An ECG Monitor with Bluetooth Low Energy

by

Lan Xu
Master of Engineering, Southeast University, China, 2006

A Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

Supervisory Committee

An ECG Monitor with Bluetooth Low Energy

by

Lan Xu
Master of Engineering, Southeast University, China, 2006

Supervisory Committee

Dr. Xiaodai Dong, Department of Electrical and Computer Engineering
Supervisor

Dr. Lin Cai, Department of Electrical and Computer Engineering
Departmental Member

Abstract

Though there have been a number of instruments for heart condition monitoring, few of them is suitable for individual portable use, which makes the expenditure of related healthcare extremely high with the increasing number of chronic patients. To meet the challenge, we develop a wireless personal electrocardiogram (ECG) monitoring system which is meant to be a portable, low cost, and easy operational device for use at home. The ECG sensor in the system is designed for recording electrocardiogram signal and transmitting it to a smartphone through the emerging short-range communication technology, namely Bluetooth Low Energy (BLE). An application is accordingly developed for smartphone terminals to receive the ECG signal via BLE, display the signal wave and transmit the data over the Internet to a web server. This project focuses on software development for the ECG sensor system. It realizes the ECG signal transmission using BLE and the ECG data transmission between the sampler and the BLE module using Serial Peripheral Interface (SPI). Furthermore, a Universal Boot Loader (UBL) is also featured for easy firmware updates by customers in future use.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank:

Dr. Xiaodai Dong for trusting me and giving me this great opportunity of being involved in this attractive and awesome project.

Ping Cheng, Leyuan Pan, Weiheng Ni for their support during the whole project.

Lan Xu

Chapter 1 Introduction

1.1. Background and System Architecture

The wireless electrocardiogram (ECG) system is designed to record and transmit a patient's electrocardiogram information wirelessly in real time to a cloud server through a smartphone. The entire system consists of three major parts: 1) A small and portable ECG sensor, wearable by a patient, which is able to record ECG data and transmit the data to a smartphone using Bluetooth Low Energy (BLE); 2) An Android application (could also be an iOS application if applicable), which is used to receive the ECG data via BLE, compute the heartbeat rate, store and transmit it over the Internet to a cloud server; 3) Web service capable of storing the ECG data and the profile information for both patients and doctors. The ECG system brings convenience to patients to monitor their heart conditions, facilitates hospital staff to access the patients' information, and makes it possible for doctors to *monitor* and *diagnose* the heart disease timely. The main purpose of this project is to implement the firmware of the ECG sensor, and standardize the device to a customer-based product. The functions of each major components are summarized as follows:

The ECG sensor is developed to monitor the ECG condition of the customer in daily life. It is expected to overcome the inconvenience of conventional ECG monitoring systems so as to facilitate medical monitoring for both patients and doctors.

Before presenting the works of this project, we first illustrate in Figure 1 the basic system architecture of an ECG system. The ECG sensor generates a packet after sensing the ECG data directly from the human body, and then it forwards the data packet to a smartphone using BLE. The hardware of the ECG sensor was mainly designed and implemented by Zander Erasmus, a former student in our group who made a great initial contribution to the ECG project. Firmware development is based on the project template provided by Texas Instruments (TI) [1], using C code. A reference to

the entire BLE protocol stack implementation can be downloaded from the TI official website [2].

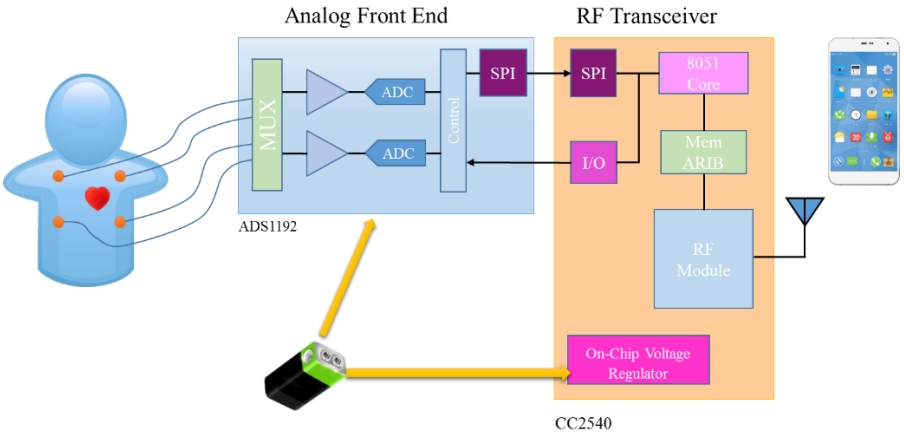


Figure 1: Architecture of the ECG sensor system

1.2. Hardware Components of ECG Sensor

ECG works by detecting and amplifying tiny potential changes in the skin that are caused when the electrical signal in heart muscle is charged and spread during each heartbeat. The following main components are needed in the ECG sensor system:

- ECG electrodes: A kind of transducer is needed to record the ECG data. It converts the ionic potentials generated within the body into electronic potentials which can be measured by conventional electronic instrumentation. In our system, the transducers are disposable and inexpensive electrodes, specifically the 3M Red Dot electrodes in our project, which work well as monitors.

- Battery: The battery that is used to power the ECG circuit is a single lithium polymer battery. It is rechargeable by the ECG sensor circuit which will be described later.

- Sensor circuit: The circuit in Figure 2 was designed and built by Zander Erasmus with Dr. Xiaodai Dong. One major purpose of this project is to verify the functionality of the circuit board, including writing and debugging the firmware code running on the microcontroller as well as troubleshooting any errors encountered.

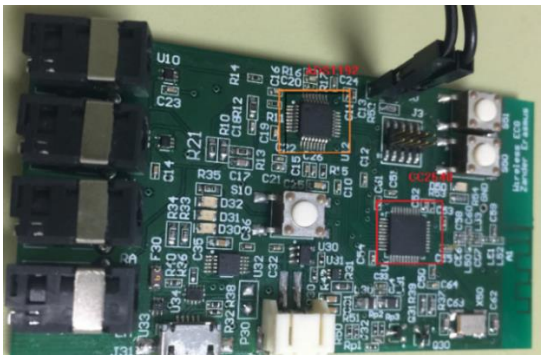


Figure 2: The sensor circuit

The basic building blocks of the sensor circuit are depicted in Figure 3. We have overall four electrodes for recording the signals coming from a human body. Left-Arm (LA) electrode and Right-Arm (RA) electrode are attached to the left arm and the right arm, respectively. Left-Leg (LL) electrode and Right-Leg (RL) electrode are attached to the left leg and the right leg, respectively. The analog-front-end (AFE) block (chip ADS1192) is the first block that the source signals go through. The ADS1192 is a highly integrated low-power AFE that features two high-resolution ECG channels. Because of its excellent features, the ADS1192 chip has been widely used in medical instrumentation including patient monitoring and sports and fitness. The CC2540 is a cost-effective, low-power, and true System-on-Chip (SoC) solution for BLE applications. BLE enables seamless connectivity to a wireless terminal that allows the end user to remotely collect and monitor the ECG data.

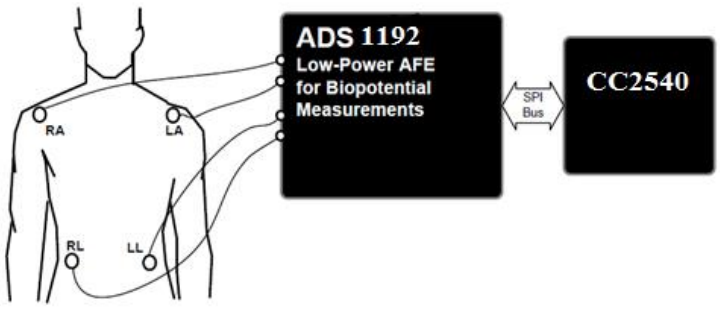


Figure 3: Basic building blocks of an ECG sensor circuit

1.3. Features of the Project

According to the main purpose of the project, it provides a solid and high-level overview of device communications using BLE in the ECG system. More specifically, the following aspects are featured works and contributions of this project.

1. Dig into the BLE protocol stack and grasp how and why BLE operates.
2. Learn how BLE device discovers each other and how to establish connections.
3. Set up tools and infrastructure for BLE application development.
4. Configure and control the chip ADS1192.
5. Receive the ECG data from ADS1192 continuously.
6. Boot loader implementation.
7. Implement USB 2.0 compliance to meet consumer product standards.

1.4. Outline of the Report

The outline of this report is organized as follows:

Chapter 2 gives a brief overview of the BLE protocol as well as some details for each layer of the BLE protocol. Details of the development and debugging tools are also presented.

Chapter 3 establishes the ECG data transmission between ADS1192 and CC2540 through SPI.

Chapter 4 develops the bootloader for efficient and easy firmware upgrade.

Chapter 5 concludes the report and suggests future work.

Chapter 2 BLE Stack and Functionality Tests

As described above, a major task of this project is to establish a wireless communication link from the ECG sensor circuit to a mobile terminal using BLE supported by CC2540 on the circuit. This chapter first provides a high-level overview of BLE, including introducing and explaining elementary fundamental concepts of BLE. Then, each of the protocol layers and their essential features are detailed with elaborations. According to the BLE protocol, we developed a software which successfully establishes the BLE link and accomplishes wireless data transmission from the ECG sensor circuit to the mobile terminal. Moreover, a procedure of the BLE functionality verification on the developed software is exemplified with debugging instructions.

2.1. BLE General Description

BLE is a wireless personal area network technology designed and marketed by the Bluetooth Special Interest Group (SIG) for short-range communications [3]. The advent of BLE has occurred while other low-power wireless solutions, such as ZigBee, WiFi, has been already gaining a certain momentum in the application domain. In the following few subsections, a brief introduction of BLE protocols is firstly presented, which are based on the popular handbook of BLE [4]. The following parts help the readers quickly gain some basic aspects of the BLE and are useful for better understanding the works of this project.

2.1.1. Specification and Configuration

The Bluetooth Specification (4.0 and above) includes two types of wireless technologies: classic Bluetooth (BR/EDR) evolved with the Bluetooth Specification since 1.0 and BLE newly introduced in the version 4.0 of the specification. Correspondingly, there are two types of devices that can be used with these technologies: Single-mode (BLE, Bluetooth Smart) device, which only implements BLE, can communicate with single-mode and dual mode devices but not classic

devices; Dual-mode (BR/EDR/LE, Bluetooth Smart Ready) device, which implements both BR/EDR and BLE, can communicate with any Bluetooth devices.

Figure 4 [4] shows the configuration possibilities between available Bluetooth versions and device types.

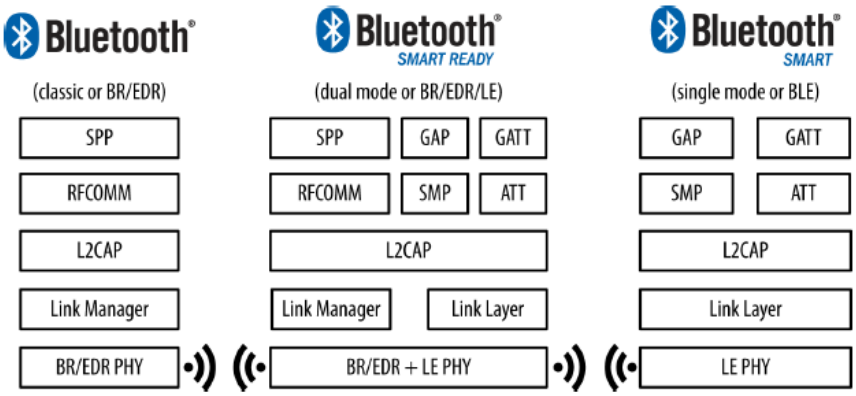


Figure 4: Configurations between Bluetooth versions and device types

Nowadays BR/EDR has already got a large market share. The trend is that the single-mode BLE devices will be more universal. The dual mode devices can forward the data received from a signal mode BLE device to the Internet through WIFI or cellular data. This feature leads to the increasing market share of the BLE device.

Our ECG system has these two mode devices: an ECG sensor which is a BLE device and a smartphone which is a dual mode device.

2.1.2. Chip Based Description

The whole BLE protocol stack of every BLE device can be split into three main parts: controller, host and application. The controller includes lower layers of the BLE protocol stack. It transmits and receives radio signals and knows how to interpret the signals. The host contains upper layers of the BLE protocol stack. It manages how two and more BLE devices communicate with each other and how services can be provided to user application over the radio. The user application interacts with the BLE protocol stack and enables a particular use case. Additionally, a special commu-

nication protocol, the Host Controller Interface (HCI) is provided by the BLE specification to connect the controller and the host.

All these layers can be implemented in a single chip or in different devices which can communicate with each other using UART, USB, SPI, etc. Three most common configurations today are shown in Figure 5 [4]. In order to reduce the cost and complexity of the circuit board, most BLE devices tend to use SoC configurations such as our ECG sensor.

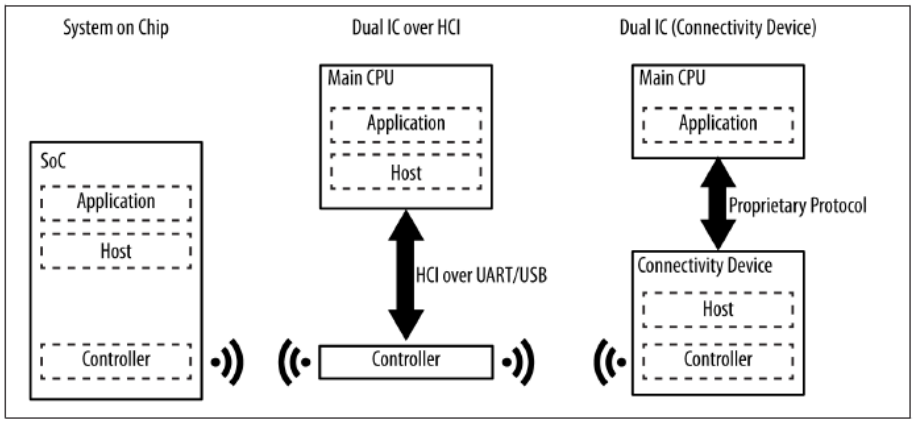


Figure 5: Hardware configuration

2.2. BLE Protocol Basics

As in the above descriptions and highlighted in Figure 6 [7], a complete single-mode BLE device usually has three parts: controller, host and application. Each of these basic modules can be split into several layers that provide the required functionalities. In the following, we elaborate the detailed responsibilities and layers of the three parts.

Application

The application, like in all other types of systems, belongs to the highest layer which is responsible for the logicality, user interface, and data processing of everything related to the actual use case that the application implements. The architecture of an application is highly dependent on each particular use.

Host

Includes the following layers:

- Generic Access Profile (GAP)
- Generic Attribute Profile (GATT)
- Logical Link Control and Adaptation Protocol (L2CAP)
- Attribute Protocol (ATT)
- Security Manager (SM)
- Host Controller Interface (HCI), Host side

Controller

Includes the following layers:

- Host Controller Interface (HCI), Controller side
- Link Layer (LL)
- Physical Layer (PHY)

In the following order of subsections, we will describe the different parts that make up a BLE device from bottom to top.

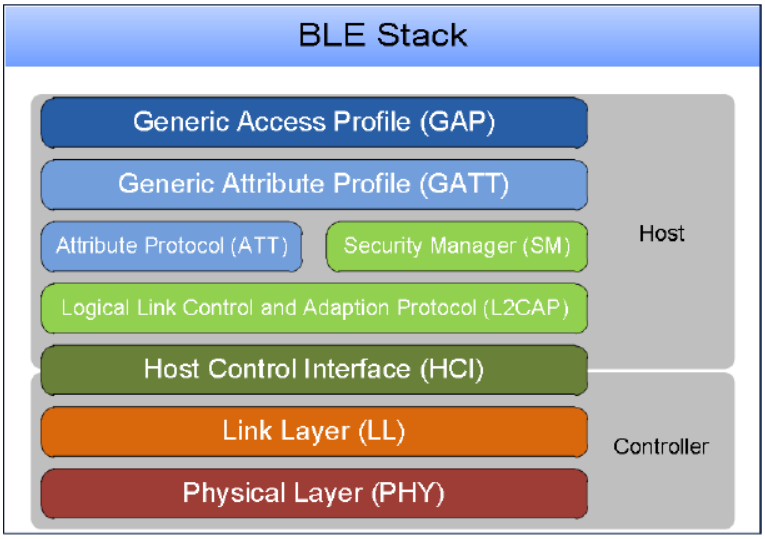


Figure 6: BLE protocol stack

2.2.1. Physical Layer

BLE uses the 2.4 GHz Industrial, Scientific, and Medical (ISM) band for wireless communication and defines 40 Radio Frequency (RF) channels with 2 MHz channel spacing. There are two types of BLE RF channels: advertising channels and data channels. As shown in Figure 7, channels 37, 38, and 39, are labeled as advertising channels for connection establishment and broadcasting, and the remaining 37 data channels are used for bi-directional communication between connected devices.

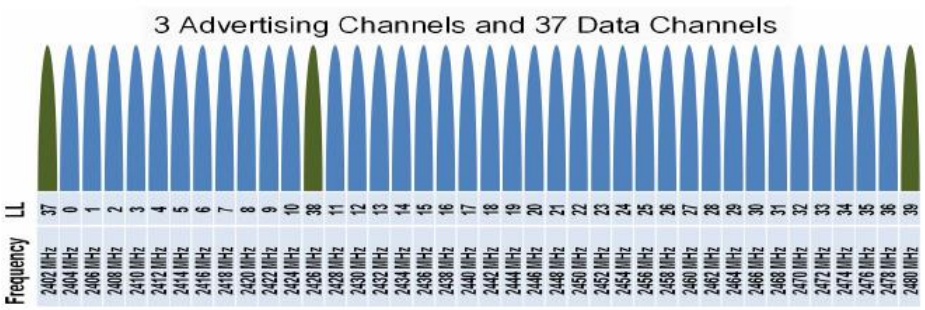


Figure 7: Frequency channels

According to the standard, the data transmission uses the adaptive frequency hopping, which is one of the most well-known spread spectrum technologies. With frequency hopping, the radio hops between channels on each connection event are defined by the formula: channel = (curr_channel + hop) mod 37.

All physical channels use a Gaussian Frequency Shift Keying (GFSK) modulation. The modulation rate for BLE is fixed at 1 Mbps, which is the upper physical throughput limit of the technology.

2.2.2. Link Layer

The link layer is the part that directly interfaces with PHY. It controls the RF state of the device. There are six possible link layer states of a BLE device.

- Standby: device is not transmitting or receiving any data, and is not connected to any other device
- Advertiser - periodically broadcasting advertisements
- Scanner - actively looking for advertisers

- Initiator - actively trying to initiate a connection with another device
- Master - connected to another device as a master
- Slave - connected to another device as a slave

The typical state chart of devices, taking the ECG sensor and a mobile terminal, for instance, are presented in Figure 8.

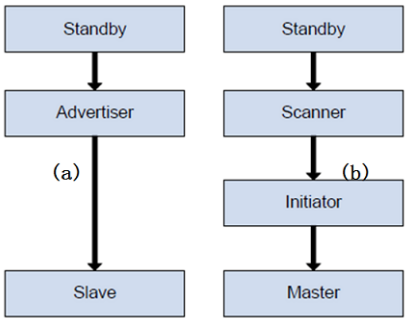


Figure 8: Link layer state flow chart

Any device that transmits advertising packets is called an advertiser. Adverting packets are transmitted through the advertising channels. Each advertising packet can carry up to 31 bytes of advertising data payload, along with the basic header information including Bluetooth device address. The transmission of packets through the advertising channels takes place in intervals of time called advertising events which range from 20ms to 10.24s. Devices that only aim at receiving data through the advertising channels are called scanners. The connection between two devices is created by the initiator sending the Connection Request message to the advertiser, and then a point-to-point connection between the two devices is created.

BLE defines two device roles at the Link Layer for a created connection: the master and the slave. These are the devices that act as initiator and advertiser during the connection creation, respectively. Once a connection between a master and a slave is created, the physical channel is divided into non-overlapping time units called connection events. Every connection event starts with the transmission of a packet from the master. If the slave receives a packet, the slave must send a packet to the master in

response. However, the master is not required to send a packet upon receipt of a packet from the slave.

A connection can be voluntarily terminated by either the master or the slave at any time. If one side initiates termination, and the other side must respond accordingly before both devices exit the connected state.

2.2.3. Host Controller Interface (HCI)

The HCI is a standard protocol that enables the communication between a host and a controller to take place across a serial interface. As mentioned earlier in this chapter, the controller is the only module that contacts with the PHY layer with hard real-time requirements. Therefore, it is often practical to separate it from the host, which implements a more complex but less timing-stringent protocol stack. This layer can be implemented either through a software API or by a hardware interface such as UART, SPI or USB.

2.2.4. Logical Link Control and Adaption Protocol (L2CAP)

The main goal of L2CAP is to multiplex the data of the upper three protocols and encapsulate them into the standard BLE packet format, and vice versa. It also performs fragmentation and recombination, which is a process to take large packets from the upper layers and break them up into the 27-byte maximum payload size of the BLE packets on the transmitter side. On the reception path, it receives multiple packets that have been fragmented and then recombines them into a single large packet. It is important to note that, whenever only default packet sizes are used, the L2CAP packet header takes up four bytes, which implies that the effective user payload length is 23 bytes.

2.2.5. Attribute Protocol (ATT)

The ATT defines the communication between two devices playing the roles of server and client. A client requests data from a server, and a server sends data to cli-

ents. Each server maintains a set of attributes, each of which is assigned a 16-bit attribute handle, a universally unique identifier (UUID), a set of permissions, and a value.

The client can access the server's attributes by sending requests to the server. For great efficiency, a service provides two types of unsolicited messages that responses to a client: 1) notification, which are unconfirmed; (ii) indications, which require the client to send a confirmation. A client can also send commands to the server to write the attribute values [8].

2.2.6. Generic Attribute Profile (GATT)

The GATT defines a service framework using the ATT protocol layer. It maintains the same client/server architecture presenting in ATT, but the data is now encapsulated in services which consist of one or more characteristics, while each characteristic is a set of data which includes a value and properties. The data related to services and characteristics is stored in attributes. The architecture of GATT is shown in Figure 9 [7]:

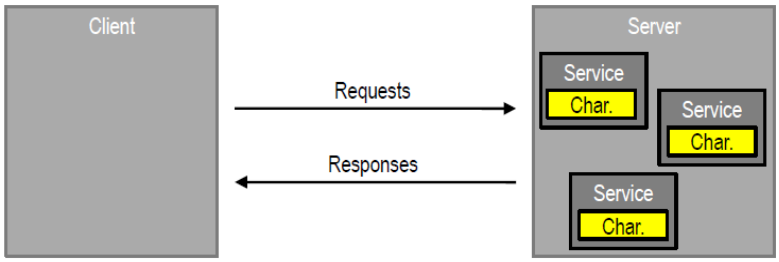


Figure 9: GATT architecture

GATT client: This is the device that wants data. It initiates commands and requests towards the GATT server. It receives responses, indications, and notifications data from the GATT server.

GATT server: This is the device that contains the data and accepts incoming commands and requests from the GATT client and sends responses, indications, and notifications to a GATT client accordingly.

2.2.7. Generic Access Profile (GAP)

The GAP lies at the highest level of the core BLE stack, it specifies device roles, modes and procedures for the discovery of devices and services, and as well as the management of connection establishment and security [8].

The BLE GAP defines the following four roles with specific requirements when operating over the physical channel.

- Broadcaster role: A device operating in the broadcaster role only sends advertising data through the advertising channels and does not support connections with other devices.
- Observer role: A device operating in the observer role is a device that receives advertising data. Its only purpose is to receive the data transmitted by the broadcaster.
- Peripheral role: A device that accepts the establishment of a physical connection. A device operating in the peripheral role will be in the slave role in the link layer connection state.
- Central role: A device that supports the central role initiates the establishment of a physical connection. A device operating in the central role will be in the master role in the link layer connection state.

In consequence, the peripheral and central roles require that the device's controller support the master and slave roles, respectively.

2.3. BLE Software Development

2.3.1. Software Overview

The TI free BLE software development kit is an efficient and valid software platform for developing single-mode BLE applications. It is based on the CC2540/41 with complete SoC solutions. The latest version of the BLE software development kit that can be downloaded from the TI official website is BLE-CC254x-1.4.0. The BLE

software development kit includes several projects implementing a variety of profiles and demonstration applications [9], [10].

Software developed using the BLE software development kit consists of the following five major components:

- Operating System Abstraction Layer (OSAL)
- Hardware Abstraction Layer (HAL)
- The BLE Protocol Stack
- Profiles
- Application

The BLE protocol stack is provided as object code, while the OSAL and HAL components are provided in source code form. Also, three GAP profiles (peripheral role, central role, and peripheral bond manager) are provided as well as several sample GATT profiles and applications.

1. OSAL

The BLE protocol stack, the profiles, and all applications are all built around the OSAL. The OSAL is not an actual operating system (OS) in the traditional sense, but rather a control loop that allows software to setup the execution of events. For each layer of software that requires this type of control, a task identifier (ID) and a task initialization routine must be defined and added to the OSAL initialization [9], and an event processing routine must be defined as well. In addition to task management, the OSAL provides additional services such as message passing, memory management, and timers. All OSAL code is provided as full source code. The following is the hierarchy of the ECG project.

```
// The order in this table must be identical to the task initialization
calls below in osalInitTask.
const pTaskEventHandlerFn tasksArr[] =
{
 LL_ProcessEvent,                                      // task 0
```

```
 Hal_ProcessEvent,                                    // task 1
 HCI_ProcessEvent,                                    // task 2
 OSAL_CBTIMER_PROCESS_EVENT(osal_CbTimerProcessEvent),// task 3
 L2CAP_ProcessEvent,                                 // task 4
 GAP_ProcessEvent,                                   // task 5
 GATT_ProcessEvent,                                  // task 6
 SM_ProcessEvent,                                    // task 7
 GAPRole_ProcessEvent,                               // task 8
 GAPBondMgr_ProcessEvent,                            // task 9
 GATTServApp_ProcessEvent,                           // task 10
 ECG_ProcessEvent                                    // task 11
};
```

2. HAL

The HAL of the CC2540 software provides an interface of abstraction between the physical hardware and protocol stack which allows for the development of new hardware without making changes to the protocol stack or application source code. The HAL includes software for the SPI and UART communication interfaces, ADC, keys, and LEDs, etc.

3. BLE protocol stack

The entire BLE protocol stack is provided as object code in a single library file. TI does not provide the protocol stack source code.

4. Profiles

The BLE software development kit includes three GAP role profiles, one GAP security profile, and several sample GATT service profiles. For GAP role profile, our ECG project only uses GAP peripheral role profile which provides the means for the ECG sensor to advertise, connect with a central device and request a specific set of connection parameters from a master device.

5. Application

The application processes the task events after the initialization of the application.

2.4.1. ECG Project Setup

All of the BLE projects included in the development kit have a similar structure, however, the project named SimpleBLEPeripheral is the reference for developing our ECG project. Another student under Dr. Dong's supervision has developed an ECG firmware which has been verified good performance on the previous device. For convenience, the new ECG project is created based on that one which largely saved our development time. On the left side of the IAR window, the workspace section lists the files used by the project in Figure 10.

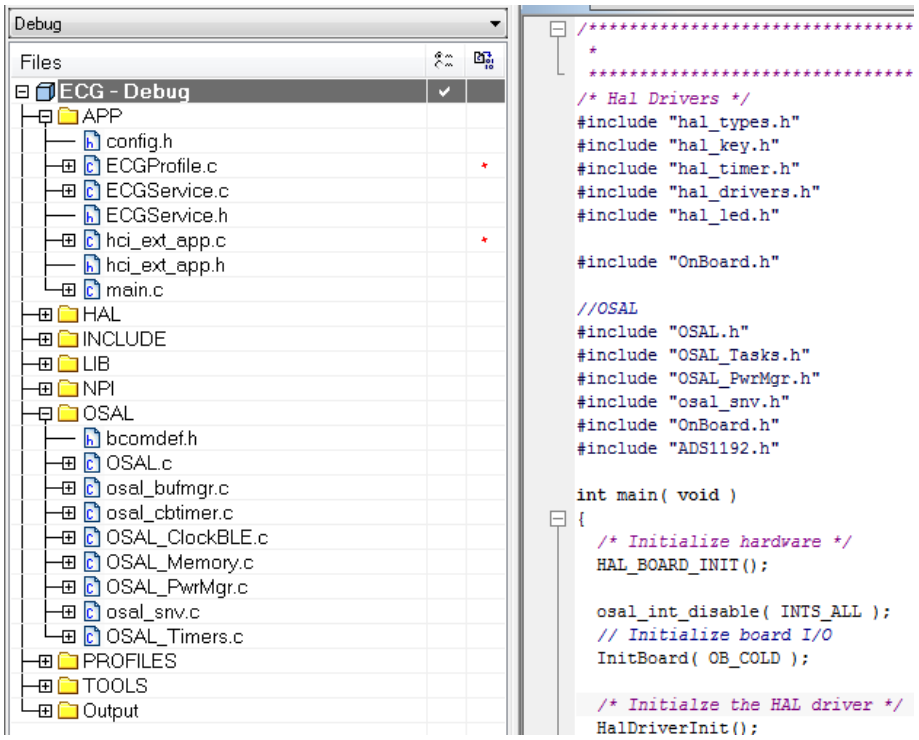


Figure 10: Project files

The file list is divided into the following groups:

• APP – This group contains the application source code and header files

• HAL – This group contains the HAL source code and header files.

• INCLUDE – This group includes all of the necessary header files for the BLE protocol stack API.

• LIB – This group contains the protocol stack library file CC2540_BLE_peri.lib.

• NPI – Network processor interface, a transport layer that allows you to route HCI data to a serial interface. The CC254X_BLE_HCI_TL_None.lib should be used when developing a single-chip application.
• OSAL – This group contains the OSAL source code and header files.
• PROFILES – This group contains the source code and header files for the GAP role profile, GAP security profile, and the sample GATT profile. In addition, this section contains the necessary header files for the GATT server application.
• TOOLS – This group contains the configuration files buildComponents.cfg and buildConfig.cfg. It also contains files OnBoard.c and OnBoard.h, which handle hardware interface functions.
• OUTPUT – This group contains files that are generated by IAR during the build process, including binaries and the map file.

The code for initialization and Bluetooth interface largely stay the same with the previous one, except for the hardware related code, which is used for initializing the hardware, mainly lies in the components of HAL and the file OnBoard.c.

2.4. Tools for Development and Debugging

The development tool that used in the project is the IAR Embedded Workbench for 8051 IDE. IAR Embedded Workbench incorporates a compiler, an assembler, a linker and a debugger into one integrated development environment. It can be used to track the source code step by step which is quick and useful during the whole development process.

Another useful tool is the CC2540EM-USB, depicted in Figure 11, which is a low-cost CC2540-based USB Dongle. It is used as a packet sniffer to sniff traffic over the air. Accompanying with the CC2540EM-USB, the SmartRF Packet Sniffer, exemplified in Figure 12, is a PC software which can display and analyze the data captured by the packet sniffer. This combination allows us to access all BLE data going out over the air at the lowest possible level.

Figure 11: USB Dongle

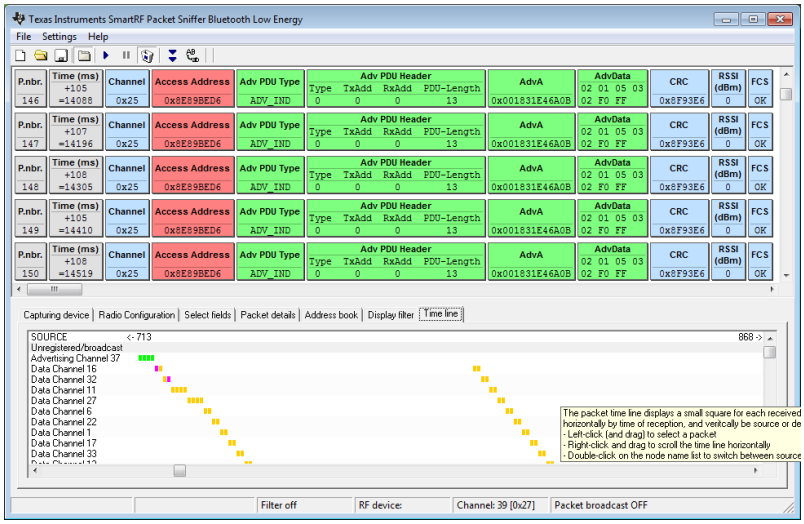


Figure 12: TI SmartRF sniffer application

2.5. Functionality Tests

In order to verify the functionality of the sensor circuit with the BLE, we did the following tests step by step at the beginning of the project. We used USB Dongle to capture the BLE data going over the air to completely understand the BLE protocol. Figure 13 shows the whole procedure of the BLE from advertising to connection termination. The test is conducted based on the following steps.

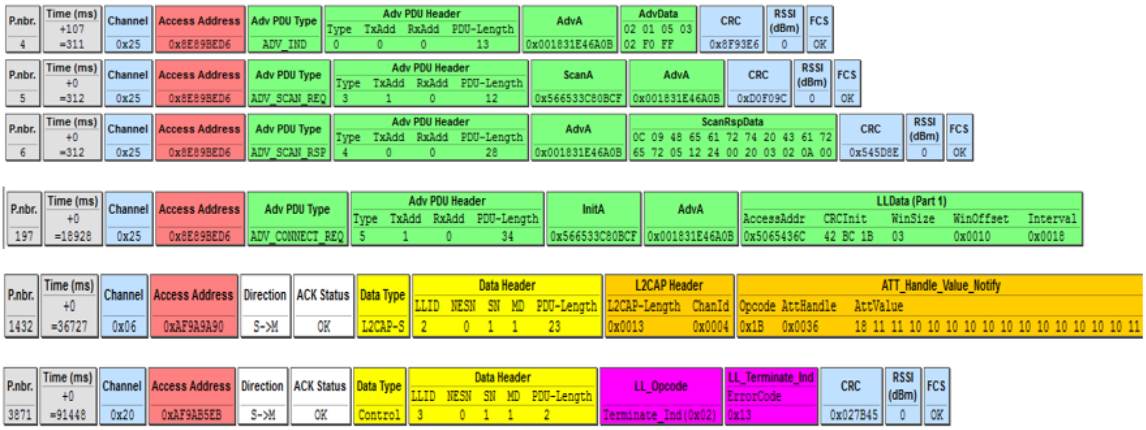


Figure 13: BLE packets

1) The sensor circuit is successfully powered up.

   Operation: Power on the circuit either by the CC Dongle or the lithium battery.

   Verification: A LED on the circuit board turns green.

2) The ECG sensor sends advertising packets after power on.

   Verification: Packet Sniffer captures advertising packet as showing in Figure 13 (P.nbr.4) with Adv PDU Type ADV_IND.

3) The ECG sensor can be scanned as a BLE peripheral by an Android application which is developed for data receiving and display on mobile terminals.

   Operation: Simply click the scan menu on the Android application.

   Verification:

   1. Packet Sniffer captures scan request packet (P.nbr.5) sending from the Android application with Adv PDU Type ADV_SCAN_REQ and scan response packet (P.nbr.6) replying from the ECG sensor with Adv PDU type ADV_SCAN_REQ. (Figure 13)

   2. The Android application successfully discovers the ECG sensor as depicted in Figure 14 with sensor name Heart Carer which is hard coded in the firmware.

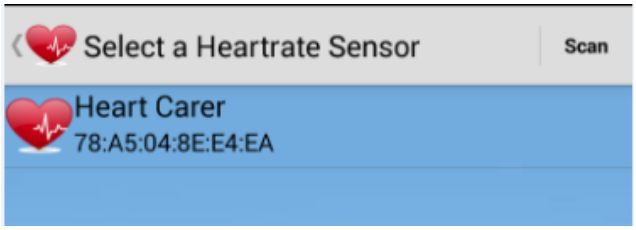


Figure 14: ECG sensor discovery by Android application

4) The BLE connection between the BLE peripheral (ECG sensor) and BLE central (Android smartphone) can be established successfully.

   Operation: Click the sensor name showed in the Android application to establish a BLE connection.

   Verification: Packet Sniffer captures connection request packet (P.nbr197) sending from the Android application with Adv PDU Type ADV_CONNECT_REQ (Figure 13).

5) The Android application receives ECG data successfully by displaying them on screen.

   Operation: At this test stage, a triangular signal is generated by hard code on the firmware. The data is generated periodically by setting a 4 ms timer to simulate a real ECG data.

   Verification:

   1. Packet Sniffer captures notification packets, e.g., P.nrb.1432 in Figure 13.
   2. Figure 15 displays the expected triangular signal received by the Android application.

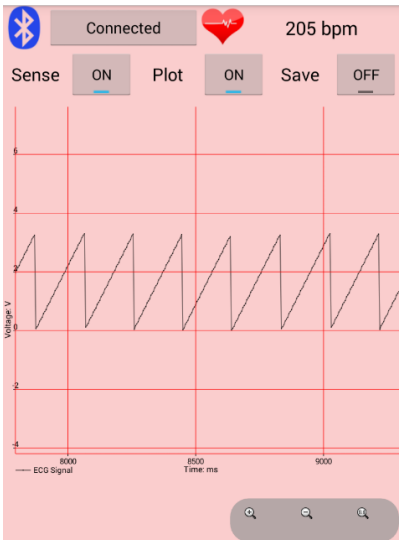


Figure 15: Triangular signal display

6) The BLE connection can be disconnected successfully.

Operation: Simply click the disconnect button.

Verification:

1. Packet Sniffer captures termination packet (e.g. P.nrb.3971 in Figure 13).
2. Plotting disappeared on the screen immediately.

The whole above process proves the functionality of the sensor circuit with BLE on both hardware and software.

Chapter 3 ECG Data Retrieval through SPI

On the ECG sensor circuit board, the project aims to establish a reliable data and control signal communication between the two essential components, i.e. the ADS1192 and the CC2540 as shown in Figure 3. This chapter describes the SPI communication between the ADS1192 and the CC2540. In order to establish the connection, the firmware is developed for the CC2540 to control the ADS1192 and fetch data from it. Flow charts of each component routine of the developed firmware are elaborated. Finally in this section, essential procedures for verifying and debugging the firmware functionalities are also provided.

3.1. Hardware Introduction

The ECG sensor circuit consists of two major components: 1) CC2540 acts as the microcontroller and BLE transceiver; 2) ADS1192, a two-channel, 16-bit, low-power, integrated AFE designed for portable ECG and respiration applications, contains all sensing elements. The ECG wire connectors are regarded as an additional interface component going into the AFE. The block diagram of the ECG sensor circuit is plotted in Figure 16.

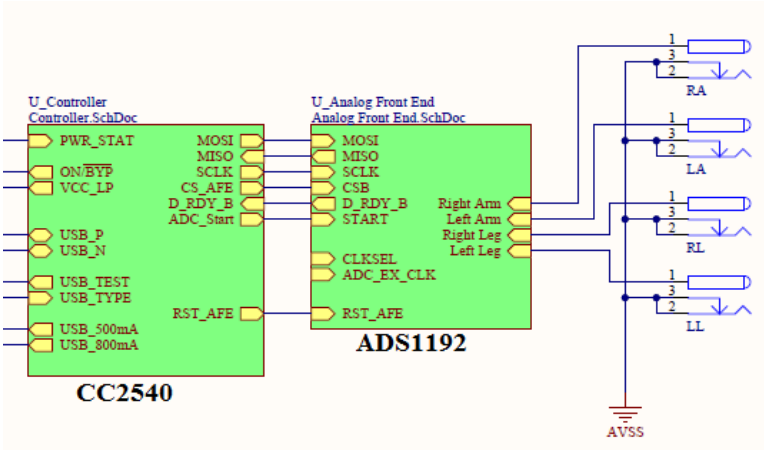


Figure 16: Block diagram of the ECG sensor circuit

According to the figure above, there are seven pins connected between CC2540 and ADS1192, namely MOSI, MISO, SCLK, CS_AFE, D_RDY_B, ADC_Start, and

RST_AFE. Because the firmware application is running on CC2540, the only way to access ADS1192 is to control the seven pin signals by CC2540.

Table 1: Signals related to AFE

Signal	Port	Port Position	Pin Number	Direction
DRDY	P0	6	13	Input
RST_AFE	P0	7	12	Output
ADC_Start	P1	0	11	Input
CS_AFE	P1	3	7	Output
SCLK	P1	5	5	Output
MOSI	P1	6	38	Output
MISO	P1	7	37	Input

3.2. Overview of ADS1192

The ADS1192 incorporates all features commonly required in portable, low-power medical, sports, and fitness ECG applications. The ADS1192 features high-resolution channels capable of operating up to 8 ksps. Each channel can be programmed independently for a specific sample rate. ADS1192 has a flexible input multiplexer per channel that can be independently connected to the internally-generated signals for test and temperature and lead-off detection [12].

Communication to the device is accomplished by using an SPI-compatible interface. The device provides two general-purpose I/O (GPIO) pins for general use. The internal oscillator generates a 512 kHz clock.

3.3. SPI Interface

The CC2540 communicates with the ADS1192 through the SPI interface. The SPI interface is a synchronous serial communication interface specification used for short distance communications [11]. SPI devices communicate in a full duplex mode using a master-slave architecture. In our system, it simply uses the single master mode and the single slave mode as shown in Figure 17. CC2540 works as the SPI Master while ADS1192 works as the SPI Slave.

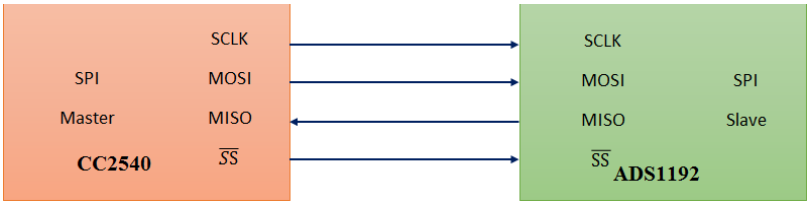


Figure 17: SPI single master and single slave mode

The SPI interface provided by the ADS1192 consists of four signals: $\overline{CS}$, SCLK, DIN, and DOUT. The CC2540 uses these interfaces to read and write registers, control the ADS1192 operation, and read conversion data.

- Chip Select ($\overline{CS}$)

  The $\overline{CS}$ selects the ADS1192 for SPI communications. $\overline{CS}$ remains low for an entire duration of the serial communication.

- Serial Clock (SCLK)

  SCLK is the SPI serial clock. It is used to shift in commands and shift out the data from the device. The SCLK features a Schmitt-triggered input and clocks data on the DIN and DOUT pins into and out of the ADS1192.

- Data Input (DIN)

  The DIN is used along with SCLK to communicate with the ADS1192. The device latches data on DIN on the falling edge of SCLK.

- Data Output (DOUT)

  The DOUT is used with SCLK to read conversion and register data from the ADS1192. Data on DOUT are shifted out on the rising edge of SCLK.

Except these four interfaces, there are three more signals related to ADS1192. The DRDY signal output from ADS1192 is used as a status signal to indicate that the conversion data are ready. The DRDY signal goes low when the new conversion data is available. Additionally, the START signal is used to let ADS1192 begin conversions when it goes high, while the RST_AFE is used to reset ADS1192 when it transitions low.

There are two methods can be used to retrieve data from DOUT. The commonly used method is to send the read data continuous command: Read Data Continuous, which set the ADS1192 device in a mode to read the data continuously without sending any other operation codes. Another one is to send read data command which can only read data once from the device. After DRDY signal goes low, the conversion data are read by shifting the data out on DOUT. The MSB of the data on DOUT outputs first. DRDY returns to high on the first SCLK falling edge. DIN remains low during the entire read operation. The number of bits in DOUT depends on the number of channels and the number of bits per channel. For the two-channel, 16-bit resolution ADS1192, the number of data output is 48 bits, including 16 status bits and two channels ×16 bits data. Figure 18 shows the data output protocol for ADS1192 [12].

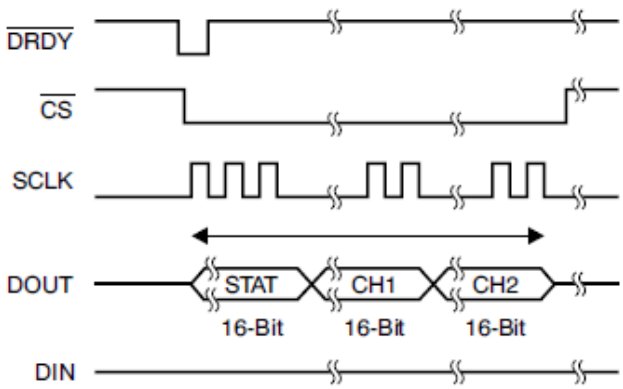


Figure 18: SPI bus data output for ADS1192

The format of the 16 status is 1100 + LOFF_STAT[4:0] + GPIO[1:0] + 5 zeros, the LOFF_STAT register is detailed in Appendix B. The status of these five bits can be used to determine whether the positive or negative electrode on each channel is on or off. The data format for each channel is two's complement and MSB first.

3.4. Operation Description

There are two main operations for the CC2540 to control the ADS1192 and acquire data.

- Setup

The CC2540 works in Master SPI Mode. Some necessary steps are needed to be set up before using the USART in Master SPI mode. The USART Mode Select bits (P1SEL[7:4]) must be set to one to select the Mater SPI mode. The SCLK is configured manually as an output to enable master clock generation.

- Byte transfer

A SPI byte transfer in master mode is initiated when the `UxDBUF` register is written. The USART generates the SCLK serial clock using the baud-rate generator and shifts the provided byte from the transmit register onto the MOSI output. At the same time, the receive register shifts in the received byte from the MISO input pin.

When using the receiver, wait for the `UxCSR.TX_BYTE` flag and then read the received byte from `UxDBUF`. Note that it is the master device's responsibility to give the slave device enough time to prepare its next byte before starting a new transfer.

3.5. Implementation

During power on, the firmware (Run-Code in CC2540) first configures the USART in Master SPI mode and then sets the ADS1192 registers with proper values. Appendix B summarizes the key register functions and corresponding configuration values. The firmware will instruct the ADS1192 to start the data conversion in continuous mode at 250 SPS. According to the sampling rate configured, the ADS1192 then generates DRDY which indicates that the conversation data is ready. Based on the interrupt, the firmware reads the data from the ADS1192. The data is stored in a buffer and sent to the Android application in individual BLE-notification packet afterward. Each notification packet consists of 20 bytes containing the following data format as defined in Table 2.

Table 2: Notification packet format

Sample 1					Sample 2					Sample 3					Sample 4				
status	Ch1		Ch2		status	Ch1		Ch2		status	Ch1		Ch2		status	Ch1		Ch2	
Byte0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

3.5.1. Main Program Flow

Flow chart of the main program is depicted in Figure 19. The main program first initializes the USART in Master SPI mode with interrupts initialized. Then the CS signal is set to low to enable the ADC chip. The program subsequently enters ECG event processing routine, the ECG data received from the SPI interface will be stored in a receive buffer and sent it out when the package is ready.

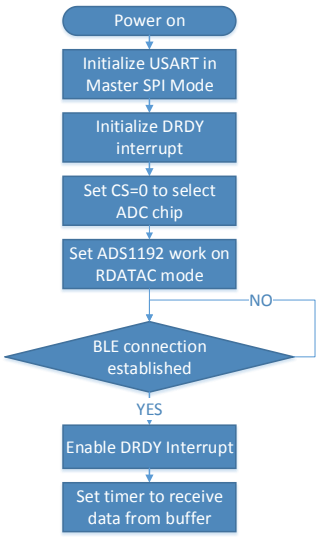


Figure 19: Main program flow

3.5.2. USART Initialization

The USART initialization subroutine performs the necessary operations to configure the USART in Master SPI mode. The flow chart is shown in Figure 20.

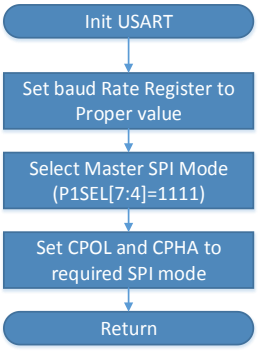


Figure 20: USART initialization

3.5.3. Receive Interrupt Handler

This interrupt handler is called when the USART has received a byte. In Master SPI mode, it happens automatically when a byte is transmitted. The interrupt handler puts the received value into the receive buffer. The flow chart is depicted in Figure 21.

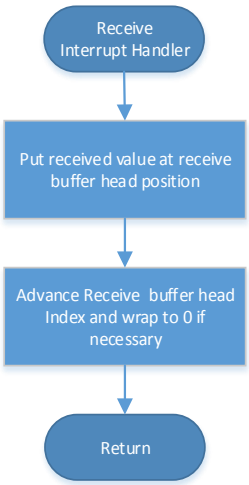


Figure 21: Receive interrupt handler

3.5.4. Retrieve Data

As illustrated in Figure 22, the main program calls this subroutine when it wants to read bytes that have been received in the interrupt handler. It retrieves the data from the receive buffer and transmits the assembled notification packet to the Android application calling BLE API.

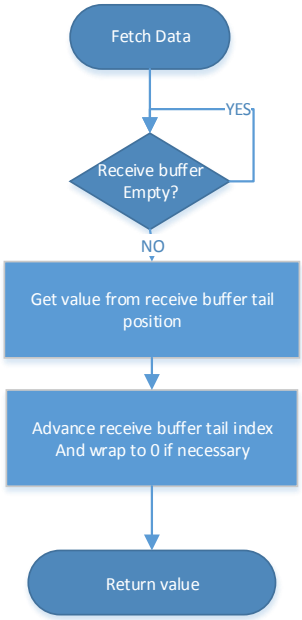


Figure 22: Retrieve data

3.6. Debugging Procedure

The above context present detailed flow charts of each component of the firmware development. Besides, it is common to know that debugging strategy is also an important procedure and necessary part of a professional firmware development. In the following, the debugging procedure carried out in this project is elaborated with some examples.

- The ADS1192 has twelve registers, and Table 3 shows the registers assignments. Please refer to [12] to see a detailed description of these registers. During the debugging procedure, the first thing we need to confirm is that the ADS1192 registers are set to the proper value. The default value we need to write to registers is carried out by the codes below.

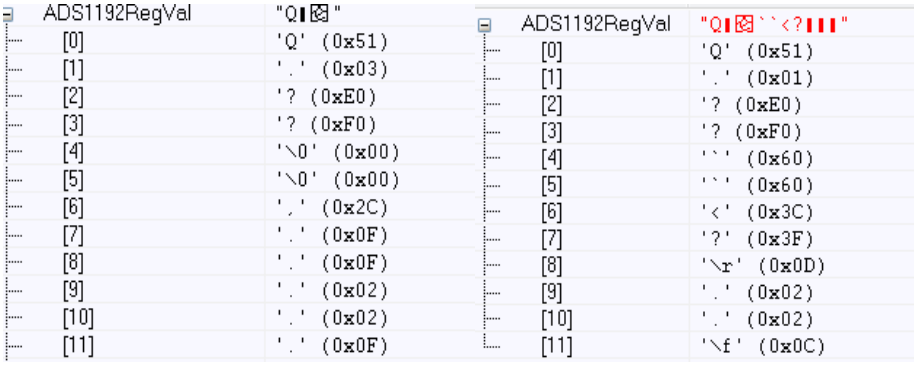
Table 3: Register assignment

AD-DRESS	REG-ISTER	RESET VALUES	BIT8	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1
Device Settings										
00h	ID	XX	REV_ID7	REV_ID6	REV_ID5	1	0	0	REV_ID2	REV_ID0
Global Setting Address Channels										
01h	CON-FIG1	02	SING_SHOT	0	0	0	0	DR2	DR1	DR0
02h	CON-FIG2	80	1	PDB_LOFF_COMP	PDB_REFBUF	VREF_4V	CLK_EN	0	INT-REST	TEST_FREQ
03h	LOFF	10	COMP_TH2	COMP_TH1	COMP_TH0	1	ILEAD_OFF1	ILEAD_OFF0	0	FLEAD_OFF
Channel Specific Settings										
04h	CH1SET	00	PD1	GAIN1_2	GAIN1_1	GAIN1_0	MUX1_3	MUX1_2	MUX1_1	MUX1_0
05h	CH2SET	00	PD2	GAIN2_2	GAIN2_1	GAIN2_0	MUX2_3	MUX2_2	MUX2_1	MUX2_0
06h	RLD_SENS	00	0	0	PDB_RLD	RLD_LOFF_SENS	RLD2N	RLD2P	RLD1N	RLD1P
07h	LOFF_SNES	00	0	0	RLIP2	RLIP1	LOFF2N	LOFF2P	LOFF1N	LOFF1P
08h	LOFF_STAT	00	0	CLK_DIV	0	RLD_STAT	IN2N_OFF	IN2P_OFF	IN1N_OFF	IN1P_OFF
GPIO and Other Registers										
09h	MISC1	00	0	0	0	0	0	0	1	0
0Ah	MISC2	02	CAL-IB_ON	0	0	0	0	0	RLDREF_INT	0
0Bh	GPIO	0C	0	0	0	0	GPIOC2	GPIOC1	GPIOD2	GPIOD1

```
/* ADS1192 Register values*/
uint8  ADS1192_Default_Register_Settings[12] = {
     0x00,       //Device ID read Ony
     0x01,       //CONFIG1   250 SPS
     0xE0,       //CONFIG2
     0xF0,       //LOFF
     0x60,       //CH1SET (PGA gain = 12)
     0x60,       //CH2SET (PGA gain = 12)
     0x3C,       //RLD_SENS
     0x3F,       //LOFF_SENS
     0x00,       //LOFF_STAT
     0x02,       //MISC1
     0x02,       //MISC2
```

```
    0x0C        //GPIO
};
```

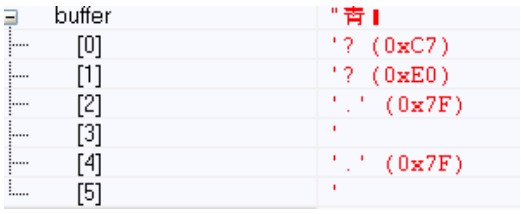
Having set the registers, we can track the source code to see whether the register value read from ADS1192 are identical with the default setting value. The debugging result in our software development is listed as follows. Note that the register values listed on the left are read before setting, while the register values listed on the right are read after setting which are exactly the same with the default setting value.



ADS1192RegVal	"Q▮⊠"
[0]	'Q' (0x51)
[1]	'.' (0x03)
[2]	'? (0xE0)
[3]	'? (0xF0)
[4]	'\0' (0x00)
[5]	'\0' (0x00)
[6]	',' (0x2C)
[7]	'.' (0x0F)
[8]	'.' (0x0F)
[9]	'.' (0x02)
[10]	'.' (0x02)
[11]	'.' (0x0F)

ADS1192RegVal	"Q▮⊠``<?▮▮▮"
[0]	'Q' (0x51)
[1]	'.' (0x01)
[2]	'? (0xE0)
[3]	'? (0xF0)
[4]	'`' (0x60)
[5]	'`' (0x60)
[6]	'<' (0x3C)
[7]	'?' (0x3F)
[8]	'\r' (0x0D)
[9]	'.' (0x02)
[10]	'.' (0x02)
[11]	'\f' (0x0C)

- As mentioned in Subsection 3.3, the number of bits obtained in each interrupt handler is 48 bits, equivalently 6 bytes including 2 status bytes and 2 data bytes for each channel. Therefore, each time we enter in the interrupt handler, 6 bytes can be received. For a test, the receive buffer shows the value below:



buffer	"青▮
[0]	'? (0xC7)
[1]	'? (0xE0)
[2]	'.' (0x7F)
[3]	'
[4]	'.' (0x7F)
[5]	'

According to the format of the 16 bit status 1100 + LOFF_STAT[4:0] + GPIO[1:0] + 5 zeros, we can see the first two bits received from the DOUT are exactly the ones as expected which conform the correctness of our developed routine of fetching data.

- Figure 23 shows the ECG data captured by the USB Dongle on the air. As we can see from the figure, each notification packet consists of 20 bytes containing the following:

  1) ECG Sample1: 1 status byte following two channel data with 2 bytes respectively.
  2) ECG Sample2: 1 status byte following two channel data with 2 bytes respectively.
  3) ECG Sample3: 1 status byte following two channel data with 2 bytes respectively.
  4) ECG Sample4: 1 status byte following two channel data with 2 bytes respectively.

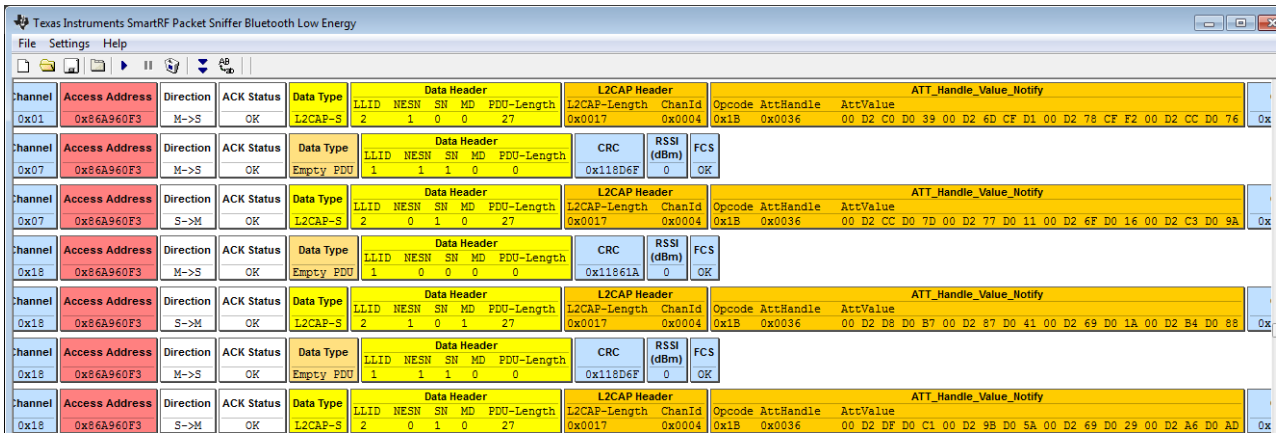


Figure 23: Notification packet

Table 4 lists an example of the ECG notification packet consisting of 20 bytes and the format.

Table 4: An example of the ECG notification packet

Byte Number	Value	Description
0	00	Lead-Off status
1	D2	ECG Sample1: Channel 1 ADC High byte
2	C0	ECG Sample1: Channel 1 ADC Low byte
3	D0	ECG Sample1: Channel 2 ADC High byte
4	39	ECG Sample1: Channel 2 ADC Low byte
5	00	Lead-Off status
6	D2	ECG Sample2: Channel 1 ADC High byte
7	6D	ECG Sample2: Channel 1 ADC Low byte
8	CF	ECG Sample2: Channel 2 ADC High byte
9	D1	ECG Sample2: Channel 2 ADC Low byte
10	00	Lead-Off status

11	D2	ECG Sample3: Channel 1 ADC High byte
12	78	ECG Sample3: Channel 1 ADC Low byte
13	CF	ECG Sample3: Channel 2 ADC High byte
14	F2	ECG Sample3: Channel 2 ADC Low byte
15	00	Lead-Off status
16	D2	ECG Sample4: Channel 1 ADC High byte
17	CC	ECG Sample4: Channel 1 ADC Low byte
18	D0	ECG Sample4: Channel 2 ADC High byte
19	76	ECG Sample4: Channel 2 ADC Low byte

- Finally, the test signals internally generated by ADS1192 are used to verify that whether the whole routine process is correct or not as well as the default register setting values. We can intentionally configure the following three related registers *CONFIG2*, *CH1SET* and *CH2SET* with 0xE3, 0x65 and 0x65, respectively, which will enable ADS1192 internally generate a square wave at 1HZ with amplitude 1 mv. As we set the programmable gain to the maximum value 12, the received signal displayed in the Android application should be a square wave at 1 HZ with amplitude 12 mv. Figure 24 shows the exactly expected signal.

- During the whole test, the BLE connection parameters setting especially the connection interval is a big concern. Either too small or too big value will cause packet loss. We did a quite a lot of tests to find the proper value. The parameter values for final setting are shown below:

```
// Whether to enable automatic parameter update request when a connection
is formed
#define DEFAULT_ENABLE_UPDATE_REQUEST        TRUE
// Minimum connection interval (units of 1.25ms, 80=100ms)
#define DEFAULT_DESIRED_MIN_CONN_INTERVAL     20
// Maximum connection interval (units of 1.25ms)
#define DEFAULT_DESIRED_MAX_CONN_INTERVAL     20
// Slave latency to use if automatic parameter update request is enabled
#define DEFAULT_DESIRED_SLAVE_LATENCY        0
// Supervision timeout value (units of 10ms 2000=20)
#define DEFAULT_DESIRED_CONN_TIMEOUT         1000
```

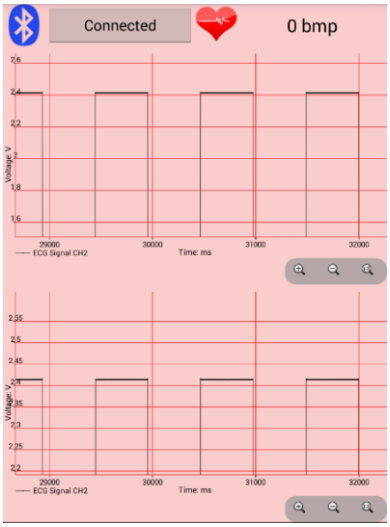


Figure 24: Square signal display

3.7. Final Result Display

Thus far, we finally obtained the normal electrode input signal coming from the human body. The related three registers *CONFIG2*, *CH1SET* and *CH2SET* should be set to 0xE0, 0x60 and 0x60 respectively. The results are successfully shown below in Figure 25.

Figure 25: Normal ECG display

Chapter 4 Implementation of UBL for CC2540

The last but not least important contribution of this project is to feature the ECG system with an easy firmware update by enabling UBL for CC2540. This chapter describes the implementation of UBL for CC2540 starting with an introduction of the principle of UBL. Detailed procedures for implementing UBL are then presented with development flow charts. For reference, an instruction on an easy firmware update using the UBL is also presented.

4.1. Introduction

CC2540 features universal serial bus (USB) access. A product with a USB bootloader allows easy firmware update. In order to develop a customer-oriented product, it is, therefore, a good choice to implement the USB bootloader in the ECG Sensor device. TI has already provided a UBL sample application in its BLE protocol stack package which uses a so-called mass storage device (MSD) class. Accordingly, in the following, we refer our bootloader as the USB-MSD bootloader.

In order to implement this USB-MSD bootloader in our ECG sensor device, it is necessary to first well understand the principle of UBL. Note that though TI has provided some example codes, this project re-developed the codes specified for our ECG sensor hardware by following the basic procedures of the example codes. With our developed codes, when the ECG sensor with the USB-MSD bootloader is connected to a host computer through a USB cable, the bootloader will enumerate as a new drive which allows user application codes to be programmed to the device. This feature perfectly addresses the difficulty for the user to update the new application using a CC Debugger.

The newly developed USB-MSD bootloader for our ECG Sensor has the following advantages:

- No driver is required to be installed on the host computer.

- No specific supporting application is needed to run on the host computer.
- It can be easily done by any user.

Currently, the USB-MSD does not work on Linux or Unix operating systems, because the MSD class developed by TI only supports the Windows file system. As a result, function test can only be done on Windows computers.

4.2. Bootloader Overview

The bootloader is a small application used to erase flash and load user applications to a device. USB-MSD bootloader provides an easy and reliable way of loading user applications to devices. The USB-MSD bootloader will enumerate on the USB as a MSD device after being plugged into the host computer, appearing to the Windows USB host as a generic flash drive. Easily dragging and dropping a user application into the flash drive, the bootloader will load the user application code and program it in the device. Then, the user application becomes available in the device. Figure 26 [13] shows the memory map of the CC2540 bootloader system.

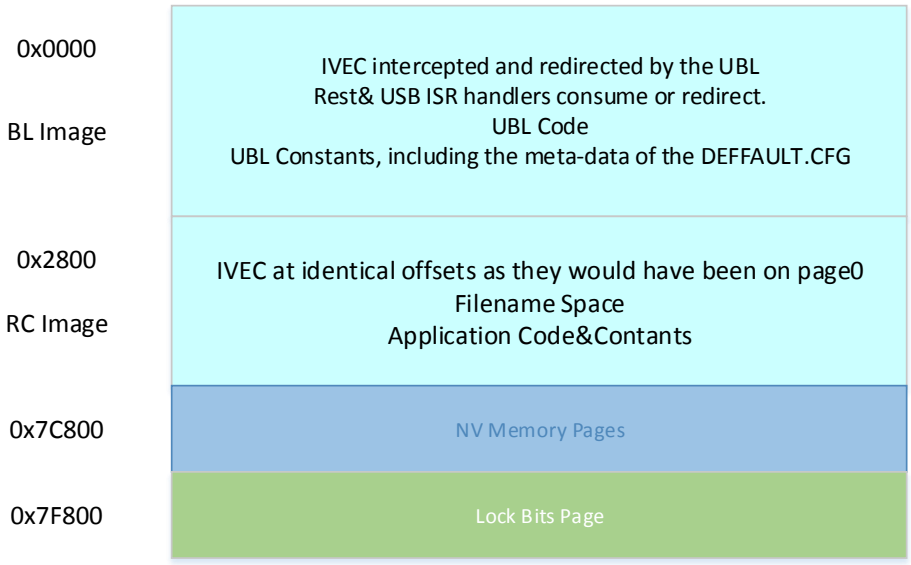


Figure 26: Typical memory map

The CC2540F256 flash memory contains up to 128 pages of 2048 bytes. The last page, page 127, contains the lock bits which cannot be programmatically erased or written to. The USB-MSD bootloader code occupies the first five flash pages.

The default interrupts and exception vectors, used by the USB-MSD bootloader, are put into the starting address of the flash area which must not be altered. The user application interrupt and exception vectors must be put into the application flash area and copied to the memory in the application startup routines.

The USB-MSD bootloader fakes a file allocation table (FAT) file system to the USB host computer. It only supports two files. The first one is the configuration file usually named DEFAULT.CFG. The second is the application image file with .bin as the most typical extension, though it can have any extension. The contents of the configuration file must be exactly a meta-structure as described in Table 5 and defined in the ubl_app.h module in our ECG project.

Table 5: USB-MSD specific meta-data structure

Offset	Len	Name	Description
0	2	CRC-RC	CRC of the RC image, not including the meta-data or any pages that are not enabled for writing
2	2	CRC-SHDW	Shadow of the CRC-RC which is written to non-0xFF only by the UBL.
4	2	CHECK-MD	Checksum of this meta-data, not including the CRC-RC or the CRC-SHDW.
6	2	DELAY-JUMP	Milliseconds to delay waiting for an indication (by GPIO, if enabled, and/or other indication received via the transport bus) to force boot loader mode.
8	32	SEC-KEY	256-bit security key that must be used to unlock the UBL in order to change the allow access bit arrays; Erase-Enable, Write-Enable, and Read-Lock.
40	16	ERASE-Enable	Page erase enable bit-mask of all flash pages.
56	16	WRITE-ENABLE	Page write enable bit-mask of all flash pages.
72	16	READ-Lock	Page read lock bit-mask of all flash pages.
88	1	CFG-DISCS	A byte array of configuration discrete bits.
89	1	GPIO-Port	A configuration byte to specify the Port and Pin of the GPIO control.
90	2	CFG-Spare	Spare configuration bytes for padding.
92	1	CNTDN-Forced	Count down the number of hard resets to force boot loader mode.
93	3	PAD1	Un-used bytes to pad to a 4-byte boundary for multiple writes to CNTDN.
96	1	CNTDN-SecKey	Count down attempted cfg-file hacks with a bad

			SEC-KEY.
97	3	PAD2	Un-used bytes to pad to a 4-byte boundary for multiple writes to CNTDN.

In this meta-data structure, we should pay more attention to the content of GPIO-Port, a configuration byte to specify the Port and Pin of the GPIO control which is used as an indication to force bootloader mode. The configuration value is calculated by the port and pin of the GPIO. For example, Port_0, Bit_1 would be 0*8+1=1; Port_1, Bit_3 would be encoded as 1*8+3 = 11, and so forth.

After the USB-MSD bootloader is downloaded to the device, only the configuration file exists in the flash disk. At this stage, simply dragging and dropping the application image file to the generic flash drive, the device will immediately reset and run the application code. When an application image file already exists on the generic flash drive, it must first be deleted before copying the new image to the bootloader.

4.3. Bootloader Software Architecture

The architecture of the USB MSD bootloader is shown in Figure 27.

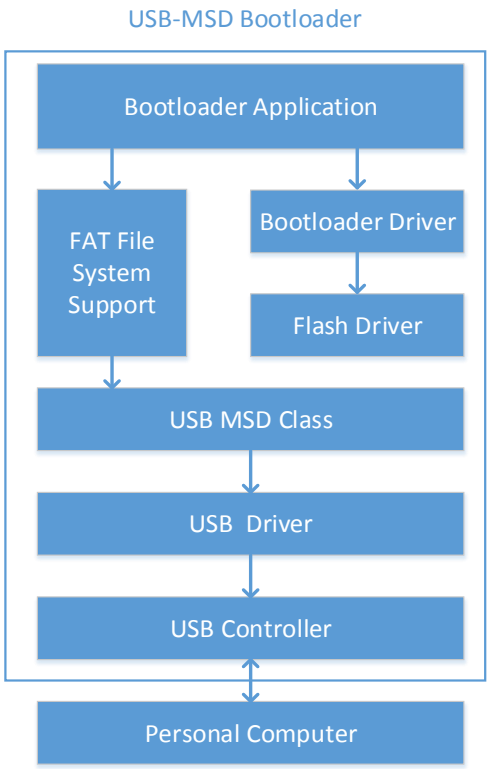


Figure 27: USB MSD bootloader architecture

• The bootloader application controls the loading process. It uses the FAT file system supporting module to read an image file and then uses the bootloader driver to program the image file to the flash memory of the device.

• The FAT file system supporting module allows the bootloader application to read a file from a personal computer with FAT32 format.

• The bootloader driver parses an image file and programs it to the flash memory.

• The flash driver supports erasing, reading, and writing of flash memory.

• USB MSD class provides API specified in the MSD class.

• USB driver and USB controller: Communicate with the USB host through USB protocols.

4.4. Bootloader Software Flow

The whole bootloader system has two parts including an independent USB-MSD bootloader and a UBL-enabled user application. The UBL-enabled user application performs the main function of the product. At reset, the bootloader executes and checks if the specific key is pressed or not to determine whether the application should start or the bootloader should start. If the specific key is pressed down, it will enter the bootloader mode, using USB to enumerate with the host computer. During this enumeration process, the device declares itself as an MSD, and the host then creates a new drive in the system. An application image then can be copied to the drive. After the application image file has been transferred to the device and the device reprograms itself. After reset, the device runs the application code and re-enumerates with the host.

4.5. Using the Bootloader

4.6.1. Preparing the Image File

Please refer to Appendix C to prepare the UBL-enabled application.

4.6.2. Using the Bootloader on ECG Sensor board

After plugging both USB cables into the board and the computer, the ECG Sensor device flashed with bootloader will enumerate on USB as a MSD device. Figure 28 shows the bootloader drive in the Windows 7 operating system. We can see that the volume name of the drive is Removable Disk, and the configuration file is DEFAULT.CFG. The bootloader is now ready to receive a firmware image file. Then, we need to copy the UBL-enabled image file and paste it into the bootloader drive. Simply dragging and dropping the file also works. After resetting the board, the bootloader will jump to the start of the application code and begin execution.

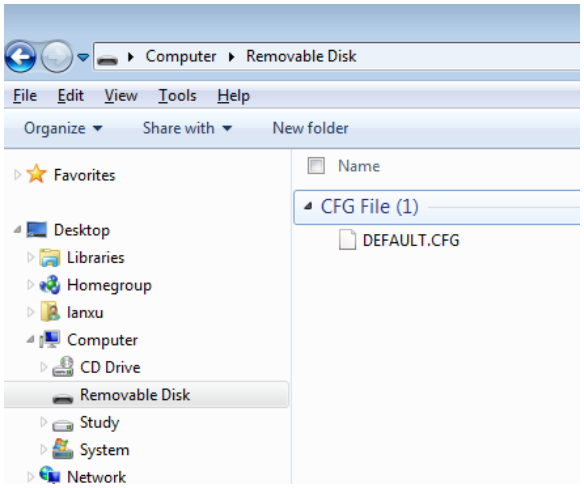


Figure 28: Bootloader drive after enumeration

Appendix D details the procedures of how to download and update the firmware image in a simple way.

Chapter 5 Conclusion and Future Work

5.1. Conclusion

This project focuses on the firmware development for the ECG sensor. The firmware enables the ECG sensor sample, analyze and transmit the heart signals to an Android application via BLE. The software is verified in real time tests. From the test, we observe that the ECG sensor works smoothly, it can be scanned by the Android application when powered on and then transmits the real time ECG signal to Android application continuously and precisely.

Supported by the software application, two channel signals are available for doctors to detect irregular heartbeats and diagnose possible heart disease. Moreover, the USB 2.0 compliance makes the system better meet the market needs, and it is easier for the customer to update the latest firmware in future use.

5.2. Future Work

Currently, the ECG signals are susceptible to motions of the customers. Sometimes, a slight action can possibly cause the instability of the detected signals. Further improvement is needed to mitigate this phenomenon.

For now, there is still a baseline drift issue existing in the ECG sensor, which could be better addressed in the future work.

Bibliography

[1] Texas Instruments CC2540/41 Bluetooth® Low Energy Sample Applications Guide. Available online: http://www.ti.com/lit/ug/swru297c/swru297c.pdf

[2] Software Installer. Available online: www.ti.com/ble-stack

[3] Wikipedia – the free encyclopedia, "Bluetooth Low Energy (BLE)", Available online: http://en.wikipedia.org/wiki/Bluetooth_low_energy

[4] Kevin Townsend, Carles Cufi, Akiba and Robert Davison: Getting Started with Bluetooth Low Energy, published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2014.

[5] Specification of the Bluetooth System, version 4.0, June 2010, Available at http://www.bluetooth.org

[6] Specification of the Bluetooth System, Covered Core Package, Version: 4.0; The Bluetooth Special Interest Group: Kirkland, WA, USA, 2010.

[7] Bluetooth Low Energy (BLE): http://www.cypress.com/file/139326/download

[8] Carles Gomez, Joaquim Oller and Josep Paradells: Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology, Sensors 2012, 12, 11734-11753; doi:10.3390/s120911734.

[9] CC2540 and CC2541 Bluetooth® low energy Software Developer's Reference Guide. Available at http://www.ti.com/lit/ug/swru271g/swru271g.pdf

[10] Texas Instruments CC2540/41 Bluetooth® Low Energy Software Developer's Guide v1.4. TI_BLE_Software_Developer's_Guide.pdf

[11] Https://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus#Interface

[12] Low-Power, 2-Channel, 16-Bit Analog Front-End for Biopotential Measurements http://www.ti.com/lit/ds/symlink/ads1192.pdf

[13] Universal Boot Loader for SOC-8051 by USB-MSD Developer's Guide.pdf.

[14] Mohamed Habbal: Bluetooth low energy – assessment within a competing wireless world, in *Proc. of Wireless Congress 2012- Systems & Applications*, 2012.

[15] Kamath, S. Measuring Bluetooth Low Energy Power Consumption; Application Note AN092; Texas Instruments: Dallas, TX, USA, 2010.

[16] Kamath, S. Measuring Bluetooth Low Energy Power Consumption; Application Note AN092; Texas Instruments: Dallas, TX, USA, 2010.

[17] Ko, J.; Terzis, A.; Dawson-Haggerty, S.; Culler, D.E.; Hui, J.W.; Levis, P. Connecting low-power and lossy networks to the internet. *IEEE Commun. Mag.*, 2011, 49, 96–101.

[18] Chang, K.-M.; Liu, S.-H.; Wu, X.-H. A Wireless sEMG recording system and its application to muscle fatigue detection. *Sensors*, 2012, 12, 489–499.

[19] Nakamura, M.; Nakamura, J.; Lopez, G.; Shuzo, M.; Yamada, I. Collaborative processing of wearable and ambient sensor system for blood pressure monitoring. *Sensors*, 2011, 11, 6760–6770.

[20] West, A. Smartphone, the key for Bluetooth low energy technology. Available online: http://www.bluetooth.com/Pages/Smartphones.aspx (accessed on 24 June 2012).

[21] Hui, J.W.; Culler, D.E. Extending IP to low-power, wireless personal area networks. *IEEE Internet Comput.*, 2008, 12, 37–45.

[22] Nieminen, J.; Patil, B.; Savolainen, T.; Isomaki, M.; Shelby, Z.; Gomez, C. Transmission of IPv6 packets over Bluetooth low energy draft-ietf-6lowpan-btle-08. 2012, in preparation.

[23] Haartsen, J.C. The Bluetooth radio system. *IEEE Pers. Commun.*, 2000, 7, 28–36.

[24] Zheng, J.; Lee, M.J.; Anshel, M. Towards secure low rate wireless personal area networks. *IEEE Trans. Mob. Comput.*, 2006, 5, 1361–1373.

[25] Echevarría, J.J.; Ruiz-de-Garibay, J.; Legarda, J.; Álvarez, M.; Ayerbe, A.; Vázquez, J.I. WebTag: Web browsing into sensor tags over NFC. *Sensors*, 2012, 12, 8675–8690.

[26] Callegati, F.; Cerroni, W.; Ramili, M. Man-in-the-Middle attack to the HTTPS protocol. *IEEE Secur. Priv.*, 2009, 7, 78–81.

Appendix A ADS1192 Setup Code

```
//ADS1192 Setup after power on
void ADS1192_SPISetup(void)
{
   volatile uint16 i, j;
   OnBoard_PortsInit();

   ADS1192_Reset();
   for (j = 0; j < DELAY_COUNT; j++)
   {
      for ( i = 0; i < 20000; i++);
   }

   Init_ADS1192_DRDY_Interrupt();          //Initial DRYIn interrupt
   Stop_Read_Data_Continuous();            // Set SDATAC command


   ADS1192_Read_All_Regs(ADS1192RegVal);
   ADS1192_Default_Reg_Init();
   ADS1192_Read_All_Regs(ADS1192RegVal);

   ADS1192_Disable_Start();                // Disable START(SET START to high)

   Set_ADS1192_Chip_Enable();              // CS =0
   for(uint16 i=0; i<300; i++);
   Start_Read_Data_Continuous();           //RDATAC command
   for(uint16 i=0; i<300; i++);

}

//Port Initialization
void OnBoard_PortsInit(){
  /* Mode select UART1 SPI Mode as master. */
   U1CSR = 0;

   /* Setup for 115200 baud. */
   U1GCR |= 0x0B;
   U1BAUD = 0xD8;

   /* Set bit order to MSB */
   U1GCR |= BV(5);
   /* Set CPOL = 0 and CPHA = 1*/
   U1GCR |= BV(6);

   /* Set UART1 I/O to alternate 2 location on P1 pins. */
   PERCFG |= 0x02;    /* U1CFG */

   /* Select peripheral function on I/O pins */
   P1SEL |= 0xE0;    /* SELP1_[7:4] */
   /* P1.0,1,2,3: , ADC_Start, ON/BYP, VCC_LP, CS_AFE. */
   P1SEL &= 0xF0;

   /* When SPI config is complete, enable it. */
   U1CSR |= 0x40;

}
```

Appendix B User Register Description

The overall user register description is detailed in [15]. Here lists several registers of importance.

```
#define ADS1192_REG_CONFIG1            (0x0001u)
/*
  * CONFIG1: Configuration Register 1
  * Address = 01h
  *-------------------------------------------------------------------
  * |BIT 7            |BIT 6 |BIT 5    |BIT 4 |BIT 3 |BIT 2 |BIT 1    |BIT 0 |
  * |-----------|------|-------|------|------|------|-------|------|
  * |SINGLE_SHOT |     0    |     0     | 0       |    0     | DR2 | DR1    | DR0 |
  *-------------------------------------------------------------------
  * Bit 7 SINGLE_SHOT: Single-shot conversion
  *    This bit sets the conversion mode
  *           0 = Continuous conversion mode (default)
  *           1 = Single-shot mode
  *    NOTE: Pulse mode is disabled when individual data rate control is selected.
  * Bits[6:3] Must be set to '0'
  *    Bits[2:0] DR[2:0]: Channel oversampling ratio
  *    These bits determine the oversampling ratio of both channel 1 and channel 2.
  * ------------------------------------------------------------
  *           BIT  |      OVERSAMPLING RATIO      |      SAMPLE RATE |
  *           000  |      fMOD/1024                  |      125SPS          |
  *           001  |      fMOD/512                   |      250SPS          |
  *           010  |      fMOD/256                   |      500SPS          |
  *           011  |      fMOD/128                   |      1KSPS           |
  *           100  |      fMOD/64                    |      2KSPS           |
  *           101  |      fMOD/32              |      4KSPS           |
  *           110  |      fMOD/16                    |      8KSPS           |
  *           111  |      Do Not Use                 |      Do Not Use      |
  * ------------------------------------------------------------
  */
#define ADS1192_REG_CONFIG2            (0x0002u)
/*
  * CONFIG2: Configuration Register 2
  * Address = 02h
  *-----------------------------------------------------------------------------------
  * |BIT 7    |BIT 6              |BIT 5           | BIT 4     |    BIT 3 |BIT 2    |BIT 1        |BIT 0       |
  * |-------|--------------|----------|---------|--------|-------|---------|---------|
  *  |      1       |PDB_LOFF_COMP  |PDB_REFBUF|  VREF_4V  |  CLK_EN  |      0       |INT_TEST
|TEST_FREQ|
```

```
 *-----------------------------------------------------------------------------
 * Configuration Register 2 configures the test signal generation. See the Input Multiplexer section for more details.
 *    Bit 7 Must always be set to '1'
 *    Bit 6 PDB_LOFF_COMP : Lead-off comparator power-down
 *    This bit powers down the lead-off comparators.
 *    0 = Lead-off comparators disabled (default)
 *    1 = Lead-off comparators enabled
 *    Bit 5 PDB_REFBUF : Reference buffer power-down
 *          This bit powers down the internal reference buffer so that the external reference can be used.
 *    0 = Reference buffer is powered down (default)
 *          1 = Reference buffer is enabled
 *    Bit 4 VREF_4V: Enables 4-V reference
 *          This bit chooses between 2.4V and 4V reference.
 *          0 = 2.4-V reference (default)
 *          1 = 4-V reference
 *    Bit 3 CLKOUT_EN: CLK connection
 *          This bit determines if the internal oscillator signal is connected to the CLK pin when an internal oscillator is used.
 *          0 = Oscillator clock output disabled (default)
 *          1 = Oscillator clock output enabled
 *    Bit 2 Must be set to '0'
 *    Bit 1 TEST_AMP: Test signal amplitude
 *          This bit determines the test signal amplitude.
 *          0 = No test signal (default)
 *          1 = ?VREFP ?VREFN)/2400
 *    Bit 0 TEST_FREQ: Test signal frequency.
 *          This bit determines the test signal frequency.
 *          0 = At dc (default)
 *          1 = Square wave at 1 Hz
 */
#define ADS1192_REG_CH1SET              (0x0004u)
/*    CHnSET: Individual Channel Settings
 *    Address = 04h
 *-----------------------------------------------------------------------------
 * | BIT 7       |BIT 6       |BIT 5     |    BIT 4    | BIT 3    | BIT 2    |    BIT 1    | BIT 0        |
 * |-----------|---------|--------|---------|--------|--------|---------|---------|
 * |     PD1       |     GAIN1_1 |GAIN1_0 |    0          |    MUX1_3| MUX1_2 |    MUX1_1 | MUX1_0 |
 *-----------------------------------------------------------------------------
 * The CH1SET Control Register configures the power mode, PGA gain, and multiplexer settings channels.
 * Bit 7 PD1: Channel 1 power-down
 *          0 = Normal operation (default)
```

```
 *          1 = Channel 1 power-down
 * Bits[6:4 ] GAIN1[2:0]: Channel 1 PGA gain setting
 *          These bits determine the PGA gain setting for channel 1.
 *          000 = 6 (default)
 *          001 = 1
 *          010 = 2
 *          011 = 3
 *          100 = 4
 *          101 = 8
 *          110 = 12
 * Bits[3:0] MUX1[3:0]: Channel 1 input selection
 *          These bits determine the channel 1 input selection.
 *          0000 = Normal electrode input (default)
 *          0001 = Input shorted (for offset measurements)
 *          0010 = RLD_MEASURE
 *          0011 = VDD/2 for supply measurement
 *          0100 = Temperature sensor
 *          0101 = Cal signal
 *          0110 = RLD_DRP (positive electrode is the driver)
 *          0111 = RLD_DRM (negative electrode is the driver)
 *          1000 = Reserved
 *          1001 = MUX RESPP/RESPN to INP/INM
 *          1010 = Reserved
 */

#define ADS1192_REG_CH2SET              (0x0005u)
/*    CHnSET: Individual Channel Settings
 *    Address = 05h
 *--------------------------------------------------------------------------------
 * | BIT 7      |BIT 6      |BIT 5     |    BIT 4    | BIT 3    | BIT 2    |    BIT 1    | BIT 0       |
 * |----------|---------|--------|---------|--------|--------|---------|---------|
 * |     PD2       |     GAIN2_2 |GAIN2_1 |    GAIN2_0|    MUX2_3| MUX2_2 |    MUX2_1 | MUX2_0
|
 *--------------------------------------------------------------------------------
 *
 * The CH1SET Control Register configures the power mode, PGA gain, and multiplexer settings
channels.
 * Bit 7 PD2: Channel 1 power-down
 *          0 = Normal operation (default)
 *          1 = Channel 1 power-down
 * Bits[6:4 ] GAIN2[2:0]: Channel 1 PGA gain setting
 *          These bits determine the PGA gain setting for channel 1.
 *          000 = 6 (default)
 *          001 = 1
```

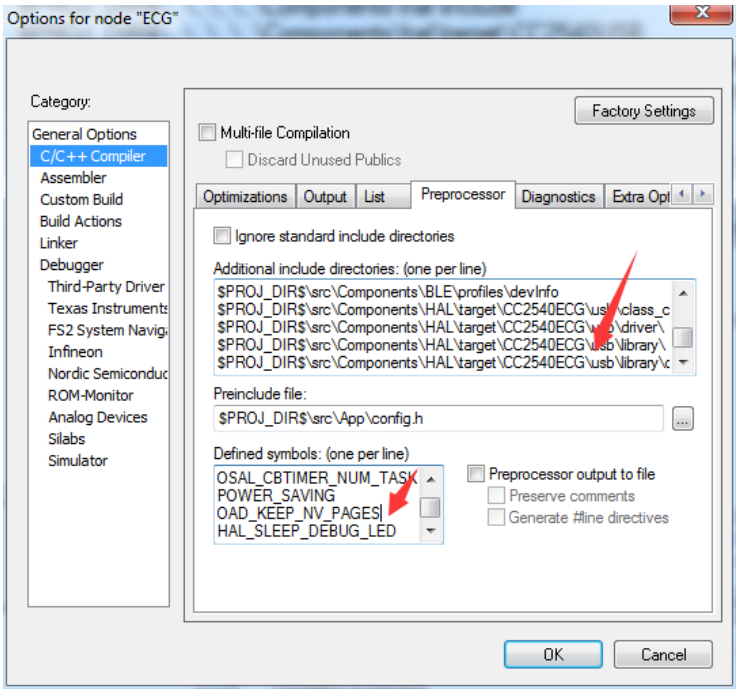
```
 *          010 = 2
 *          011 = 3
 *          100 = 4
 *          101 = 8
 *          110 = 12
 * Bits[3:0] MUX2[3:0]: Channel 1 input selection
 *          These bits determine the channel 1 input selection.
 *          0000 = Normal electrode input (default)
 *          0001 = Input shorted (for offset measurements)
 *          0010 = RLD_MEASURE
 *          0011 = VDD/2 for supply measurement
 *          0100 = Temperature sensor
 *          0110 = RLD_DRP (positive electrode is the driver)
 *          0111 = RLD_DRM (negative electrode is the driver)
 *          1000 = Reserved
 *          1001 = Route IN3P and IN3N to channel 2 inputs
 *          1010 = Reserved
 *          1011 = Reserved
 *          1100 = Reserved
 *          1101 = Reserved
 *          1110 = Reserved
 *          1111 = Reserved
 */
#define ADS1192_REG_LOFF_STAT          (0x0008u)
/*
 * LOFF_STAT
 * Address = 08h
 *-------------------------------------------------------------------------------------------------------------------
 * | BIT 7      |BIT 6      |BIT 5     |    BIT 4        | BIT 3      | BIT 2      |    BIT 1      | BIT 0      |
 * |----------|---------|--------|------------|----------|----------|-----------|------------|
 * |    0    |CLK_DIV    |0     |    RLD_STAT    | IN2N_OFF | IN2P_OFF | IN1N_OFF    | IN1P_OFF
|
 *-------------------------------------------------------------------------------------------
 *    This register stores the status of whether the positive or negative electrode on each channel is on
or off. See the
 *    Lead-Off Detection subsection of the ECG-Specific Functions section for details. Ignore the
LOFF_STAT values
 *    if the corresponding LOFF_SENS bits are not set to '1'.
 *    '0' is lead-on (default) and '1' is lead-off. When the LOFF_SENS bits are '0', the LOFF_STAT bits
should be
 *    ignored.
 *
 *    Bit 7 Must be set to '0'
 *    Bit 6 CLK_DIV: System frequency selection
```

```
*          This bit sets the modultar frequency. Two external clock values are supported: 512 kHz and
2.048 MHz. This bit must be set
*          so that MOD_FREQ = 128 kHz.
*          0 = External_CLK/4 (default)
*          1 = External_CLK/16
*    Bit 5 Must be set to '0'
*    Bit 4 RLD_STAT: RLD lead-off status
*          This bit determines the status of RLD.
*          0 = RLD is connected (default)
*          1 = RLD is not connected
*    Bit 3 IN2N_OFF: Channel 2 negative electrode status
*          This bit determines if the channel 2 negative electrode is connected or not.
*          0 = Connected (default)
*          1 = Not connected
*    Bit 2 IN2P_OFF: Channel 2 positive electrode status
*          This bit determines if the channel 2 positive electrode is connected or not.
*          0 = Connected (default)
*          1 = Not connected
*    Bit 1 IN1N_OFF: Channel 1 negative electrode status
*          This bit determines if the channel 1 negative electrode is connected or not.
*          0 = Connected (default)
*          1 = Not connected
*    Bit 0 IN1P_OFF: Channel 1 positive electrode status
*          This bit determines if the channel 1 positive electrode is connected or not.
*          0 = Connected (default)
*          1 = Not connected
*
*/
```

Appendix C Build UBL-Enabled Application

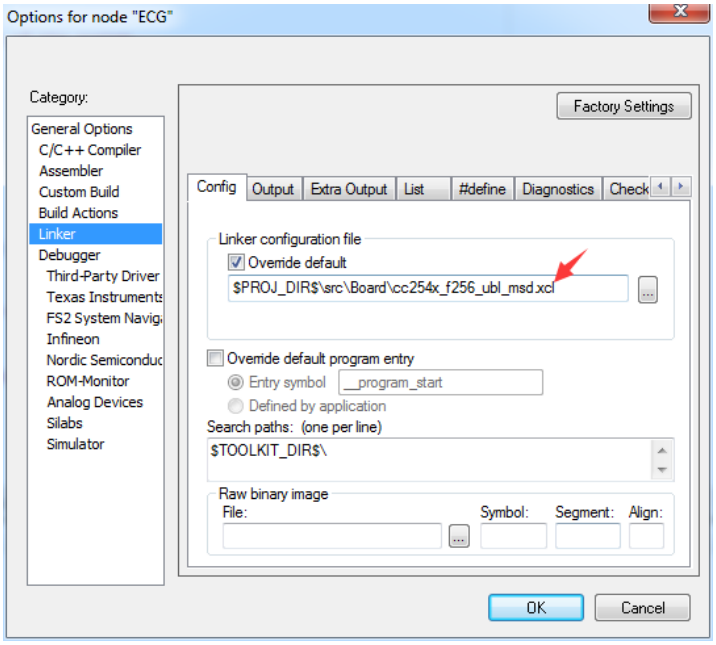
In order to build the UBL-enable application, the following configuration steps need to be done [11].

1. FEATURE_UBL_MSD and OAD_KEEP_NV_PAGES are added to the pre-processor define list, related usb directories must be added to additional in-clude directories.

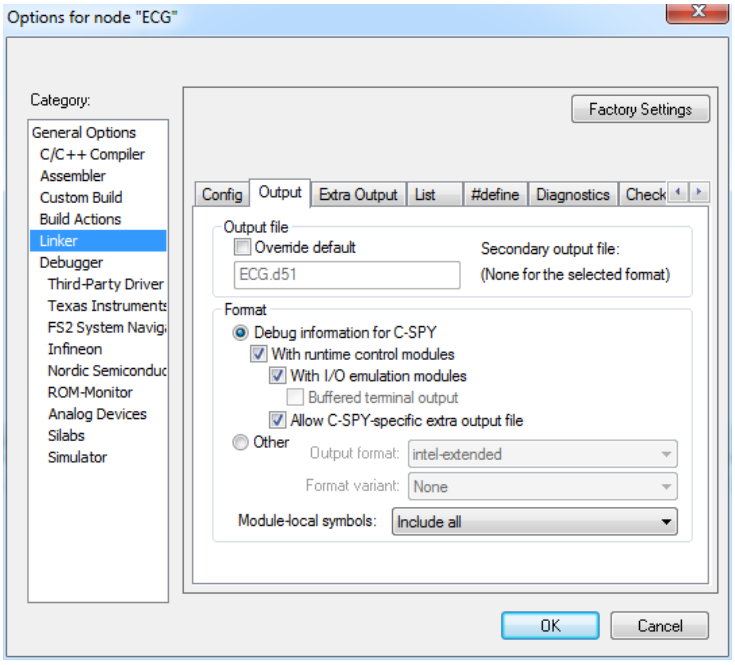


2. The *ubl_app.c* is included in the application project.
   File locaton : $PROJ_DIR$\src\UBL\soc_8051\usb_msd\app. It is linked in through OnBoard.c. The application must have a call to appForceBoot() in order to force IAR to include the data structure in the ubl_app.c module. Such a call is crucial as well to be able to set the “DELAY-JUMP” to zero and have fast power-ups that start running the application immediately without delaying in BL mode.

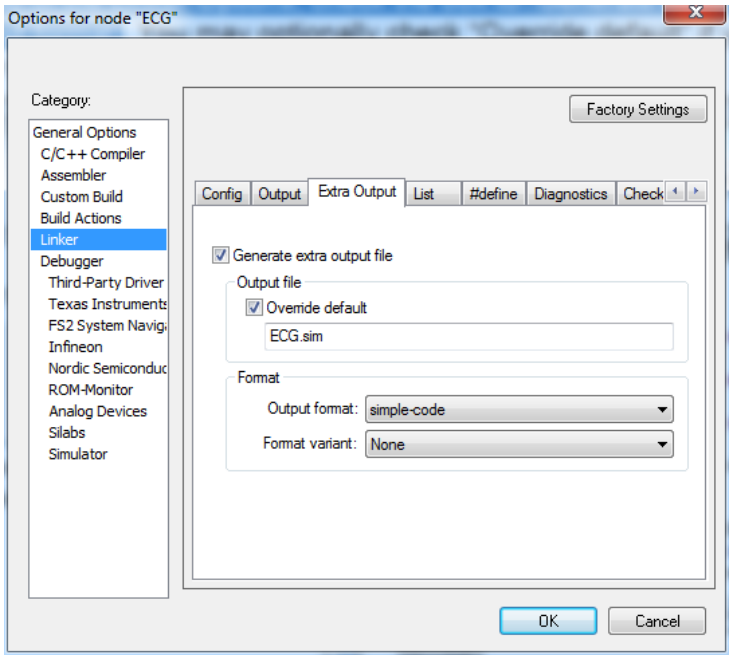
3. In the “Config” tab of the Linker Options, check “Override default” and add linker file “cc254x_f256_ubl_msd.xcl”.



4. In the “Output” tab of the Linker Options, check the box to “Allow C-SPY-specific extra output file”, uncheck the box “Override default”, as shown below:



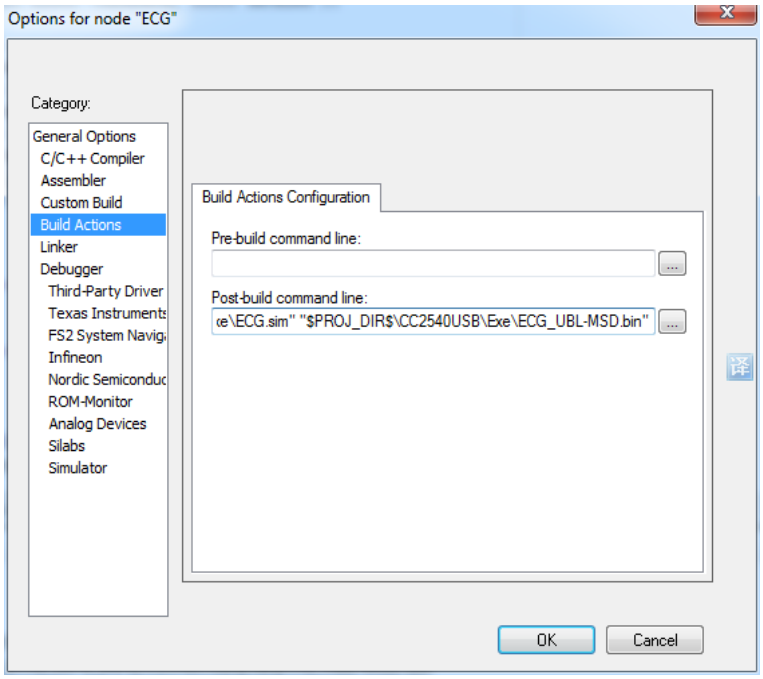
5. In the “Extra Output” tab of the Linker Options, check the box to “Generate extra output file” and choose the “simple-code” output format, optionally check “Override default” to rename the output file name.



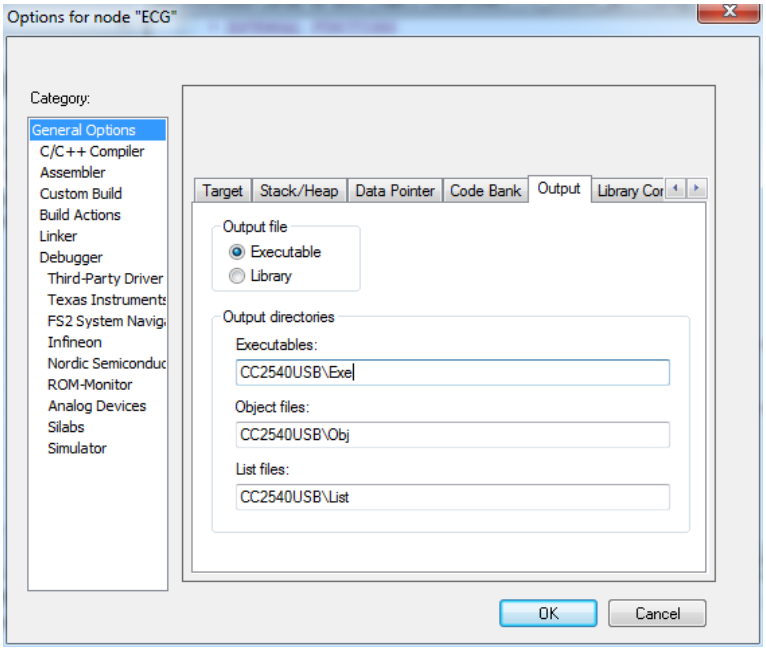
6. On the "Post-build command line" of the Build Actions, paste the following 3 lines as one line with each part separated by a space.

   "$PROJ_DIR$\src\Board\cc254x_sim2bin.exe"
   "$PROJ_DIR$\CC2540USB\Exe\ECG.sim"
   "$PROJ_DIR$\CC2540USB\Exe\ECG_UBL-MSD.bin"

   The first string is an executable which converts a ".sim" file – the second string – into a ".bin" file – the third string.



7. In the Output tab of General Options, change the Output directories as follows:



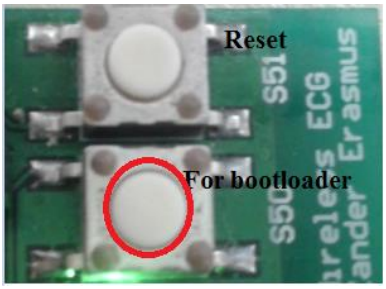
8. Compile the project, find the executable file "ECG_UBL-MSD.bin" from the directory *"$PROJ_DIR$\CC2540USB\Exe\"*

Appendix D Steps to Update A New Firmware Image File

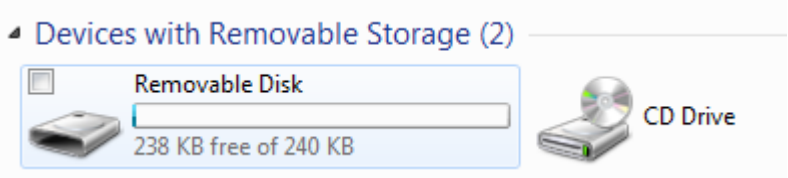
Steps to update new firmware image to ECG sensor on Windows PC.

1. Windows PC is on and running
2. Using a USB cable with micro-USB connector, insert the micro-USB connector to ECG sensor first.

3. Press and Hold the button on ECG sensor board while connecting another end of the USB cable to PC, keep the push button down until ECG sensor appears as Removable Disk on Windows.



4. New hardware is detected by Windows, wait until USB Driver installation to complete (only the first time).
5. A new Removable Dish would appear on Windows, actually this is ECG sensor board.



6. Release the push button on ECG Sensor.
7. Delete any existing firmware on the dish.
8. Copy a new firmware to the disk.
9. ECG sensor will restart automatically and Green LED turned ON. And the Removable Disk cable no longer appears.
10. Disconnect the USB cable from PC and ECG Sensor.
11. Firmware update completed.