is currently advertising. After connecting to the board through the Android device, the light changed to Blue to show that it was connected. These changes in light can be seen in the images below. Also shown below is the CySmart Android app showing the device selection page and the service selection page.

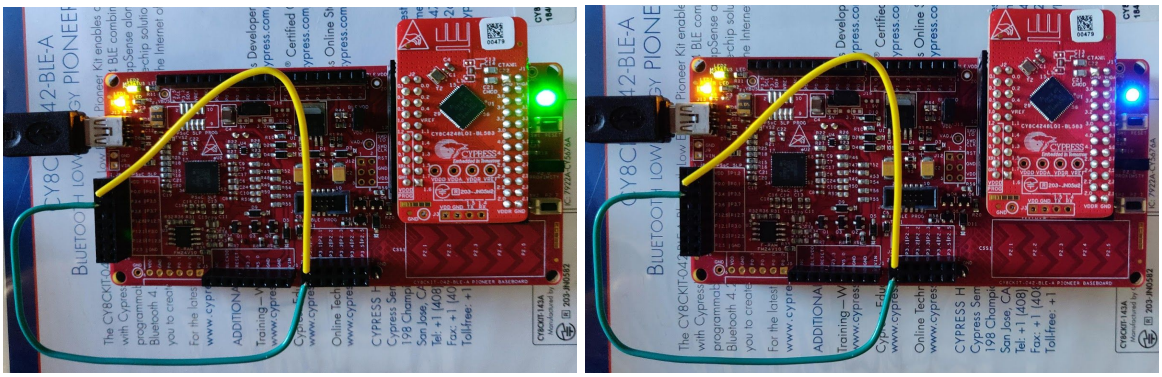


Figure *- Green LED when Advertising and Blue LED when Connected.*

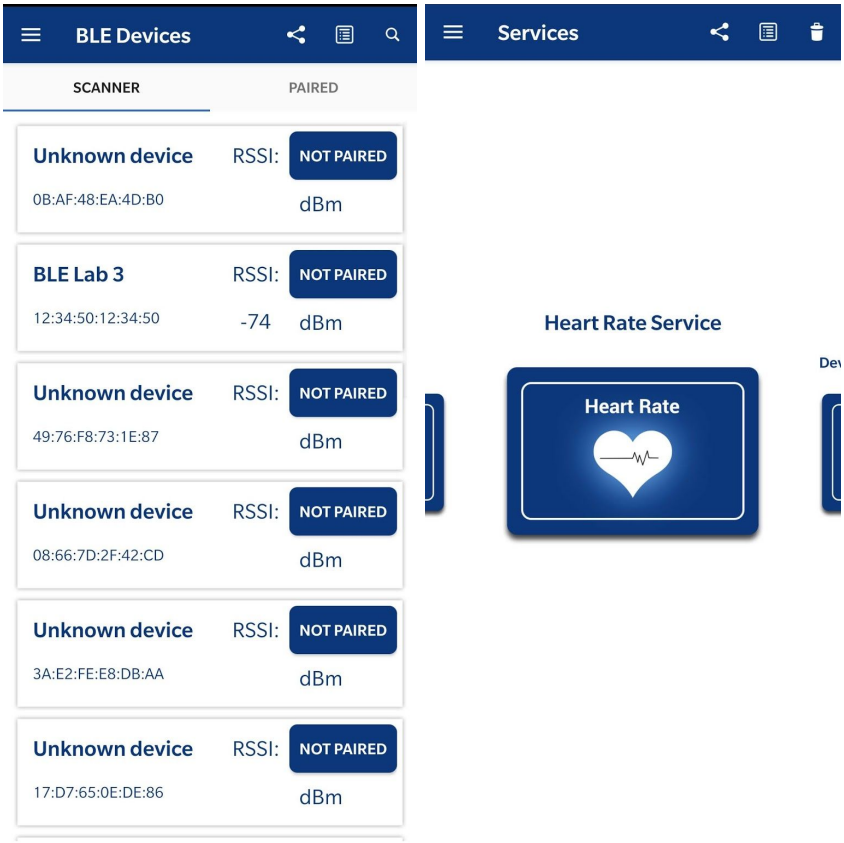


Figure *- Device and Heart Rate Service Selector.*

The heart rate that was being transmitted was between 60 bpm and 115 bpm which was also expected. Below is the heart rate when initially connected, and the heart rate after pressing SW2 to change it.

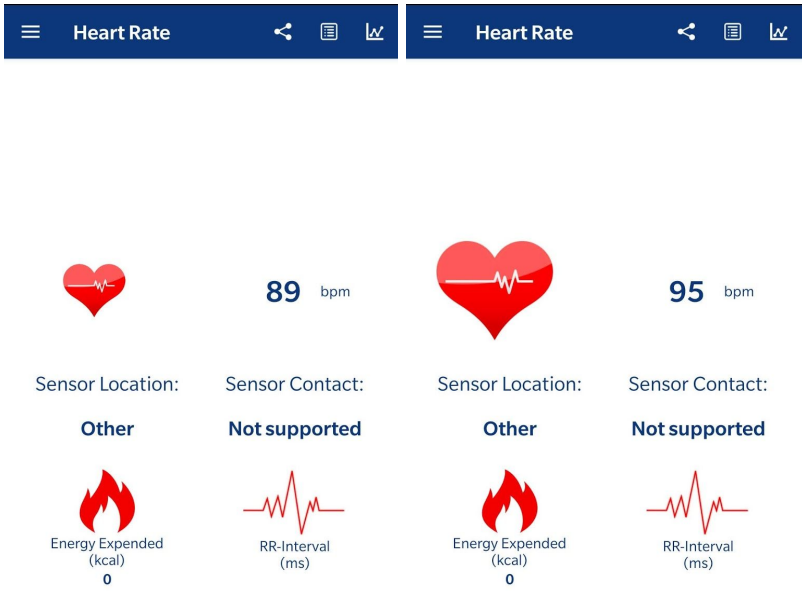


***Figure** - Heart Rate when Connected (left) and Heart Rate after pressing SW2 (right).*

1.7 Summary

The lab ran as expected and was generally easy to implement because the settings were set up beforehand. We attempted to run through the settings ourselves to see what the process was like, but ran into some issues along the way. After restarting with the initial file, we were able to get it to run as expected. Overall, the lab and report took about two hours to complete.