ÇUKUROVA UNIVERSITY
INSTITUTE OF NATURAL AND APPLIED SCIENCES

MSc THESIS

Abdinasir Abdullahi FARAH

ELECTRIC VEHICLE BATTERY MANAGEMENT SYSTEM USING BLUETOOTH LOW ENERGY TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

ADANA-2022

ABSTRACT

MSc THESIS

ELECTRIC VEHICLE BATTERY MANAGEMENT SYSTEM USING BLUETOOTH LOW ENERGY TECHNOLOGY

Abdinasir Abdullahi FARAH

ÇUKUROVA UNIVERSITY
INSTITUTE OF NATURAL AND APPLIED SCIENCES
DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

Supervisor : Assoc. Prof. Dr. M. Uğraş CUMA
Year: 2022, Pages: 79
Jury Members : Assoc. Prof. Dr. M. Uğraş CUMA
: Assoc. Prof. Dr. M. Mustafa SAVRUN
: Assist. Prof. Dr. Oğuzhan TİMUR

Wireless Battery Management Systems (WiBMS) became a great topic of interest for researchers recently, as it seems a viable substitution for conventional wire-based BMS. While the conventional BMS faces quite severe problems including safety concerns like electric faults and wire connection complexities that cause added weight and higher costs for manufacturing and maintenance, wireless-enabled BMS offers an alternative solution to these problems.

This thesis work discussed problems that challenged conventional battery management systems and explained why WiBMS is a solution. A Bluetooth Low Energy (BLE) based wireless BMS was developed. A VDA battery module of 12 cells with 44.4V voltage, battery capacity of 156 Ah and energy capacity of 6.96 kWh is used. Two BLE-enabled microcontrollers named CC2640R2F have been used for wireless interfacing. BQ76PL455A-Q1 ASIC was used to monitor and gather voltage data from the battery modules. The code for the Bluetooth stack as well as Real Time Operating System (RTOS) running on CC2640R2F MCU was developed on Code Composer Studio (CCS) development environment by Texas Instruments (TI). The Bluetooth LE stack implements the communication link between the two MCUs.

One MCU functioning as a slave or peripheral connected to the BQ76PL455A-Q1 device reads the voltage data from the battery module and sends it to the other master or central MCU over BLE. In all cases, all data sent from the slave MCU to the central one was successfully transmitted and no data loss was observed. It was suggested that further research should be done about this research area as it has a promising future for improving BMS for EVs.

Keywords: Battery Management System, Bluetooth Low Energy, Wireless BMS,CC2640R2F MCU, Electric Vehicles.

ÖZ

YÜKSEK LİSANS TEZİ

BLUETOOTH DÜŞÜK ENERJİ KULLANARAK ELEKTRİKLİ ARAÇLARIN BATARYA YÖNETİM SİSTEMİ

Abdinasir Abdullahi FARAH

ÇUKUROVA ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
ELEKTRİK ELEKTRONİK MÜHENDİSLİĞİ ANABİLİM DALI

Danışman : Doç. Dr. M. Uğraş CUMA
Yıl: 2022, Sayfa: 79
Jüri üyeleri : Doç. Dr. M. Uğraş CUMA
: Doç. Dr. M. Mustafa SAVRUN
: Dr. Öğr. Üyesi. Oğuzhan TİMUR

Kablosuz Batarya Yönetim sistemi, konvansiyonel kablo tabanlı BMS'nin yerini tutacak gibi göründüğü için, son zamanlarda araştırmacılar için büyük bir ilgi konusu haline gelmiştir. Konvansiyonel BMS, elektrik arızaları gibi güvenlik endişeleri ve üretim ve bakım için ek ağırlık ve daha yüksek maliyetlere neden olan kablo bağlantı karmaşıklıkları gibi oldukça ciddi sorunlarla karşı karşıya kalırken, kablosuz özellikli BMS bu sorunlara alternatif bir çözüm sunmaktadır

Bu tez çalışmasında, konvansiyonel batarya yönetim sistemlerini zorlayan sorunlar üzerinde tartışılmış ve kablosuz BMS'nin neden bir çözüm olduğunu açıklanmıştır. Bluetooth Düşük Enerji (BLE) tabanlı bir kablosuz BMS geliştirilmiştir. 44,4V gerilim, 156 Ah batarya kapasitesi ve 6,96 kWh enerji kapasitesine sahip 12 hücreli bir VDA batarya modülü kullanılmıştır. Kablosuz arayüz için CC2640R2F adlı iki BLE özellikli mikrodenetleyici kullanılmıştır. Batarya modüllerinden voltaj verilerini izlemek ve toplamak için BQ76PL455A-Q1 cihazı kullanılmıştır. CC2640R2F MCU üzerinde çalışan Gerçek Zamanlı İşletim Sisteminin (RTOS) yanı sıra Bluetooth yığınının kodu Texas Instruments tarafından üretilen Code Composer Studio (CCS) geliştirme ortamında geliştirilmiştir. Bluetooth LE yığını iki MCU arasındaki iletişim bağlantısını gerçekleştirmektedir.

BQ76PL455A-Q1 cihazına bağlı slave veya periferik olarak çalışan bir MCU, batarya modülünden voltaj verilerini okur ve BLE üzerinden diğer master veya merkezi MCU'ya gönderir. Tüm durumlarda, bağımlı MCU'dan merkezi olana gönderilen tüm veriler başarıyla iletilmiş ve herhangi bir veri kaybı gözlemlenmemiştir. Elektrikli araçlar için BMS iyileştirmelerine yönelik güvenilir çözümler sunduğu için bu araştırma alanı hakkında daha fazla araştırma yapılması önerilmiştir.

Anahtar Kelimeleri: Batarya Yönetim Sistemi, Bluetooth Düşük Enerji, CC2640R2F, Kablosuz BMS, Elektrikli araçlar.

EXTENDED ABSTRACT

Electric vehicles (EVs) are being developed and seen as the future mode of transportation to reduce hazardous gas emissions, costs, and extra weight. Battery Management System (BMS) is one of the main and most important parts of electric vehicles and other energy storage systems and for that reason, it became a significant area for improvement.

A BMS is an integral part of any hybrid or electric vehicle that manages monitors, and safeguards the battery pack, to ensure the best possible performance and dependability. By measuring voltage, temperature, State of Charge (SoC), State of Health (SoH), and the critical limits of charging and discharging currents, the BMS seeks to maximize the lifespan of batteries.

BMS is needed to supervise the battery pack to provide safer operation and reliable performance as the capacity of the battery pack can easily get changed by the charging-discharging and overcharging or under-discharging cycles as well as overheating. By functionality, during charging and discharging operations, a BMS gathers all data (voltage, current, temperature, and SoC) from the sensors connected to a battery pack and activates the correct circuitry to execute certain suitable actions.

The BMS uses internal sensors inside the battery pack to measure necessary parameters (i.e., voltage, current etc.) for smooth functioning and performance of the battery as well as blocking or minimizing any possible fault and pack failures caused by a single malfunctioning cell. Battery cells and battery sensing modules and control management units are connected by a complicated wire network in traditional battery management systems (BMS).

Eight to fourteen battery cells' worth of sensory data can normally be collected by each slave BMS. A collection of slave BMSs are linked to a master

BMS, which stores the acquired data from the battery cells for later processing to support a high number of battery cells.

The hierarchical BMS architecture has some significant drawbacks despite being widely accepted and utilised. The wiring assembly of a BMS can result in many problems, including physical power interruption in vibratory environments, connection failure from EMI-induced areas, complexity in the design of the battery pack, and challenges in automated manufacturing. Thus, lower reliability and lower productivity could be inherited from these issues.

To seek a solution for the above-mentioned problems faced by the traditional BMS, novel ways and better alternative battery management systems have been proposed by benefitting from the power of wireless technologies. One of the most promising wireless technologies proposed is the ISM-band operating 2.4GHz Bluetooth Low Energy which is relatively new but got wide acceptance among wireless technology enthusiasts.

In this thesis work, at first; battery management systems particularly WiBMS will be thoroughly reviewed. Then, research about several different protocols used for wireless BMS and why Bluetooth Low Energy was preferred will be discussed. Finally, by using Bluetooth LE, a model of a wireless battery management system will be developed.

The proposed wireless BMS was carried out using a BQ76PL455A-Q1 integrated battery monitor and protector as a monitoring unit, two CC2640R2F microcontrollers for Bluetooth communication – one MCU was programmed as a slave and was employed to get the voltage data of the battery modules and send it over BLE while the other MCU was functioning as a central device and displaying the voltage data on the display of a host PC.

As a battery system for the proposed BMS, one module with a 12 serial 2 Parallel (12S2P) configuration VDA battery module was used. Each cell of the module had a 3.7 nominal voltage and 78 Ah capacity. The overall module had a voltage of 44.4V, a battery capacity of 156 Ah and an energy capacity of 6.96

kWh. The communication between MCUs was happening over Bluetooth Low Energy protocol while the Master MCU was communicating with the host computer via USB. BQ76PL455 board and the slave MCU were communicating over serial UART protocol using RS422 UART to TLL cable.

Bluetooth Stack which is developed by Bluetooth Special Interest Group (SIG) and provided as a Software Development Kit (SDK) by Texas Instruments (TI) was used to write the code for the microcontrollers so that they can implement the communication protocol. TI's CCS Integrated Development Environment (IDE) was used for code writing, debugging, and executing.

A slave CC2640R2F MCU is connected across the BQ76PL455A-Q1 to the Battery Module and a command to request voltage data of cells was sent from the BQ76PL455A-Q1 GUI application. The response data from the battery module containing packets of cell voltages were read by the slave MCU using UART RS-422 serial connection. Then, the slave MCU was sending the voltage data to the Master MCU over Bluetooth LE.

The command has been sent from the BQ76PL455A-Q1 GUI application in a hexadecimal form and the voltage data of the battery module was read successfully and displayed in a hexadecimal format by the host PC for confirmation. Data communication to and from the BQ76PL455A-Q1 device was in standard 8 data bit, 1 stop bit, no-parity universal asynchronous receiver and transmitter (UART) format. Every Byte of data in the data packet was shown as a two-character hexadecimal number.

The voltage data sent by the slave MCU over Bluetooth LE was successfully received by the Master MCU. The master MCU displayed the data on a host PC using the Realterm™ serial emulator over UART. The command to request voltage data was sent several times and every time the data transfer and communication between the slave MCU and the Master MCU was successful. The successful transaction of data shows that the system is real-time and reliable.

The transfer of the battery module data over Bluetooth Low Energy shows that some of the problems faced by the traditional wired BMS could be solved by the BLE-enabled BMS. This experiment also contributes to the previously done research about this specific topic and suggests that further research needs to be carried out on the wireless BMS field

GENİŞLETİLMİŞ ÖZET

Elektrikli araçlar (EA), zararlı gaz emisyonlarını, maliyetleri ve ekstra ağırlığı azaltmak için geliştirilmekte olup geleceğin ulaşım şekli olarak görülmektedir. Batarya Yönetim Sistemi (BMS), elektrikli araçların ve diğer enerji depolama sistemlerinin ana ve en önemli parçalarından biridir, bu nedenle geliştirilmesi gereken önemli bir alan haline gelmiştir.

BMS, herhangi bir hibrit veya elektrikli aracın ayrılmaz bir parçasıdır, böylece mümkün olan en iyi performansı ve güvenilirliği sağlamak için batarya paketini izler ve korur. BMS, voltaj, sıcaklık, Şarj Durumu (SoC), Sağlık Durumu (SoH) ve şarj ve deşarj akımlarının kritik sınırlarını ölçerek pillerin ömrünü en üst düzeye çıkarmaya çalışmaktadır.

Batarya paketinin kapasitesi, şarj-deşarj ve aşırı şarj veya düşük deşarj döngülerinin yanı sıra aşırı ısınma nedeniyle kolayca değişebileceğinden, daha güvenli çalışma ve güvenilir performans sağlamak amacıyla batarya paketini denetlemek için BMS'ye ihtiyaç vardır. İşlevsel olarak, şarj ve deşarj işlemleri sırasında bir BMS, bir batarya paketine bağlı sensörlerden gelen tüm verileri (voltaj, akım, sıcaklık ve SoC) toplar ve belirli uygun eylemleri gerçekleştirmek için doğru devreyi etkinleştirir.

BMS, bataryanın düzgün çalışması ve performansı için gerekli parametreleri (örn. voltaj, akım vb.) ölçmek ve tek bir arızalı hücrenin neden olduğu olası arıza ve paket arızalarını engellemek veya en aza indirmek için batarya paketinin içindeki dahili sensörleri kullanmaktadır. Geleneksel batarya yönetim sistemlerinde batarya hücreleri, batarya algılama modülleri ve kontrol yönetim birimleri karmaşık bir kablo ağı ile birbirine bağlı olmaktadır.

Normalde her bir slave BMS tarafından sekiz ila on dört batarya hücresi değerinde sensör verisi toplanabilir. Bir dizi slave BMS, çok sayıda batarya

hücresini desteklemek amacıyla batarya hücrelerinden elde edilen verileri daha sonra işlenmek üzere depolayan bir master BMS'ye bağlanır.

Yaygın olarak kabul görüp kullanılsa da hiyerarşik bir yapıya sahip olan BMS'nin bazı önemli dezavantajları vardır. Bir BMS'nin kablolama montajı, titreşimli ortamlarda fiziksel güç kesintisi, EMI kaynaklı alanlardan kaynaklanan bağlantı arızası, batarya paketinin tasarımındaki karmaşıklık ve otomatik üretimdeki zorluklar gibi bir dizi sorunla sonuçlanabilir. Dolayısıyla, bu sorunlardan daha düşük güvenilirlik ve daha düşük üretkenlik doğabilir.

Geleneksel BMS'nin karşılaştığı yukarıda belirtilen sorunlara çözüm aramak için, kablosuz teknolojilerin sunduğu imkânlardan faydalanarak yeni yollar ve daha iyi alternatif batarya yönetim sistemleri önerilmektedir. Önerilen en umut verici kablosuz teknolojilerden biri, nispeten yeni olan ancak kablosuz teknoloji meraklıları arasında geniş kabul gören ISM bandında çalışan 2.4GHz Bluetooth Düşük Enerjidir (BLE).

Bu tez çalışmasında, ilk olarak; batarya yönetim sistemleri, özellikle kablosuz BMS'yi kapsamlı bir şekilde incelenecektir. Daha sonra, kablosuz BMS için kullanılan birkaç farklı protokol hakkında araştırma ve neden Bluetooth Düşük Enerji'nin tercih edildiği tartışılacaktır. Son olarak, Bluetooth Düşük Enerji (BLE) kullanılarak bir kablosuz batarya yönetim sistemi modeli geliştirilecektir.

Önerilen kablosuz BMS, izleme ünitesi olarak bir BQ76PL455A-Q1 entegre batarya monitörü ve koruyucusu, Bluetooth iletişimi için iki CC2640R2F mikrodenetleyici kullanılarak gerçekleştirilmiştir- bir MCU slave olarak programlanmış ve batarya modüllerinin voltaj verilerini alıp BLE üzerinden göndermek için kullanılırken, diğer MCU merkezi bir cihaz olarak işlev görmekte olup voltaj verilerini bir ana bilgisayarın ekranında görüntülemektedir.

Önerilen BMS için batarya sistemi olarak, 12 seri 2 Paralel (12S2P) konfigürasyonlu bir VDA batarya modülü kullanılmıştır. Modülün her bir hücresi 3,7 nominal gerilime ve 78 Ah kapasiteye sahipti. Toplam modül 44,4V gerilime, 156 Ah batarya kapasitesine ve 6,96 kWh enerji kapasitesine sahipti. MCU'lar

arasındaki iletişim Bluetooth Düşük Enerji protokolü üzerinden gerçekleşirken, Ana MCU ana bilgisayar ile USB üzerinden iletişim kuruyordu. BQ76PL455 kartı ve slave MCU, RS422 UART- TLL kablosu kullanarak seri UART protokolü üzerinden iletişim kuruyordu.

Bluetooth Özel İlgi Grubu (SIG) tarafından geliştirilmekte olup Texas Instruments (TI) tarafından Yazılım Geliştirme Kiti (SDK) olarak sağlanan Bluetooth Stack, iletişim protokolünü uygulayabilmeleri için mikrodenetleyicilere kod yazmak amacıyla kullanılmıştır. Kod yazma, hata ayıklama ve çalıştırma için TI'ın CCS Entegre Geliştirme Ortamı (IDE) kullanılmıştır.

Bir slave CC2640R2F MCU, BQ76PL455A-Q1 üzerinden Batarya Modülüne bağlanmış ve BQ76PL455A-Q1 GUI uygulamasından hücrelerin voltaj verilerini talep eden bir komut gönderilmiştir. Hücre voltajları paketlerini içeren batarya modülünden gelen yanıt verileri, UART RS-422 seri bağlantısı kullanılarak slave MCU tarafından okunmuştur. Ardından slave MCU voltaj verilerini Bluetooth LE üzerinden Master MCU'ya göndermiştir.

Komut, BQ76PL455A-Q1 GUI uygulamasından onaltılık biçimde gönderilmiş ve batarya modülünün voltaj verileri başarıyla okunmuş olup teyit için ana bilgisayar tarafından onaltılık biçimde görüntülenmiştir. BQ76PL455A-Q1 cihazına ve cihazdan veri iletişimi standart 8 veri biti, 1 stop biti, paritesiz evrensel asenkron alıcı ve verici (UART) formatındaydı. Veri paketindeki her veri baytı iki karakterli onaltılık sayı olarak gösterilmiştir.

Slave MCU tarafından Bluetooth LE üzerinden gönderilen voltaj verileri Master MCU tarafından başarıyla alındı. Master MCU, UART üzerinden Realterm™ seri emülatörü kullanarak verileri bir ana bilgisayarda görüntüledi. Voltaj verisi isteme komutu birkaç kez gönderilmiş ve her seferinde slave MCU ile Master MCU arasındaki veri aktarımı ve iletişim başarılı olmuştur. Verilerin başarılı bir şekilde aktarılması sistemin gerçek zamanlı ve güvenilir olduğunu göstermektedir.

Model üzerine birkaç test yaparak ve batarya modül voltaj verileri slave mikrodenetleyiciden merkezi mikrodenetleyiciden gönderildiğinde sistem sorunsuzca çalıştığını görülmüştür. Bu bağlamda düşük enerjili Bluetooth iletişim bakamından batarya yönetim sistemde önemli rol alabileceği sonucuna varılmıştır.

ACKNOWLEDGEMENTS

I wish to express my most sincere gratitude and appreciation to my supervisor Assoc. Prof. Dr Mehmet Uğraş CUMA, who has been by my side since day one of my graduate education journey. I will always appreciate his guidance and mentorship.

I would also like to present my gratitude to the members of the jury who despite their busy schedules agreed to witness the defence of my book. Without their honourable attendance, views, and suggestions this work would not be effectively completed and valued.

I am also very thankful to my parents who were always there for me whenever I need their support despite the distance. Without their presence, prayers, and encouragement I would not be able to complete this program successfully.

Also, I want to send my deepest thanks and gratitude to the Presidency for Turks Abroad and Related Communities (YTB) and the Republic of Türkiye for their support, hospitality, and sponsorship throughout my stay in this great country.

Last but not least I am grateful for my friends, roommates, and colleagues in Adana. I can't mention all their names here but many thanks go to my friend Abdirizak Samatar, Abdilatif H. Samatar and Abdihamid Adur.

CONTENTS **PAGE**

LIST OF TABLES **PAGE**

LIST OF FIGURES **PAGE**

ABBREVIATIONS

BMS	: Battery Management System
WiBMS	: Wireless Battery Management System
Li-ion	: Lithium Ion
EV	: Electric Vehicle
HEV	: Hybrid Electric Vehicle
IC	: Integrated Circuit
ISM	: Industrial Scientific and Medical band
ASIC	: Application-Specific Integrated Circuit
VDA	: Verband der Automobilindustrie (German Association of the Automotive Industry)
MCU	: Microcontroller Unit
SDK	: Software Development Kit
IDE	: Integrated Development Environment
CCS	: Code Composer Studio
BLE	: Bluetooth Low Energy
UART	: Universal Asynchronous Receiver and Transmitter
RF	: Radio Frequency
Bluetooth SIG	: Bluetooth Special Interest Group
BR	: Basic Rate
EDR	: Enhanced Data Rate
GHz	: Giga Hertz
SoC	: State of Charge
SoH	: State of Health
GPIO	: General Purpose Input Output
IC	: Integrated Circuit
ADC	: Analog Digital convertor

API	: Application Programming Interface
UUID	: Universal Unique Identifier
GAP	: Generic Access Protocol
ATT	: Attribute Protocol

1. INTRODUCTION

Worsening Global climate change crises and the increasing world temperature caused by excessive emissions of greenhouse gases across the globe call for immediate action. One of the main emission sources of those gases is the use of traditional fossil fuels as a source of energy for transportation or other daily life activities. As mentioned in research done by Bessagnet et al., (2022), even though there is huge improvement done in diesel engines in terms of efficiency and modifying it as a more environmentally friend by improving issues related to fuel combustion and exhaust gas after-treatment process, it is still one of the main contributors to air pollution. In addition to other biomass burning remains, vehicle emissions are the second most important emitter of carbonaceous Particulate Matter in Europe, specifically in urban regions.

Scientists and climate change activists are advocating for cleaner energy to tackle the alarming state of climate change before it is too late to do anything about it. Fortunately, research about clean and renewable energies as well as electric vehicles and battery management systems are in the spotlight of scientists these days. As a result, seeking an alternative source of energy caused change for cleaner and renewable energy means which also need to be carefully handled to store. For that reason, Energy Storage Systems are becoming more important day after day. The importance of energy storage systems and batteries calls for extensive security measures and management. Hence Battery Management Systems became one of the most important research areas lately.

Batteries are used in almost all electronic devices used for daily life purposes. But depending on the size of the battery, its use cases and handling scenarios change. Lately, vehicles that operate by using battery energy are on the increase. There is a massive increase in the global sales of passenger plug-in electric cars which shows the promising future in this field. In May 2022 alone, about 699,708 new passenger plug-in electric cars were registered according to

EV-Volumes data. That is a 55% increase comparing the previous year and the market share of EVs counts about 12% of which 8.6% of it is for all-electric cars.

To simply some terminologies before we go further, let's define some basic elements of BM.

- Cell: is the minimum basic constituent of battery (provides 3-4 V in Li-ion batteries).
- Battery: a collection of cells connected and wired together in series to become a single physical module in order to provide higher voltage
- Pack: Number of batteries connected either in serial or parallel.

As mentioned above, for safety and functional reasons batteries are one of the most important parts of the vehicle. Using and maintaining batteries (especially high voltage ones) needs extra maintenance and care. That is where Battery Management System comes into work. The rapid growth of battery packs used in EVs and HEVs made battery management systems an important area for improvement. To assure and improve battery pack reliability and effectiveness across different styles and use cases, passenger Electric Vehicles rely greatly on BMS as the use of electrical drivetrain source in these vehicles increases (Shell et al., 2015).

A BMS is an integral part of any hybrid or electric vehicle that manages, monitors, and safeguards the battery packs, to ensure the best possible performance and dependability. By measuring voltage, temperature, SoC, SoH, and the critical limits of charging and discharging currents, the BMS seeks to maximize the lifespan of batteries. There are four primary estimating methods in a BMS: battery fault, SoC, SoH, and battery life (Cuma & Koroglu, 2015).

BMS is needed to supervise the battery pack to provide safer operation and reliable performance as the capacity of the battery back can easily get changed by the charging-discharging and overcharging or under discharging cycles as well as

overheating. By functionality, during charging and discharging operations, a BMS gathers all data from the sensors (voltage, current, temperature, and SOC) connected to a battery pack and activates the correct circuitry to execute a certain action (Rahman et al., 2017).

The BMS uses internal sensors inside the battery pack to measure necessary parameters (i.e., voltage, current etc.) for smooth functioning and performance of the battery as well as blocking or minimizing any possible fault and pack failures caused by a single malfunctioning cell. The BMS hardware, however, could be expensive due to the several elements it contains. In addition to that, it is sensitive to mechanical malfunctions and failures. The battery pack's weight, size, and location demand for placement in a distributed architecture which makes it more vulnerable to vibrational disturbances that could lead to failure by cable terminations and energy spikes across loose wires (Shell et al., 2015).

Battery cells and battery sensing modules, or slave BMSs, are connected by a complicated wire network in traditional battery management systems. Eight to fourteen battery cells' worth of sensory data can normally be collected by each slave BMS. A collection of slave BMSs are linked to a master BMS, which stores the acquired data from the battery cells for later processing, in order to support a high number of battery cells. Despite being widely accepted and utilized, the hierarchical BMS architecture has some significant drawbacks. The wiring assembly of a BMS can result in a number of problems, including physical power interruption in vibratory environments, connection failure from EMI-induced areas, complexity in the design of the battery pack, and challenges in automated manufacturing. Lower reliability and lower productivity could also be inherited from these issues. In addition to that, a considerable amount of the price of the battery pack is made up of the cost of the wiring assembly, connection cables, protective cases, dis-connectors, and manual work. A regular car or any other high voltage application that needs energy storage uses between a few hundred and a

few thousand battery cells, and as the quantity of battery cells rises, the above-mentioned issues become worse (Lee et al., 2014).

1.1. The Aim of the Study

The main aim of this study is to make a smart and robust battery management system using a new technology called Bluetooth Low Energy. BLE replaces the physical communication wires between the Battery Control Unit (BCU) and Module Supervision Circuits (MSC) of conventional BMS. So, the working principle of this new system is to process and send the data measured and sensed by Cell Sensor Circuits (CSC) from MSU to BCU in a wireless media using the ISM band of Bluetooth. Then BCU takes the relative action based on the information gotten from the MSU and sends the feedback back to it.

Considering the importance of battery management systems in electric vehicles for safety and economy, this thesis work will look at the development of a wireless battery management system prototype using Bluetooth Low Energy technology.

The outline of the next parts of this thesis work will be as follows:

In the literature review, some background history about battery management systems and different schematics used in it will be discussed. Several wireless technology systems employed by WiBMS and each one's advantages and disadvantaged will be added.

In the Materials and methods section, Bluetooth Low Energy will be explained and the reason that made it the most significant technology for WiBMS will be mentioned. In addition to that, the hardware and software used for this project will be further elaborated.

In the Discussion part, the block structure of the project will be shown. How the connections were being set up and the data communication link configurations will be discussed.

In the last section, project shortcomings and some recommendations about regrading wireless Battery Management System technology will be given.

2. THEORETICAL BACKGROUND AND LITERATURE REVIEW

In this chapter, an inclusive literature review about Battery Management System technology will be covered. Different types of communications used inside the BMS will be clearly explained. In addition to that, advantages, and strong sides, as well as limitations and drawbacks faced by previously developed BMSs, will be mentioned.

The increasing acceptance of batteries – particularly Li-ion batteries – as a key energy storage device in important industry sectors like automotive and manufacturing and the development of renewable energy-based mobile technologies such as EVs and HEVs have caused a mass transition from fossil fuel-dependent practices to electric power machinery and consequently redefined the world of energy storage. Because of batteries' importance as a source of energy, to ensure that the overall electrical system meets high standards of safety, reliability, and quality, regardless of whether it is used for stationary energy storage or electric transportation, functional safety considerations, or those relating to automatic protection in battery management for battery pack technologies are particularly crucial (See et al., 2022).

In addition to that, in terms of commercial-scale production of batteries and their services and management calls for high quality, high safety standards and performance so that they can supply the unprecedented demand for battery applications. Although research about this area is heavily invested in and some edge-cutting developments have been already made, there are still some methodology and theoretical foundation issues and challenges that need to be systematically addressed(Li et al., 2016).

Series connected batteries have been used for high-voltage and high-power applications such as EVs and renewable energy storage devices. According to Li et al., (2016), Kukut et al., (1995) states that because of external factors including manufacturing, degradation with age and environmental variance and internal

factors like charging and discharging imbalance, internal impedance, and differences in thermal conditions as well as self-charging rate, the stored charge or energy in different battery cells becomes unequal. As a result of this mismatch the batteries' lifetime, performance and capacity are greatly reduced. There could also be chances of greater hazards such as an explosion or fire. To reduce that mismatch and remove the imbalance between battery cells battery equalizers which are a very important part of the BMS are being developed and thus the overall battery performance is improved.

Different equalizing techniques have been used for initial BMSs, then distributed BMS is developed which comes with advantages such as easy reconfiguration and installation, efficiency in terms of equalizing and speed as well as improvement in protection and reliability compared to the traditional BMS architecture. This distributed BMS architecture could be further embedded together by integrating batteries and power electronics as one module and a localized management system can be integrated. In that way, issues such as communication, control and monitoring can be easily addressed.

In this distributed BMS proposed by Li et al., (2016) communication between the local controllers and the global controller is carried out by a bidirectional communication bus. Each cell's specific information is gathered by the local controller, sends it to the global controller using a serial communication bus (such as CAN, RS232 etc.) and in its turn; the global controller responds with commands that would adjust the status of each module. The authors suggested that for a possible elimination of serial lines, wireless technologies such as Wi-Fi and Bluetooth could be used as a means of communication between the controllers. A software algorithm is designed to manage the local and global controllers. The algorithm determines optimal system requirements and cell characteristics and based on that the global controller sends commands containing required power, voltage, current and other necessary information to the local controller in order to maintain that optimal state. In return, the Local controller updates the global

controller about the condition of the battery cells and ensures that they are operating in optimal condition by controlling the converter input/output voltage and current, and by continuously monitoring battery cells.

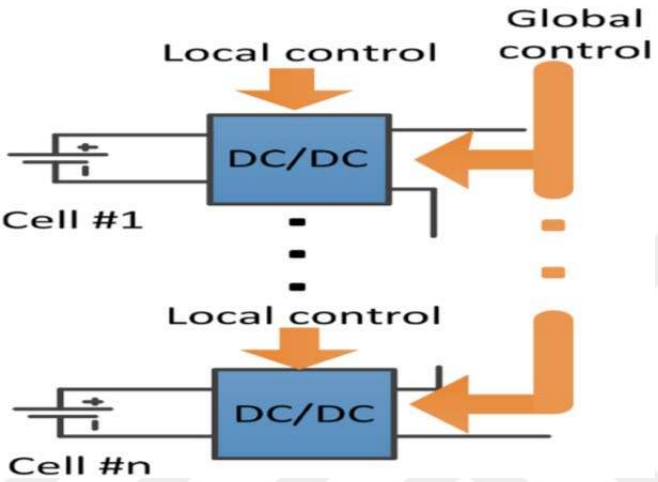

Figure 2.1. Distributed Battery Management System (Li et al., 2016)

Li-ion batteries require extreme care to operate reliably over a long time. To realize that goal, the BMS needs to do essential tasks of measuring and controlling the SoC and SoH of the individual cells as well as balancing charges and preventing faults. Battery monitor IC is the main element in the battery management system, and it accurately carries out the process of measuring the voltage, current and temperature of each individual cell and then sends the data to the control circuit. The data is then used by a controller for the purpose of computing the state of charge and state of health of the battery pack. Based on the processed data, the controller may also send commands for charging or discharging certain cells to the battery monitor so that the battery pack can stay in a balanced state(Zimmer, 2012).

A number of Li-ion cells connected in series and grouped together are called battery modules. To get the desired voltage or current, these battery modules

are connected in series or parallel form and become the battery pack. In a typical hybrid car's chassis, several 8 to 12 battery modules lie with 10 to 12 cells for each. The communication for this kind of commercial BMS is transmitted through standard protocols like CAN-bus and for a large number of modules that has many numbers of communication nodes, there will be a need for interconnection which will require a complex and large number of extra wirings. That complexity can eventually lead to increased cost, weight, and difficulties in production in addition to issues related to the galvanic isolations of the cells and limitations for the need to change faulted individual modules. For these reasons and others, it is inevitable to seek and develop other alternative and innovative means of communication that can efficiently control and monitor the battery pack.

Wireless data transmission seems to be the best approach for this issue as it offers alterative solutions. Some studies done in the WiBMS field used protocols such as IEEE 802.15.4 with a maximum data rate of 250 kbps, but they did not study the properties of the wireless channel inside the battery pack. Also, the data rate seems to be unsuitable to use in large battery packs as Alonso et al., (2014) reported. In a feasibility study done by the same authors, a WiBMS using ISM-band-based planar, helix and chip antennas were proposed. The idea was to exchange information between the Battery Control Unit (BCU) which computes and determines the values of the SoC and SoH and the Cell Sensor Unit (CSU) which measures parameters like voltage and temperature of an individual cell. The authors used the battery pack interior as a wireless channel and mounted antennas on each cell and used two emulators of different sizes to measure and analyze the transmission capabilities and frequency ranges of different antenna types inside a battery pack. As a result, the authors suggested that while low-frequency antennas are less position dependent inside one emulator, they can be susceptible to distortions and could be costly, but a communication based on them could be possible. On the other hand, higher frequency antennas (800 MHz and above) are smaller in size so that they could be printed on planar antennas and consequently,

they are economical as they can be PCB extensions and hence no extra wire will be needed. Further extensive research on antennas operating in ranges above 2 GHz that are planar printed is suggested.

Lee et al., (2014) developed a wireless BMS using proprietary Wireless Battery Area Network (WiBaAN™) protocol. The system is a distributed star topology in which the master BMS uses a single chip for battery management with a transceiver operating in the frequency band between 810 MHz – 990 MHz of the ISM band to communicate wirelessly with each slave BMS on the battery. The slave BMS modules also have a Battery Sensing ASIC chip package that has two ICs that include a transceiver with other embedded sensors to measure important parameters of each cell and send it to the master. The proposed system has many advantages in terms of productivity, reliability, and reduction of cost as it removes wire harnesses, connectors, and protective case problems to mention some, but the protocol used for wireless communication is not well explained and also its bandwidth is limited to 1MB.

In an experiment carried out by Rahman et al., (2017), ZigBee communication protocol was used to transfer the data measured by several sensor node and processed by microcontrollers to user-end controlled display devices for monitoring the BMS. In this WiBMS the design consists of a BMS part and a monitoring/user end controlling unit parts that communicate through ZigBee wireless devices. ZigBee is chosen as it has preferable features such as low power consumption, reliability, cheap cost, etc. while this WiBMS showed a good work in battery balancing, the same would not be true for battery temperature efficient management.

Battery management with all its parts can be challenging and difficult because the number of battery cells and corresponding weight required to power a vehicle generally demands a battery pack that is distributed over a significant portion of the undercarriage or other similar space. So, every battery cell should be managed separately so that the battery pack's group behavior can be adjusted to

ensure maximum performance while avoiding premature pack failure due to individual cell failure. For that reason, a BMS that relies on the information from every cell is necessary and to accomplish that battery pack with cells that each one has its sensor wire are constructed.

However, these battery cells can be hundreds or thousands in number depending on the type of the cell or size of the battery, giving Tesla Motors for instance uses 6800 lithium-ion cells for their automobiles. Because of the large number of cells in a pack, tremendous number of wires would be needed which also require terminations at their ends for safety. Typically, sensor wires are altogether routed to one location in which the management system for collecting and analyzing battery information would be located. This calls for wires to be bundled and terminated at one or more connectors which may require using high-density high-pin count connectors.

As Fig 2.2 shows, in a modern electric vehicle propulsion system, the connectors alone account for 8% of the cost of the electrical components. This number of wires would be costly to construct as well as maintain, also they would be the source of pack failures from recurring disconnections in sensor wire due to vibration of vehicles or other external sources. To solve these problems, wireless communication system is great viable option (Shell et al., 2015)

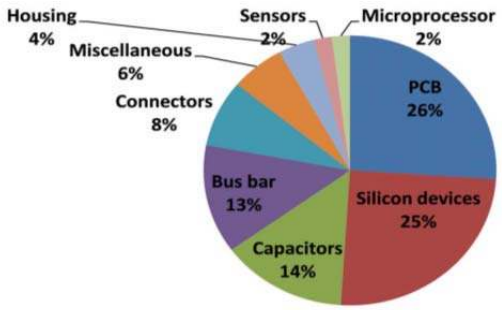

Figure 2.2. Cost of power electronic components in modern EV's propulsion system (Whaling, 2014)

Vishnu et al., (2020) predict that Bluetooth will become more popular in vehicle industries compared to other wireless technologies. The authors conducted research in which they compared use cases of different wireless technologies (including Zigbee, IoT, Bluetooth and Cloud based technologies) in electric vehicles. They got to a conclusion of that Bluetooth and Zigbee are the most suitable ones for use as they have good qualities such as low power consumption. However, Bluetooth has more bandwidth than Zigbee and it can be preferable than Zigbee in some cases. The table below shows a comparison of the payload of different wireless technologies used in BMS (Bansal & Pr, 2019).

Table 2.1. Raw data rate vs payload for wireless protocols

Communication Protocol	**Raw Data rate**	**Payload throughput**
Zigbee	250 Kbps	200 Kbps
BLE5.0	1 Mbps	305 Kbps
Wi-Fi(IEEE 802.11)	11 Mbps	6 Mbps
Wi-Fi HaLow	720 Kbps	650 Kbps
Near Field	424 Kbps	106 Kbps

Among all candidate wireless technologies, Bluetooth seems to be the most fit for communication interface of BMS. In the next section of the book, why Bluetooth has been chosen will be elaborated. In addition to that, materials used and how the experiment was conducted will be explained.

3. MATERIALS AND SYSTEM-LEVEL DESIGN METHODOLOGY

In this chapter, system design and the materials used will be explained. Details of the software and hardware that were part of the project will be given. Moreover, how the system was developed and the reasons behind chosen devices will be mentioned. The chapter will give deep explanation about how the project has been done and what kind of process has been followed.

As mentioned previously, the main aim of the project was to develop a Bluetooth based interface that would be integrated to the BMS to use it as a wireless communication system instead of using old fashioned wires that their shortcomings have been already discussed in previous sections. Although that wireless interface would be very realistic to do, due to lack of some hardware, we tested the Bluetooth based system along with the old system by connecting it across using RS 422 interface.

The system was developed by using the equipment that will be shown and detailed in the coming sections. The following parts of the book will explain all devices used as well as software and how the system was developed.

The proposed system consists of several devices including BQ76PL455A-Q1 integrated battery monitor and protector as a monitoring unit, CC2640R2F microcontroller for Bluetooth communication, 12S2P configured VDA battery module and RS-422 for UART to TLL communication. To display the measured data Realterm application was used as emulator. For code development environment TI's Code Composer Studio was used. The following sections will explain each item in a detailed form.

3.1. BQ76PL455A-Q1 Integrated Battery Monitor

BQ76PL455A-Q1 is an ASIC that is designed to monitor and protect 16 cell battery applications with high reliability.

An integrated high-speed communication interface with capacitor isolation allows up to 16 devices of BQ76PL455 to communicate with a single host via a high speed UART. The below picture is showing how the two devices are connected and the left one is connecting all of them to the host microcontroller.

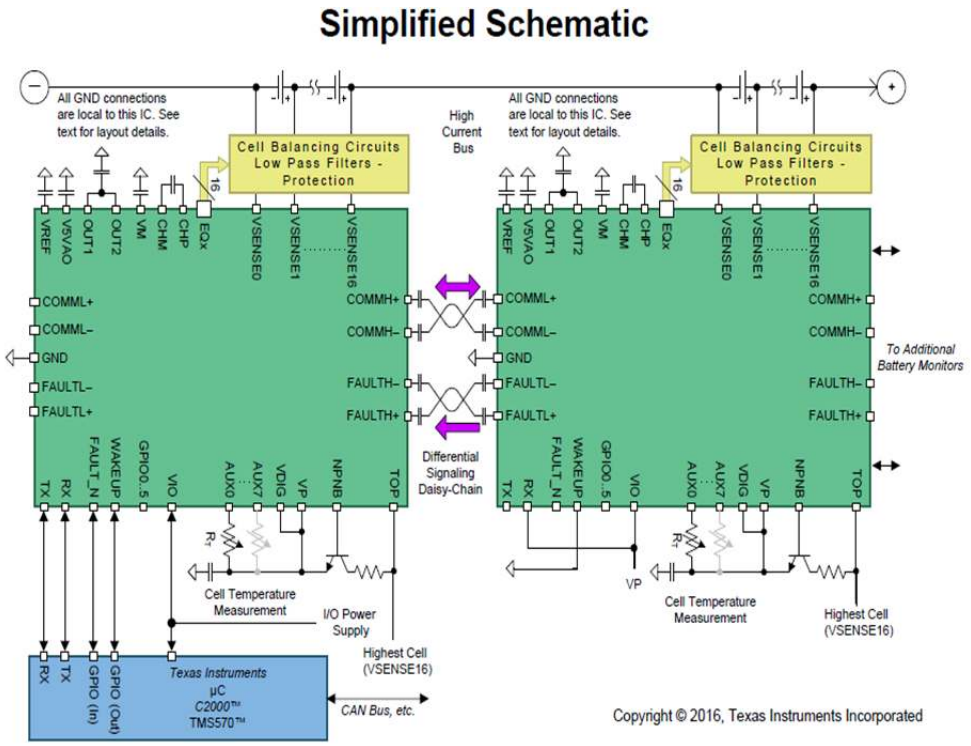

Figure 3.1. simplified schematic of BQ76PL455 device

A number of different fault conditions, such as over-voltage, under-voltage, over-temperature, and communication faults are monitored and detected by the BQ76PL455A-Q1. Eight analog AUX ADC inputs and six General Purpose Input Output (GPIO) connections are provided for additional monitoring and programming capability. For further security, a secondary thermal shutdown is also included in it.

The high-speed differential communications interface, which has been assessed for compliance with Bulk Current Injection (BCI) standards, enables the serial connection of up to 16 BQ76PL455A-Q1 devices. Effective common-mode noise rejection is accomplished by this capacitor-isolated communications link. Through a fast UART interface, the BQ76PL455A-Q1 connects with the host.

In the event of more than one BQ76PL455A-Q1 device, they are connected to each other in series, and they communicate in a stacked daisy-chain communication configuration. In the stacked design, the primary microcontroller initially uses the UART communications interface to connect to a BQ76PL455A-Q1 device. Then, utilizing a proprietary differential communications protocol, communication is broadcasted up the chain of linked slave BQ76PL455A-Q1 devices through AC-coupled differential links connected by the COMMH+/- and COMML+/- pins.

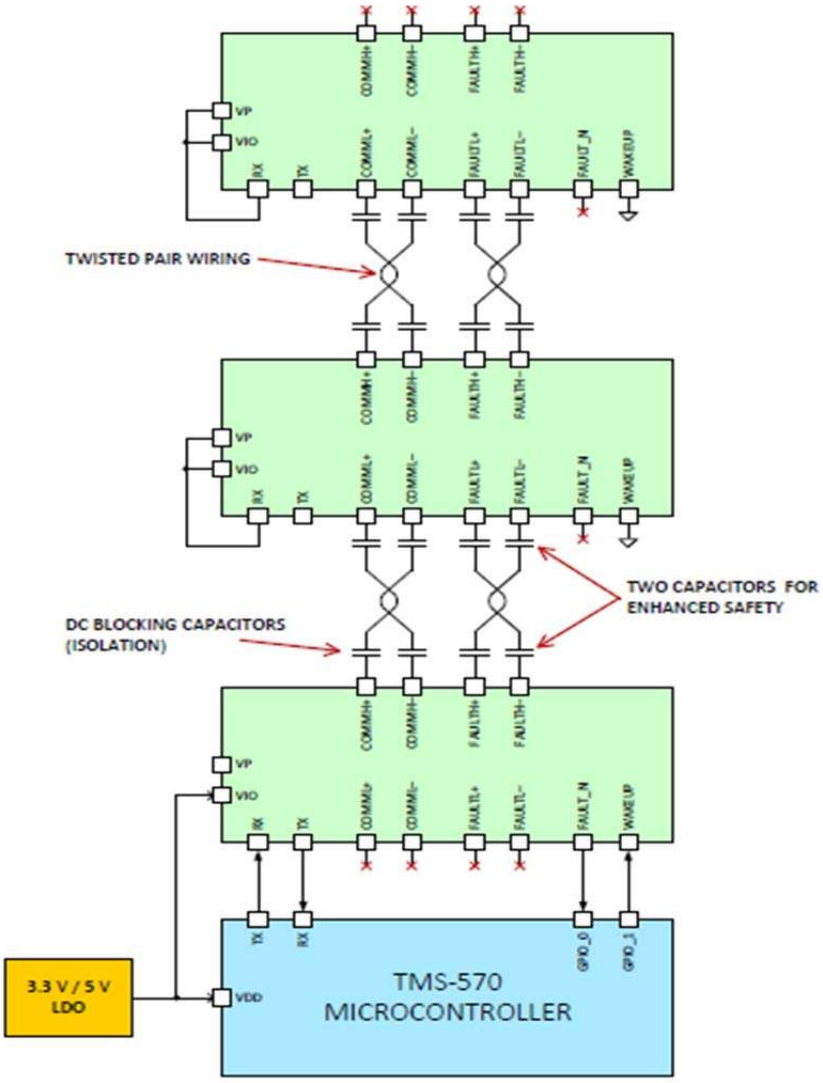

Figure 1.2. Simplified stack configuration communication system (Texas Instruments, 2015)

The base device (i.e., the one connected to the host microcontroller) can receive or send data by using UART at different baud rates between 125 kb/s to 1Mb/s while the communication between BQ76PL455 devices through the VBUS daisy chain occurs at a fixed data rate of 4 Mb/s as the propriety asynchronous protocol requires. The bottom device retransmits from/ to host system's single-

ended interface at the baud rate selected for that interface. All other BQ76PL455A-QA devices must be at the same baud rate to avoid timing misalignment.

The following figure shows the BQ76PL455A-QA in an evaluation module used I this specific project.

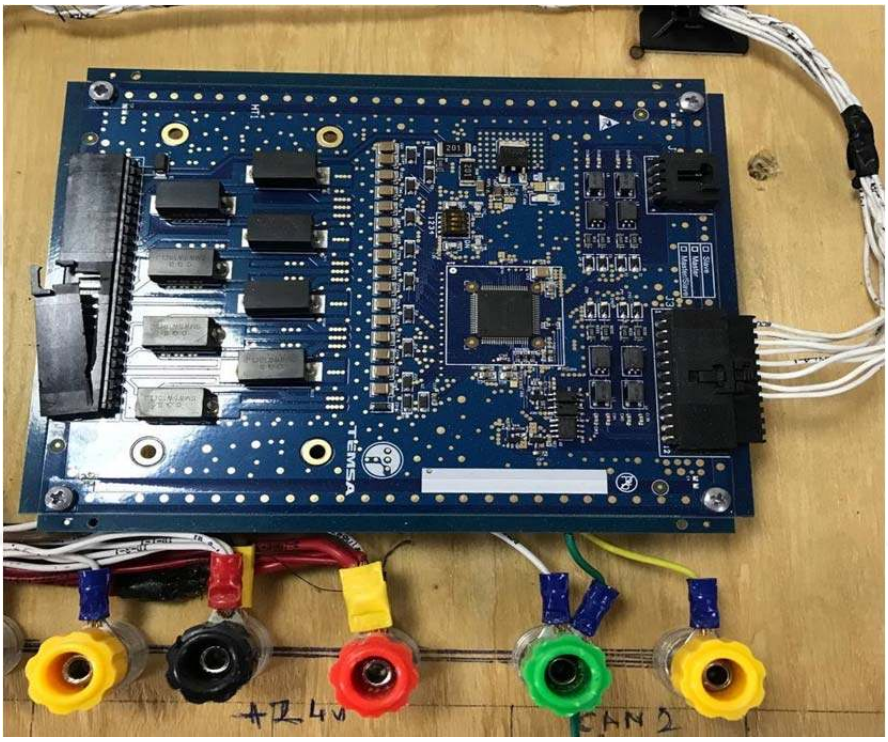

Figure 3.3. BQ76PL455A-QA Evaluation Module

3.2. CC2640R2 LaunchPad and MCU

This microcontroller is one of the Bluetooth Low Energy enabled microcontrollers that has the most outstanding capabilities for a variety of use cases, including medical, sports as well as automotive applications.

The SimpleLink™ BLE CC2640R2F wireless microcontroller (MCU) is one of the smallest Bluetooth low energy solutions with strong radiofrequency (RF) performance targeted for Internet of Things (IoT) end nodes. Since it can be

readily added to an existing host MCU as a network processor to provide system design flexibility, the CC2640R2F device is also excellent for industrial applications.

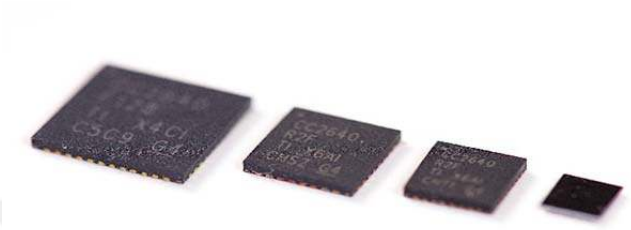

Figure 3.4. C2640R2F MCU (TI E2E Support Forums, 2017)

A SimpleLink is a broad portfolio of wired and wireless solution ARM Cortex -M enabled microcontrollers that has effective features such as Low power, up-to-date security, easily installable to previous systems and a common Software Development Kit (SDK) that offers more memory and processing power and saving time. Supported wireless connectivity standards include Bluetooth Low Energy plus others and the connectivity stack runs on an optimized host MCU for SimpleLink wireless network processors.

To simplify the development and prototyping process, a LaunchPad development kits are available. The LaunchPad offers productive features like integrated on-board emulator to simplify programming and debugging process. It also offers a plug-in module header that can be connected to displays LED driver and other hardware resources. Moreover, the hardware design is available as an open-source and a custom hardware can be developed by using LaunchPad kit as a reference design.

As a main processor the device is enabled with a 32-bit ARM® Cortex®-M3 processor that is running at 48 MHz. In this system core (CM3), BLE layers

from user application to the higher layers of the protocol stack runs on it. A second processor, the Cortex M0 (CM0) core is used for radio hardware interfacing, also, it translates the complex instructions from CM3 core into bits of data that can be sent over the radio. CM0 operates the PHY layer of the BLE protocol. BLE apps on iOS (e.g., LightBlue) and Android (Like BLE Scanner) as well as other Bluetooth enabled apps by TI can be connected to the MCU device to read its buttons, control its LEDs and other I/O pins and signals.

As a power source for the device, a USB-compliant power source, like a USB charger or a computer can be used as well as other means of external sources. Low Dropout LDO regulator powered from the USB VBUS supply supplies 3.3V to the XDS debugger, the CC2640R2, and other associated circuitry including 3V3-marked pins. Additionally, the device is designed to operate in temperature range between -25 to +70 C which makes it ideal for our use case.

Other worth mentioning features of the device include 275KB of nonvolatile memory with 128KB of in-system programmable flash memory, SRAM of up to 28KB and efficiently written code size architecture by placing drivers, RTOS and Bluetooth stack software in ROM to get more Flash available for the application.

The peripherals that the device presents include UART, I2C, I2S among others while all digital peripherals can be routed to any GPIO. It also offers four general-purpose timer modules, 12-bit ADCs as well as other important peripherals. Figure 3.4 shows different peripherals and features found in the MCU.

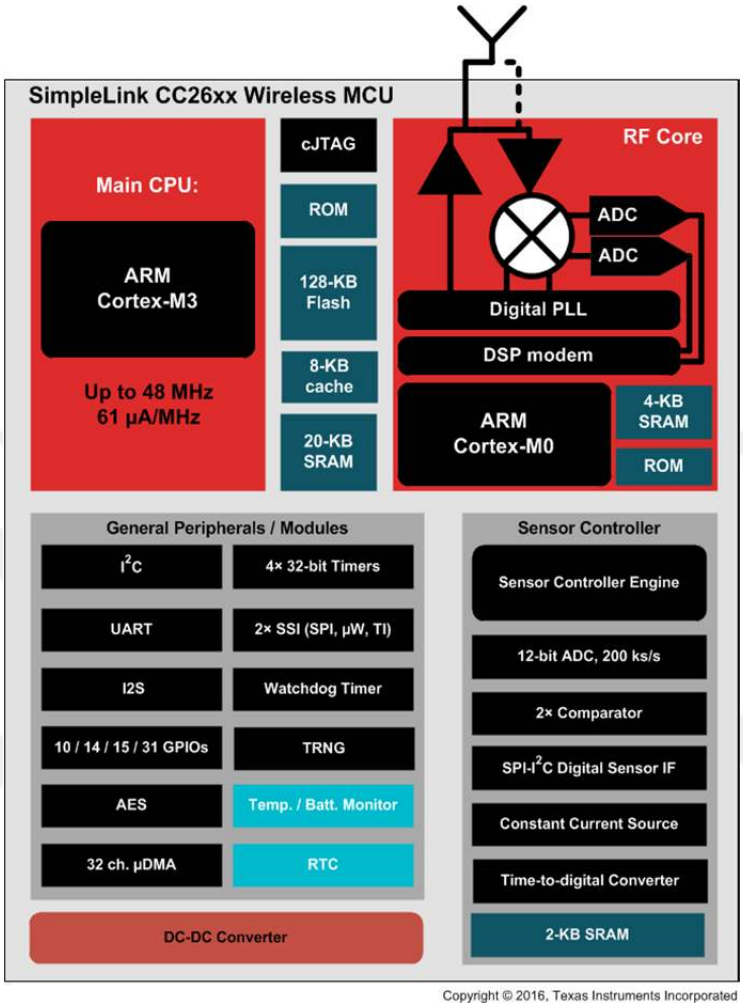

Figure 3.5. CC2640R2f architecture block diagram (Texas Instruments Incorporated, 2020)

An SDK that comes with the LaunchPad simplifies the process of code development and prototyping as it contains set of development tools that can be used for application development on the microcontroller. The SDK components include Hardware Abstraction Layer (HAL) which is a layer of drivers and OS kernel containing a set of C functions that abstract writes to registers to access hardware features.

There is also OS/Kernel in the SDK that provides timing and scheduling services for tasks. Driver APIs such as the one used in this project (UART) are also available in the SDK.

To implement projects on the MCU, stack project which contains Bluetooth protocol specific APIs and Application project must be imported from the SDK to the development environment and hence the project would be created.

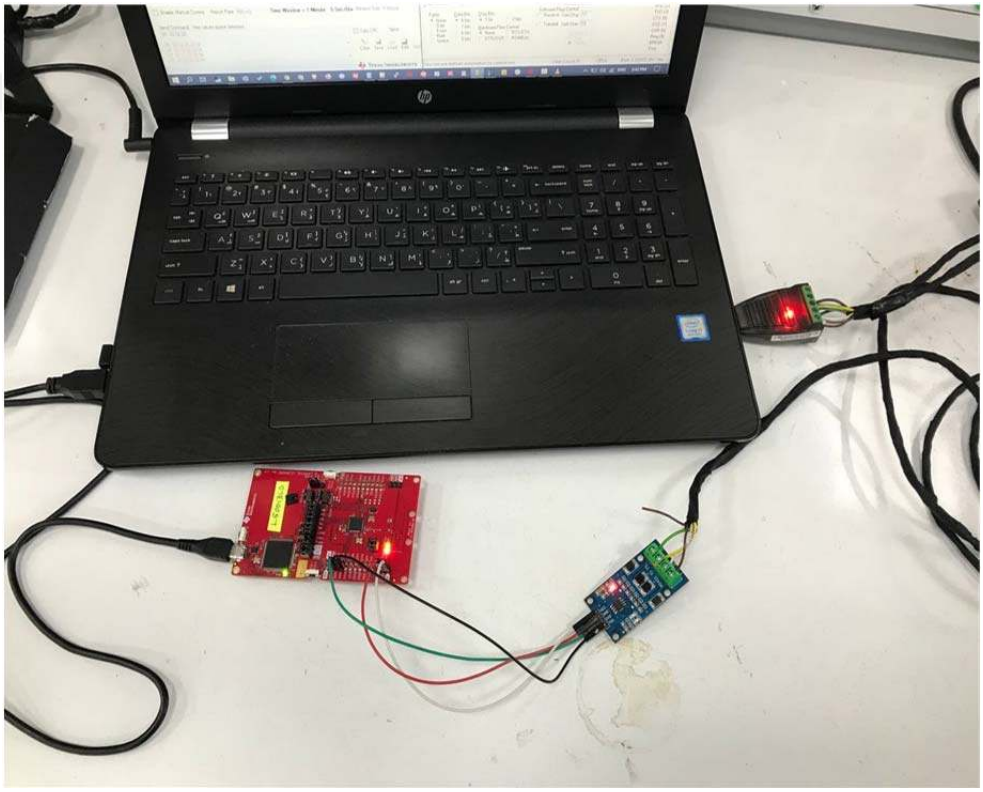

Figure 3.6. Slave CC2640R2F MCU connected to the attery module through RS 422 to TLL converter.

3.3. Software Development Environment

As a software development environment Code Composer Studio IDE was used. Code Composer Studio or CCS is developed by TI, and it is very simple code development environment. In this specific project CCS version 10.4.0 is used.

Code Composer Studio™ software is (IDE) that implements TI's microcontroller (MCU) and embedded processor coding and development tasks. CCS software contains set of tools for developing and debugging embedded applications. The software which has brilliant features such as an optimizing C/C++ compiler, built-in source code editor, environment for project building among others, offers an intuitive IDE that provides friendly single-user interface with simple step by step application development flow.

The following figure shows a snippet of the interface of Code Composer Studio and different sections it displays on the screen.

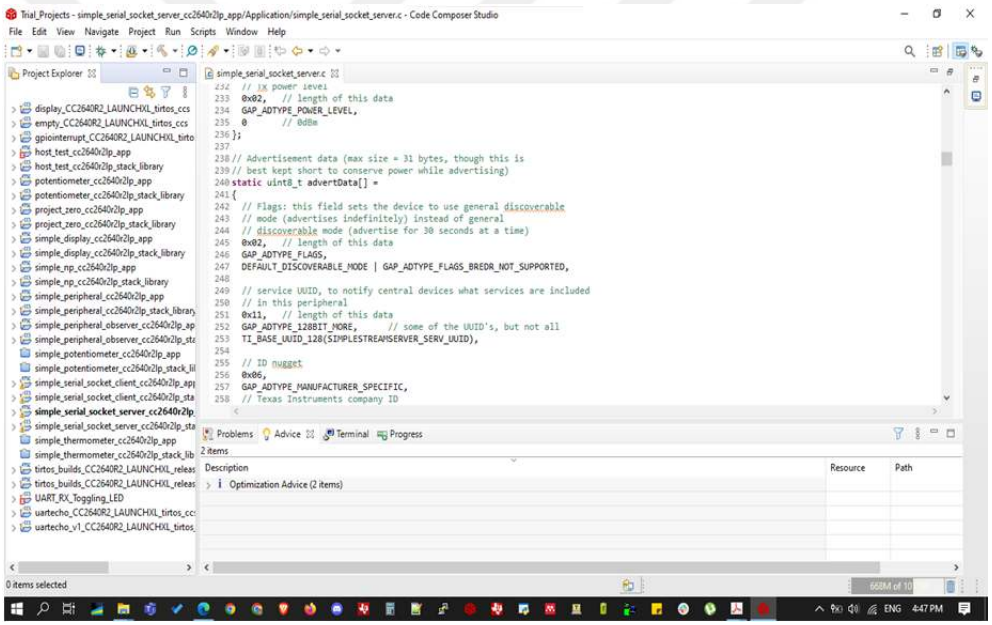

Figure 3.7. Display of CCS IDE interface

3.4. Bluetooth Low Energy

Bluetooth Low Energy or BLE is a relatively new technology developed by SIG started as part of the Bluetooth 4.0 Core Specification. It was developed to function as a smaller, highly optimized version of its predecessor, classic

Bluetooth, but in reality, BLE comes with an entirely different lineage and design goals (Townsend et al., 2014)

Starting from the Bluetooth Specification 4.0 and above, there are two wireless technologies that has been defined, namely BR/EDR (classic Bluetooth) and BLE (Bluetooth Low Energy). The BLE system was developed to transmit small packets of data while consuming significantly less power than BR/EDR devices.

A complete understanding of Bluetooth LE calls for close familiarity with its specifications. Bluetooth LE's architecture, operations, and protocols are fully described under a single essential document known as the Bluetooth Core Specification. **The Bluetooth Core Specification** which is also the primary specification for Bluetooth Classic defines the way Bluetooth technology works as well as the steps to be followed by developers when implementing a Bluetooth stack or its other features. Another collection of specifications containing two special type specifications called profiles and services covers the interoperability of products over Bluetooth LE. Figure 3.5 gives more details about the constituents of the specification

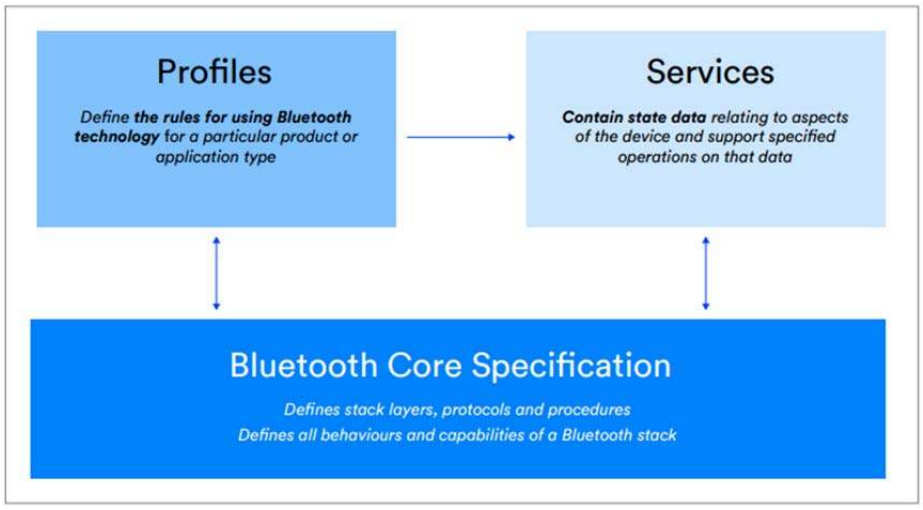

Figure 3.8. Bluetooth LE Core Specifications (Woolley, 2022)

As usually client / server relationship is formed between two BLE devices that are communicating over a link, a **profile specification** defines the roles they assume, and particularly it defines the client's behavior and the data on the server that it will work with. It covers what interactions are possible with Bluetooth LE, which services are required to achieve certain goals, and how value reading and writing should be presented and sequenced to accomplish those goals.

On the other hand, the data on the server are contained in items known as *characteristics* and *descriptors*. Characteristics and descriptors are then grouped into structures known as *services* as they provide a context within which to assign meaning and behaviors to the characteristics and descriptors that they contain. The **service specification** functions to implement the definition of a single service along with the characteristics and descriptors that it contains. It also defines the behavior that the service hosting device is expected to show in response to different conditions and state data values (Woolley, 2022).

There are roles and responsibilities defined by what is known as Generic Access Profile (GAP) that BLE device applies when interacting with other BLE device. Two roles are being defined under GAP: Central vs Peripheral.

In Peripheral role the device gets the values or data through sensors or other means, and it makes advertisement so that a central can connect to it and read from it. On the other hand, the device that is scanning the advertisement sent by peripheral device and upon connection reads/writes a data to/from the peripheral is in the central role.

Generic Attribute (GATT) protocol determines how the actual communication between two or more BLE devices happen after they make a connection. GATT profile is the roadmap that specifies how to send and receive short pieces of data called "*attributes*" over BLE connection. After connection establishment, devices will share GATT metadata with each other and depending on what kind of a data they want to send, they will have two different GATT roles. A device that has a data to send (i.e., report sensor data etc.) will act as GATT

server. And the device that is reading data (i.e., receive update from a sensor etc.) will be the GATT client. Usually the device with abundant resources (like Mobile phones) acts as a GATT client device in BLE.

The ATT layer enables one device to expose specific pieces of data called attributes to another. The Generic Attribute Profile (GATT) layer is a service framework that defines the ATT sub-procedures. GATT sub-procedures (attributes) handle data exchanges between two devices in a Bluetooth Low Energy connection. GATT is directly used and interacted by the application and/or profiles. GATT layer also defines characteristics and attributes.

While sometimes In Bluetooth LE, characteristics and attributes are interchangeably used they are different from each other. Characteristics can be considered as groups of information called attributes while the actual information that is transferred between devices are attributes. Attributes are organized as data values, properties and configuration information and used by characteristics.

A typical characteristic contains the following attributes which are stored in attribute table in the GATT server: a characteristic value (its data value), a declaration (descriptor containing properties, type etc. of characteristic value), Client Characteristic Configuration (configuration enabling the GATT server to configure the characteristics to notify or indicate when sending messages), and characteristic user description (ASCII string that describes the characteristic). Each of the above attributes has Handle (unique number showing its index of the table of attributes), Type (indicator of attribute data called UUID -Universal Unique Identifier), and Permissions (showing how the client device is expected to access the value of an attribute).

Finally, a collection of characteristics creates a GATT service. For example, in this project Simple Serial Socket service is used and it contains a DataIn and a DataOut characteristics. Multiple services grouped together can form a profile. Many profiles could only implement one single service, so the two terms could be sometimes interchangeably used.

3.5. VDA Battery Module

One module of VDA battery modules was used in the project. VDA stands for Verband der Automobilindustrie (German Association of the Automotive Industry) and it is a German interest group of the German automobile industry that has proposed standards for the size of the battery cell technology used in automotive industry.

The VDA battery module (see figure below) used is 12S2P (12 series 2 parallel) configured with each cell having 3.7 nominal voltage and 78 Ah capacity. The overall module has a voltage of 44.4V, battery capacity of 156 Ah and energy capacity of 6.96 kWh.

Figure 3.9. VDA Battery module

3.6 System Design Implementation

The project is implemented upon previously used conventional wire battery management system using previously explained hardware devices. The only thing changed was the addition of CC2640R2F to the system to read data along with BQ76PL455A board, so that it can send the measured data to the other MCU or Mobile phone application over Bluetooth LE link.

BQ76PL455A board which was directly connected to the VDA battery module for battery monitoring and maintenance was configured and managed through BQ76PL455A-Q1 GUI application provided by Texas Instruments. The

application interface displays information about battery cells voltages and their status such as fault conditions. It also has poll start and stop button where you can read battery voltages in real time. It has a comms section where UART serial connection parameters can be configured to connect it to the host computer. For this project, the communication between the BQ76PL455 board and the host PC happened over ECON-485 USB serial cable with a configuration parameter of 125 K baud rate 8-bit data, 1 stop bit, no parity, no flow control. Then, a CC2640r2f MCU was connected to the ECON- 485 cable in a cross-link configuration using RS422 interface on a Single-chip MAX490 to TTL converter module (see figure 3.7).

Figure 3.10. ECON-485 USB and RS422 to TLL converter Module

The overall connection was done as the figure 3.8 shows. The ECON- 485 was connected to the host PC while RS422 cable was also cross connected to it. The connection between the host PC and BQ76PL455 was full duplex with speed of 125 Kbps. The Rx+ and Rx- of RS422 cable was connected to the Rx+ ad Rx- of Econ RS485 cable respectively. The receiver pins were also the same configuration. The TTL side of the MAX490 to TTL converter module was connected to the CC2640R2F MCU, which is also connected to the host PC by Bluetooth to display data through Realterm Serial Emulator Program. The MCU was sending data by Bluetooth LE to either master CC2640R2F MCU or mobile phone application installed in the iOS phone that was called BlueLight Bluetooth LE Application.

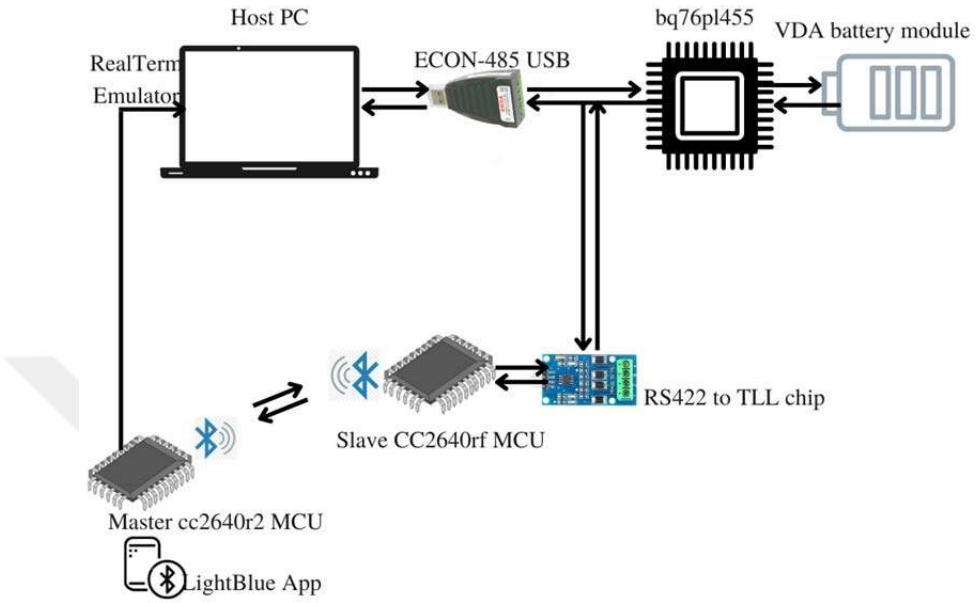

Figure 3.11. System block design

3.7 Software Development

As mentioned above, the code development part of the project was done by using TI's CCS as a development environment and its Software Development Kit as a reference for projects. SDK was preferred for convenience because it contains Bluetooth stacks and almost all necessary driver APIs needed for BLE project to run on the MCU.

Also, for the MCU to run different tasks such as Bluetooth tasks and other data processing tasks, a kind of operating system must be used. For multitasking and time and resource constrained tasks as the case is in the embedded systems, Real-time Operating System (RTOS) is used. Fortunately, TI provides device specific RTOS that already comes with all necessary tools and library of services.

TI-RTOS kernel provides library of services that helps with simplification of application design and maintenance. TI-RTOS also has scheduler which

executes threads according to their priorities, clock system, synchronization, real-time analysis, and memory management and driver tools.

As mentioned in the previous section, TI provides SDK that comes with different libraries and documents intended to use as a starting point for application development. The SDK is a complete software platform that can be used for developing single-mode Bluetooth Low Energy applications intended to run on the CC2640R2F MCU used in this project.

The SDK contains solution platform of the following components:

- TI's Real-Time Operating System (TI-RTOS) with the TI-RTOS kernel, optimized power management support, and peripheral drivers (SPI, UART, and others)
- CC26xxware DriverLib which provides a layer of register that is used by software and drivers to control the CC2640R2 System on Chip (SoC)
- The Bluetooth LE protocol stack that is provided in a library form with parts of the protocol stack in the CC2640R2 ROM.
- Sample applications and profiles intended to use as a reference for application development using both proprietary and generic solutions in easier way.

The CC2640R2 Bluetooth LE environment that comes with the SDK consists of two parts: An application image with the TI-RTOS kernel, drivers and Bluetooth profile and a stack image or library that implements Bluetooth Low Energy protocol.

As mentioned above, TI-RTOS is a multithreaded, real-time, pre-emptive operating system that runs the software solution with task synchronization. Within the RTOS, both the application and Bluetooth Low Energy protocol stack reside as independent tasks. But the highest priority is for the Bluetooth LE protocol stack.

For thread-safe synchronization between the application and stack, an indirect call (ICall) messaging framework is utilized. For an example architecture, see figure 3.9.

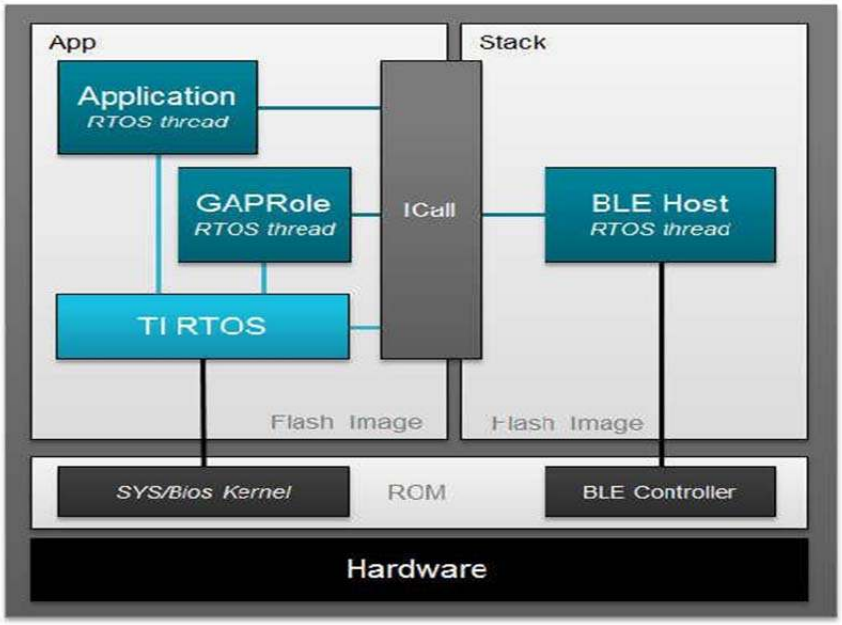

Figure 3.12. BLE example Software architecture (Software Architecture ti.com)

The stack image contains the lower layers of the Bluetooth Low Energy protocol stack, beginning with the Link Layer (LL) and as well as the GAP and GATT layers among other layers. The majority of the Bluetooth Low Energy protocol stack code is available as a library. The application image contains the RTOS, profiles, application code, drivers, and the ICall module.

As recently mentioned, sample examples are provided in the SDK. It is more practical to use those applications as a base and build custom Bluetooth LE application on top of them. Likewise, the examples used in this project uses “simple_pheripheral” and “simple_central” as a reference.

Since the aim of the project is to transfer measured battery data over Bluetooth, a project realizing that objective is used. Simple Serial Socket Server is the software example installed in the MCU that will act as a server or peripheral. The example showcases a simple UART over BLE implementation and it is based on TI's Simple Stream Server service to create a simple serial socket. The project is meant to allow the actual data sink and source to be readily exchanged for the purpose of developing a generic basic stream application. The following code snippet shows the UART API configuration written to read data from the BQ76PL455A and send it over BLE.

```
// Initialize UART
UART_Params uartParams;
UART_init();

// Open UART in callback mode for both read and write
UART_Params_init(&uartParams);
uartParams.writeDataMode  = UART_DATA_BINARY;
uartParams.readDataMode   = UART_DATA_BINARY;
uartParams.readReturnMode = UART_RETURN_FULL;
uartParams.readMode       = UART_MODE_CALLBACK;
uartParams.writeMode      = UART_MODE_CALLBACK;
uartParams.readCallback   = uartReadCallback;
uartParams.writeCallback  = uartWriteCallback;
//uartParams.readEcho       = UART_ECHO_OFF;
uartParams.baudRate       = 250000;

uartHandle = UART_open(Board_UART0, &uartParams);

if (uartHandle == NULL) {
  // UART_open() failed
  while (1);
}

// Enable partial return
UART_control(uartHandle, UARTCC26XX_CMD_RETURN_PARTIAL_ENABLE, NULL);

// Setup an initial read
UART_read(uartHandle, uartReadBuffer, UART_MAX_READ_SIZE);
UART_write(uartHandle, uartReadBuffer, sizeof(uartReadBuffer));
```

Figure 3.13. UART implementing code

On the other hand, the data sent by peripheral CC2640R2F MCU was either received by central MCU or by BLE application on the Mobile phone acting as a central called LightBlue. LightBlue is a mobile that can be used for simple BLE debugging and prototyping developed by company called Punch Through.

On the central MCU, the one that used to function as a master BLE device, Simple Serial Socket Client project is installed. It is also based on TI's Simple Stream Client profile to interact with counterpart Simple Stream Server peripheral. It contains two services: simple_service_discovery that provides APIs for populating service table and simple_stream_profile_client service which implements data sending and receiving services. Upon a connection, the profile provides a service table to be populated with service handles before any data can be transmitted from the client to the server. The service table could be easily populated by using APIs provided by simple service discovery service.

The client must activate notifications in order to receive incoming data from the peripheral. To accomplish this, the SimpleStreamClient_enable Notifications() API can be used which enable notifications given the connection handle.

The code will be available in the appendences, but the full stack project examples are on the SDK which can be downloaded from the TI's website.

4. RESULTS AND DISCUSSIONS

The project was implemented using one VDA battery module containing 12 cells of battery monitored and managed by one BQ76PL455A-Q1 integrated battery monitor. A slave CC2640R2F MCU is connected across the BQ76PL455A-Q1 to the Battery Module and a command to request voltage data of cells was sent from the BQ76PL455A-Q1 GUI application. The response data from the battery module containing packets of cell voltages was read by the slave MCU using UART RS-422 serial connection. Then, the slave MCU sends the voltage data to the Master MCU over Bluetooth LE.

By using hardware and software configuration that was explained in the previous chapter, the measured voltage data was read from the battery module and sent over Bluetooth LE from slave (peripheral) MCU to the master (Central) MCU.

A command to read battery voltage values was sent from BQ76PL455A-Q1 GUI application in a hexadecimal form and the voltage data of the battery module was read successfully and displayed in a hexadecimal format. As mentioned in the design reference document, data communication to and from the BQ76PL455A-Q1 device is in standard 8 data bit, 1 stop bit, no-parity universal asynchronous receiver and transmitter (UART) format. Every Byte of data in a data packet is shown as a two-character hexadecimal number (Holland, 2014).

The command **81 00 02 20 2884** was sent to the BQ76PL455 board to read voltages. The command translates as follows:

- 81 = Single Device Write with Response (8-bit register addressing)
- 00 = Device Address 0
- 02 = Register Address 2 (Command register)
- 20 = READ SAMPLED VALUES command
- 2884 = CRC

As shown in figure 4.1, the voltage data response was read by the slave or peripheral CC2640R2F MCU device. The data was show on display by using Realterm emulator application from COM port 6 with a baud rate of 250KB. The response packet contains 31 bytes of data which will be sent to the master or central CC2640R2F MCU over Bluetooth LE.

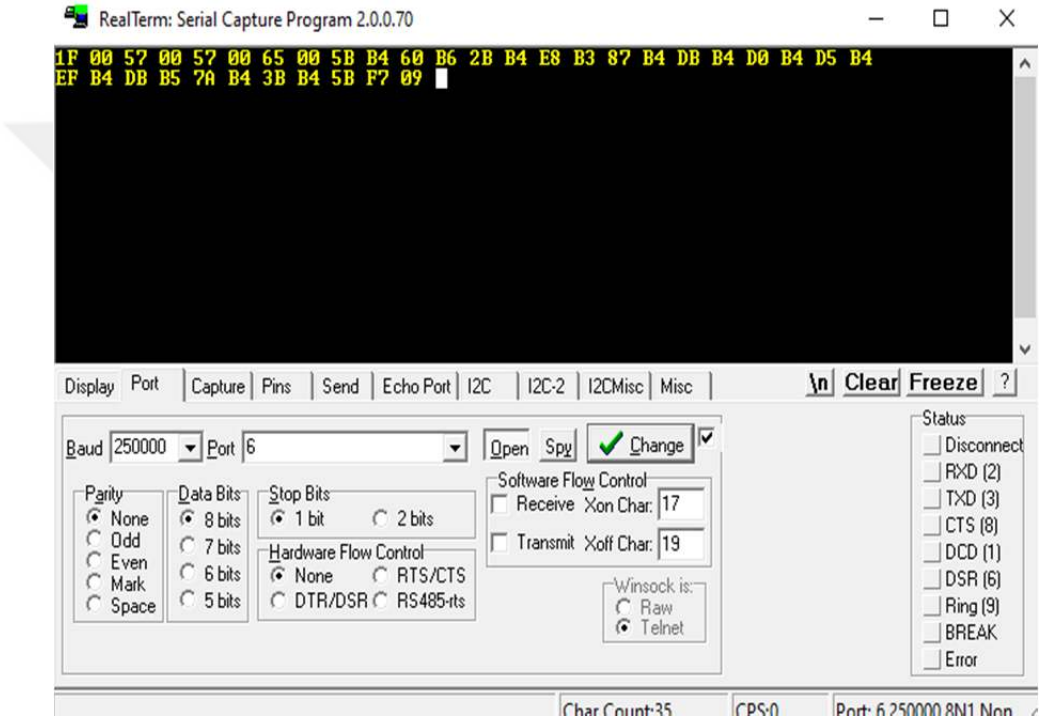

Figure 4.1. Response from BQ76PL455 board displayed over serial emulator app from communication port 6 of slave MCU before sending it over BLE

The first 2 characters show how many bytes of data has been transferred (In this case 31 bytes), the rest are cell voltages read and the last four characters are CRC for data corruption check. The breakdown of received response data from the BQ76PL455 board is as follows:

1F = Header file

0057 = cell 16 voltage data, in decimal (0.0068 V)

0057 = cell 15 voltage data, in decimal (0.0067 V)
0065 = cell 14 voltage data, in decimal (0.0077 V)
005B = cell 13 voltage data, in decimal (0.0069 V)
B460 = cell 12 voltage data, in decimal (3.5233 V)
B62B = cell 11 voltage data, in decimal (3.5581 V)
B4E8 = cell 10 voltage data, in decimal (3.5331 V)
B387 = cell 09 voltage data, in decimal (3.5061 V)
B4DB = cell 08 voltage data, in decimal (3.5321 V)
B4D0 = cell 07 voltage data, in decimal (3.5315 V)
B4D5 = cell 06 voltage data, in decimal (3.5322 V)
B4EF = cell 05 voltage data, in decimal (3.5339 V)
B4DB = cell 04 voltage data, in decimal (3.5320 V)
B57A = cell 03 voltage data, in decimal (3.5447 V)
B43B = cell 02 voltage data, in decimal (3.5200 V)
B45B = cell 01 voltage data, in decimal (3.5225 V)
F709 = CRC

The above voltage data measured by battery monitoring board was then sequentially sent over Bluetooth LE from the peripheral MCU to the master or central MCU. The Master CC2640R2F MCU was also connected to the serial emulator software (Realterm) through communication port 4 with the same UART configuration as the peripheral board. Every time data was read from the batteries and send it over BLE, it was observed that the data was immediately shown on the display of the master MCU side. Figure 4.2 shows voltage data from the peripheral MCU captured by Master MCU over Bluetooth LE.

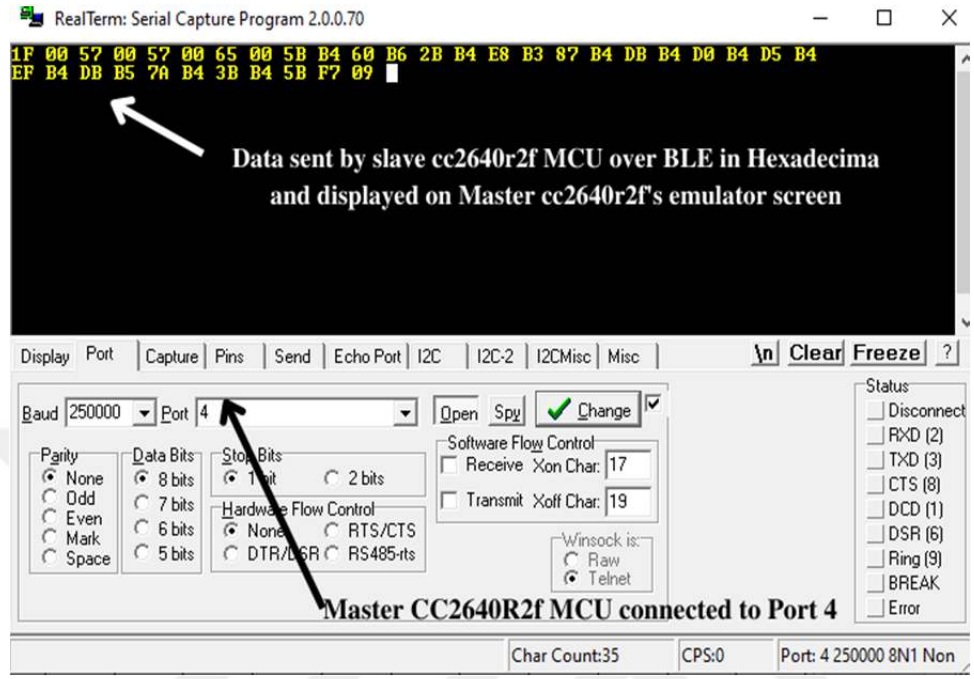

Figure 4.2. Measured Voltage data sent by Slave MCU over BLE to the Master MCU, displayed on Master MCU's emulator.

As can be seen from the above picture, the same data sent by the slave MCU was captured by the Master MCU. The process of sending commands of voltage read was repeated several times in different environments and frequencies, and there were no errors encountered in any time. All times in which the voltage data was sent from the slave MCU the same data has been captured by the master MCU without any corruption and loss and that shows the reliability of the technology.

Likewise, the same process was repeated for further validation and assurance, and the voltage data was sent over Bluetooth LE again, but this time the data was captured by LightBlue App on the mobile phone. This step was to make sure and test the reliability of the proposed system. It also shows that the BMS can be monitored over mobile phone or even IoT technology can be integrated to it.

The following figure shows data captured by the LightBlue app which was sent by the slave MCU over Bluetooth LE.

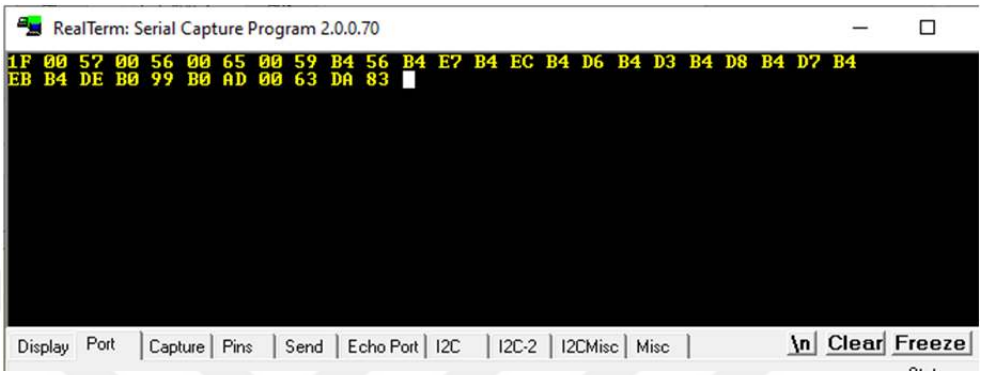

Figure 4.3. Voltage data sent on slave MCU's emulator to send it to the LightBlue app.

Figure 4.4 shows the interface of the LightBlue app. The firs line that starts with F000… is the UUID of the data reading service used by BLE for data transferring as explained section 3.4 in chapter Three. The voltage data starts from 0x1F and so forth. All the data sent by the slave MCU is received successfully.

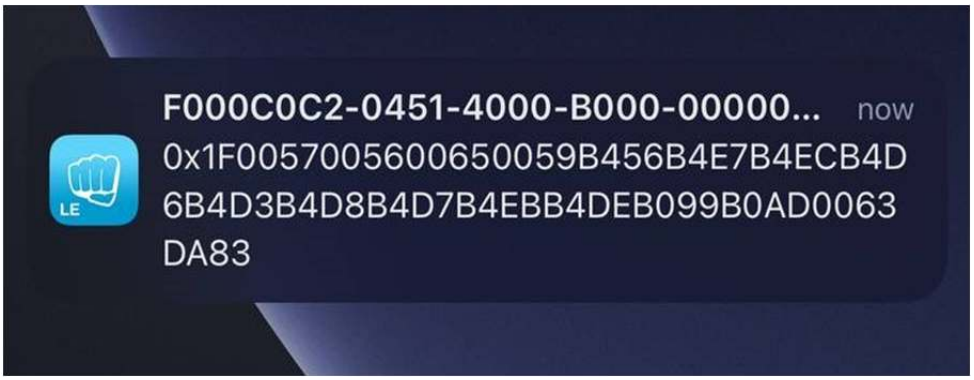

Figure 4.4. Voltage data of the battery module sent by slave MCU over BLE and received by LightBlue BLE app on iOS.

Finally, it is clear that the proposed WiBMS project can be reliable substitute for the traditional one. It can even be an interface that can be used for integration port of IoT, so that it can be connected to the cloud which can simplify the process of battery system management and make it be monitored efficiently and troubleshoot wirelessly.

5. CONCLUSION AND FUTURE WORK

In this project work, WiBMS is proposed by employing Bluetooth Low Energy protocol implemented on two CC2640R2F MCUs. Low cost and reliability features make BLE more preferable compared to other wireless technologies.

This experiment was carried out successfully and the system worked out as planned. Moreover, the developed Bluetooth Low Energy based system gave the expected results without worth mentioning challenges while respecting time boundaries and data integrity. Hence, these results show that BLE could the best candidate among other wireless technologies, thanks to its features such as high data throughput and low power consumption which make it high reliable and saves both time and cost when it is used in BMS for communications means

The work done in this book could be a starting point for future researchers interested in this specific topic of using BLE enabled MCU in battery management systems or related topics. It could save time and effort for them and would be beneficial in the academia

That being said, to make more promising BLE based WiBMS and address the shortcomings of this work, a full battery pack should be used for experiment and each module should have its own Bluetooth MCU. Also, the MCUs used in this project could be used for monitoring battery cells but further hardware tools may be needed.

Finally, it is clear that this topic has promising potential in the future as the world is shifting to wireless day after day, and battery management system needs to quickly adapt with the emerging technology to stay update and robust. Thus, it is highly advised and important that this field gets further research and development.

REFERENCES

Alonso, D., Opalko, O., Sigle, M., & Dostert, K. (2014, November 24). Towards a wireless battery management system: Evaluation of antennas and radio channel measurements inside a battery emulator. *IEEE Vehicular Technology Conference*. https://doi.org/10.1109/VTCFall.2014.6966212

Bansal, P., & Pr, N. (2019, December 1). Wireless Battery Management System for Electric Vehicles. *2019 IEEE Transportation Electrification Conference, ITEC-India 2019*. https://doi.org/10.1109/ITEC-India48457.2019.ITECIndia2019-83

Bessagnet, B., Allemand, N., Putaud, J.-P., Couvidat, F., André, J.-M., Simpson, D., Pisoni, E., Murphy, B. N., & Thunis, P. (2022). Emissions of Carbonaceous Particulate Matter and Ultrafine Particles from Vehicles—A Scientific Review in a Cross-Cutting Context of Air Pollution and Climate Change. *Applied Sciences*, *12*(7), 3623. https://doi.org/10.3390/app12073623

Cuma, M. U., & Koroglu, T. (2015). A comprehensive review on estimation strategies used in hybrid and battery electric vehicles. *Renewable and Sustainable Energy Reviews*, *42*, 517–531. https://doi.org/10.1016/j.rser.2014.10.047

Holland, S. (2014). *Application Note SLVA617A*. www.ti.com

Kutkut, N. H., Divan, D. M., & Novotny, D. W. (1995). Charge equalization for series connected battery strings. *Ieeexplore.Ieee.Org*, *31*(03). https://ieeexplore.ieee.org/abstract/document/382117/

Lee, M., Lee, J., Lee, I., Lee, J., & Chon, A. (2014). Wireless battery management system. *2013 World Electric Vehicle Symposium and Exhibition, EVS 2014*, 1–5. https://doi.org/10.1109/EVS.2013.6914889

Li, J., Zhou, S., & Han, Y. (2016). *Advances In Battery Manufacturing, Service, And Management Systems*.

Rahman, A., Rahman, M., & Rashid, M. (2017). Wireless Battery Management System of Electric Transport. *IOP Conference Series: Materials Science and Engineering*, *260*(1). https://doi.org/10.1088/1757-899X/260/1/012029

See, K. W., Wang, G., Zhang, Y., Wang, Y., Meng, L., Gu, X., Zhang, N., Lim, K. C., Zhao, L., & Xie, B. (2022). Critical review and functional safety of a battery management system for large-scale lithium-ion battery pack technologies. *International Journal of Coal Science & Technology*, *9*(1), 36. https://doi.org/10.1007/s40789-022-00494-0

Shell, C., Henderson, J., Verra, H., & Dyer, J. (2015). Implementation of a wireless battery management system (WBMS). *Conference Record - IEEE Instrumentation and Measurement Technology Conference*, *2015-July*, 1954–1959. https://doi.org/10.1109/I2MTC.2015.7151581

Software Architecture — SimpleLink™ CC2640R2 SDK User's Guide for BLE-Stack 3.x.x 3.03.08.00 documentation. (n.d.). Retrieved September 2, 2022, from https://dev.ti.com/tirex/explore/content/simplelink_cc2640r2_sdk_5_30_00_03/docs/blestack/ble_user_guide/html/cc2640/ble-software-architecture.html

Texas Instruments. (2015). *bq76PL455A-Q1*. www.ti.com

Texas Instruments Incorporated. (2020). *CC2640R2F SimpleLink™ Bluetooth ® 5.1 Low Energy Wireless MCU 1 Features • Microcontroller-Powerful Arm ® Cortex ®-M3-EEMBC CoreMark ® score: 142-Up to 48-MHz clock speed-275KB of nonvolatile memory including 128KB of in-system Programmable Flash-Up to 28KB of system SRAM, of which 20KB is ultra-low leakage SRAM-8KB of SRAM for cache or system RAM use-2-Pin cJTAG and JTAG debugging-Supports over-the-air upgrade (OTA)*. www.ti.com

The secret to moving faster with Bluetooth® 5 - Embedded processing - Technical articles - TI E2E support forums. (n.d.). Retrieved August 26, 2022, from https://e2e.ti.com/blogs_/b/process/posts/the-secret-to-moving-faster-with-bluetooth-5

Townsend, K., Cufi, C., Akiba, & Davidson, R. (2014). Getting Started with Bluetooth Low Energy. In *Pro Multithreading and Memory Management for iOS and OS X*. https://doi.org/10.1007/978-1-4302-4117-1_4

Vishnu, K., Nema, R. K., & Ojha, A. (2020, February 1). Various Types of Wireless Battery Management System in Ev. *2020 IEEE International Students' Conference on Electrical, Electronics and Computer Science, SCEECS 2020*. https://doi.org/10.1109/SCEECS48394.2020.115

Whaling, C. (2014). *EV power electronics cost analysis - IEEE Transportation Electrification Community*. https://tec.ieee.org/newsletter/may-june-2014/ev-power-electronics-cost-analysis

Woolley, M. (2022). *The Bluetooth® Low Energy Primer*.

Zimmer, G. (2012). Highly Accurate Hybrid/Electric Battery Monitor. *Power Electronics Europe*, *8*, 25–28.

CURRICULUM VITAE

Abdinasir A. Farah, attended his primary and secondary school in Lasanod, Northern Somalia. In 2014 he got an offer of scholarship from a local foundation operating in Somaliland and moved to Hargeisa to pursue his undergraduate journey. He graduated from Gollis University with a bachelor's degree in telecommunication engineering in 2017. In September 2018, he was granted a chance to do master's degree in electrical and electronics engineering at the institute of applied and natural sciences of Çukurova University in Adana, Türkiye. He will be graduating in October 2022. The author can be reached via his email: abdinaser.farah@gmail.com.

His academic interests and research areas include Bluetooth Low Energy, Embedded Microcontrollers, Battery Management Systems, and wireless technologies.

APPENDIX

simple_serial_socket_server.c

```
/*********************************************************************
***********

 @file       simple_serial_socket_server.c

 @brief This file contains the Simple Serial Socket server sample
application for use with the CC2640R2F Bluetooth Low Energy
Protocol Stack.
 Target Device: CC2640R2


*********************************************************************
**********
/*********************************************************************
***
 * INCLUDES
 */
#include <string.h>
#ifdef SHOW_BD_ADDR
#include <stdio.h>
#endif

#include <ti/sysbios/knl/Task.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/sysbios/knl/Event.h>
#include <ti/sysbios/knl/Queue.h>
#include <ti/display/Display.h>

#include <ti/drivers/UART.h>
#include <ti/drivers/uart/UARTCC26XX.h>
#include <ti/drivers/utils/List.h>

#include "board.h"

#include <icall.h>
/* This Header file contains all BLE API and icall structure
definition */
#include "icall_ble_api.h"
#include "devinfoservice.h"
#include "ll_common.h"
#include "peripheral.h"
#include "util.h"

#include "simple_serial_socket_server.h"
```

```
#include "simple_stream_profile_server.h"

/*********************************************************************
***
 * CONSTANTS
 */

// Advertising interval when device is discoverable (units of
625us, 160=100ms)
#define DEFAULT_ADVERTISING_INTERVAL          160

// General discoverable mode: advertise indefinitely
#define DEFAULT_DISCOVERABLE_MODE
GAP_ADTYPE_FLAGS_GENERAL

// Minimum connection interval (units of 1.25ms, 80=100ms) for
automatic
// parameter update request
#define DEFAULT_DESIRED_MIN_CONN_INTERVAL     80

// Maximum connection interval (units of 1.25ms, 800=1000ms) for
automatic
// parameter update request
#define DEFAULT_DESIRED_MAX_CONN_INTERVAL     800

// Slave latency to use for automatic parameter update request
#define DEFAULT_DESIRED_SLAVE_LATENCY         0

// Supervision timeout value (units of 10ms, 1000=10s) for
automatic parameter
// update request
#define DEFAULT_DESIRED_CONN_TIMEOUT          1000

// After the connection is formed, the peripheral waits until the
central
// device asks for its preferred connection parameters
#define DEFAULT_ENABLE_UPDATE_REQUEST
GAPROLE_LINK_PARAM_UPDATE_WAIT_REMOTE_PARAMS

// Connection Pause Peripheral time value (in seconds)
#define DEFAULT_CONN_PAUSE_PERIPHERAL         6

// Task configuration
#define SSSS_TASK_PRIORITY                     1
#ifndef SSSS_TASK_STACK_SIZE
#define SSSS_TASK_STACK_SIZE                   644
```

```
#endif

// Application events
#define SSSS_STATE_CHANGE_EVT                 0x0001
#define SSSS_CONN_EVT                         0x0002
#define SSSS_OUTGOING_DATA                    0x0020

// Internal Events for RTOS application
#define SSSS_ICALL_EVT                        ICALL_MSG_EVENT_ID
// Event_Id_31
#define SSSS_QUEUE_EVT                        UTIL_QUEUE_EVENT_ID
// Event_Id_30

// Bitwise OR of all events to pend on
#define SSSS_ALL_EVENTS                       (SSSS_ICALL_EVT
| \
                                              SSSS_QUEUE_EVT)

// Set the register cause to the registration bit-mask
#define CONNECTION_EVENT_REGISTER_BIT_SET(RegisterCause)
(connectionEventRegisterCauseBitMap |= RegisterCause )
// Remove the register cause from the registration bit-mask
#define CONNECTION_EVENT_REGISTER_BIT_REMOVE(RegisterCause)
(connectionEventRegisterCauseBitMap &= (~RegisterCause) )
// Gets whether the current App is registered to the receive
connection events
#define CONNECTION_EVENT_IS_REGISTERED
(connectionEventRegisterCauseBitMap > 0)
// Gets whether the RegisterCause was registered to recieve
connection event
#define CONNECTION_EVENT_REGISTRATION_CAUSE(RegisterCause)
(connectionEventRegisterCauseBitMap & RegisterCause )

//  UART read buffer size
#define UART_MAX_READ_SIZE    (256)

// Minimum heap headroom for BLE application
#define MIN_HEAP_HEADROOM     (4000)

/*********************************************************************
***
 * TYPEDEFS
 */

// App event passed from profiles.
typedef struct
```

```
{
  appEvtHdr_t hdr;    // event header.
  uint8_t    *pData;  // event data
  uint16_t   arg0;    // Generic argument
} ssssEvt_t;

// UART msg list element
typedef struct
{
  List_Elem elem;
  uint8_t len;
  uint8_t buffer[];
} uartMsg_t;

// The following typedef is used for registration to connection
event
typedef enum
{
  NOT_REGISTER       = 0,
  FOR_ATT_RSP        = 1,
  FOR_STREAM         = 2
}connectionEventRegisterCause_u;

/*********************************************************************
***
 * GLOBAL VARIABLES
 */

/*********************************************************************
***
 * LOCAL VARIABLES
 */

// Handle the registration and un-registration for the connection
event, since only one can be registered.
static uint32_t       connectionEventRegisterCauseBitMap =
NOT_REGISTER; //see connectionEventRegisterCause_u

// Entity ID globally used to check for source and/or destination
of messages
static ICall_EntityID selfEntity;

// Event globally used to post local events and pend on system and
// local events.
static ICall_SyncHandle syncEvent;
```

```
// Queue object used for app messages
static Queue_Struct appMsg;
static Queue_Handle appMsgQueue;

// Task configuration
Task_Struct ssssTask;
uint8_t ssssTaskStack[SSSS_TASK_STACK_SIZE];

// Scan response data (max size = 31 bytes)
static uint8_t scanRspData[] =
{
  // complete name
  0x09,   // length of this data
  GAP_ADTYPE_LOCAL_NAME_COMPLETE,
  'B',
  'M',
  'S',
  'S',
  'L',
  'A',
  'V',
  'E',

  // connection interval range
  0x05,   // length of this data
  GAP_ADTYPE_SLAVE_CONN_INTERVAL_RANGE,
  LO_UINT16(DEFAULT_DESIRED_MIN_CONN_INTERVAL),   // 100ms
  HI_UINT16(DEFAULT_DESIRED_MIN_CONN_INTERVAL),
  LO_UINT16(DEFAULT_DESIRED_MAX_CONN_INTERVAL),   // 1s
  HI_UINT16(DEFAULT_DESIRED_MAX_CONN_INTERVAL),

  // Tx power level
  0x02,   // length of this data
  GAP_ADTYPE_POWER_LEVEL,
  0       // 0dBm
};

// Advertisement data (max size = 31 bytes, though this is
// best kept short to conserve power while advertising)
static uint8_t advertData[] =
{
  // Flags: this field sets the device to use general discoverable
  // mode (advertises indefinitely) instead of general
  // discoverable mode (advertise for 30 seconds at a time)
  0x02,   // length of this data
  GAP_ADTYPE_FLAGS,
```

```
  DEFAULT_DISCOVERABLE_MODE | GAP_ADTYPE_FLAGS_BREDR_NOT_SUPPORTED,

  // service UUID, to notify central devices what services are
included
  // in this peripheral
  0x11,   // length of this data
  GAP_ADTYPE_128BIT_MORE,      // some of the UUID's, but not all
  TI_BASE_UUID_128(SIMPLESTREAMSERVER_SERV_UUID),

  // ID nugget
  0x06,
  GAP_ADTYPE_MANUFACTURER_SPECIFIC,
  // Texas Instruments company ID
  0x0D,
  0x00,
  // Custom data identifier
  0xC0,
  0xFF,
  0xEE
};

// GAP GATT Attributes
static uint8_t attDeviceName[GAP_DEVICE_NAME_LEN] = "BMS Slave ";

// Globals used for ATT Response retransmission
static gattMsgEvent_t *pAttRsp = NULL;
static uint8_t rspTxRetry = 0;

// Global connection handle
static uint16_t connHandle = 0xFFFF;

// UART Interface
static UART_Handle uartHandle = NULL;

// UART read buffer
static uint8_t uartReadBuffer[UART_MAX_READ_SIZE] = {0};

// UART write list
static List_List  uartWriteList;
static uartMsg_t* uartCurrentMsg;
static uint8_t    uartWriteActive = 0;

// Pin driver handle
static PIN_Handle ledPinHandle;
static PIN_State ledPinState;
```

```
// PIN configuration for LEDs
static PIN_Config ledPinTable[] = {
    Board_RLED | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL |
PIN_DRVSTR_MAX,
    Board_GLED | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL |
PIN_DRVSTR_MAX,
    PIN_TERMINATE
};

/*********************************************************************
***
 * LOCAL FUNCTIONS
 */
static void uartReadCallback(UART_Handle handle, void *rxBuf,
size_t size);
static void uartWriteCallback(UART_Handle handle, void *rxBuf,
size_t size);

static void SimpleStreamServer_cccUpdateCB(uint16_t value);
static void SimpleStreamServer_incomingDataCB(uint16_t connHandle,
                              uint8_t paramID, uint16_t len,
                              uint8_t *pValue);

static void SimpleSerialSocketServer_init( void );
static void SimpleSerialSocketServer_taskFxn(UArg a0, UArg a1);

static uint8_t SimpleSerialSocketServer_processStackMsg(ICall_Hdr
*pMsg);
static uint8_t
SimpleSerialSocketServer_processGATTMsg(gattMsgEvent_t *pMsg);
static void
SimpleSerialSocketServer_processConnEvt(Gap_ConnEventRpt_t
*pReport);
static void SimpleSerialSocketServer_processAppMsg(ssssEvt_t
*pMsg);
static void
SimpleSerialSocketServer_processStateChangeEvt(gaprole_States_t
newState);

static void SimpleSerialSocketServer_sendAttRsp(void);
static void SimpleSerialSocketServer_freeAttRsp(uint8_t status);

static void SimpleSerialSocketServer_stateChangeCB(gaprole_States_t
newState);
static uint8_t SimpleSerialSocketServer_enqueueMsg(uint8_t event,
uint8_t state,
```

```
                                                uint8_t *pData,
uint16_t arg0);

static void SimpleSerialSocketServer_connEvtCB(Gap_ConnEventRpt_t
*pReport);

/*********************************************************************
***
 * EXTERN FUNCTIONS
 */
extern void AssertHandler(uint8 assertCause, uint8 assertSubcause);

/*********************************************************************
***
 * PROFILE CALLBACKS
 */

// Peripheral GAPRole Callbacks
static gapRolesCBs_t SimpleSerialSocketServer_gapRoleCBs =
{
  SimpleSerialSocketServer_stateChangeCB     // GAPRole State
Change Callbacks
};

// SimpleStreamServer Profile Callbacks
static SimpleStreamServerCBs_t
SimpleStreamServer_SimpleStreamServerProfileCBs =
{
  .pfnCccUpdateCb     = SimpleStreamServer_cccUpdateCB,    //
Characteristic configuration update handler
  .pfnIncomingDataCb  = SimpleStreamServer_incomingDataCB, //
Incoming data handler
};

/*********************************************************************
***
 * PUBLIC FUNCTIONS
 */

/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_RegisterToAllConnectionEvent()
 *
 * @brief   register to receive connection events for all the
connection
 *
```

```
 * @param connectionEventRegisterCause represents the reason for
registration
 *
 * @return @ref SUCCESS
 *
 */
bStatus_t SimpleSerialSocketServer_RegisterToAllConnectionEvent
(connectionEventRegisterCause_u connectionEventRegisterCause)
{
  bStatus_t status = SUCCESS;

  // in case  there is no registration for the connection event,
make the registration
  if (!CONNECTION_EVENT_IS_REGISTERED)
  {
    status =
GAP_RegisterConnEventCb(SimpleSerialSocketServer_connEvtCB,
GAP_CB_REGISTER, LINKDB_CONNHANDLE_ALL);
  }
  if(status == SUCCESS)
  {
    //add the reason bit to the bitamap.

CONNECTION_EVENT_REGISTER_BIT_SET(connectionEventRegisterCause);
  }

  return(status);
}

/*********************************************************************
***
 * @fn
SimpleSerialSocketServer_UnRegisterToAllConnectionEvent()
 *
 * @brief   Unregister connection events
 *
 * @param connectionEventRegisterCause represents the reason for
registration
 *
 * @return @ref SUCCESS
 *
 */
bStatus_t SimpleSerialSocketServer_UnRegisterToAllConnectionEvent
(connectionEventRegisterCause_u connectionEventRegisterCause)
{
  bStatus_t status = SUCCESS;
```

```
CONNECTION_EVENT_REGISTER_BIT_REMOVE(connectionEventRegisterCause);
  // in case  there is no more registration for the connection
event than unregister
  if (!CONNECTION_EVENT_IS_REGISTERED)
  {
    GAP_RegisterConnEventCb(SimpleSerialSocketServer_connEvtCB,
GAP_CB_UNREGISTER, LINKDB_CONNHANDLE_ALL);
  }

  return(status);
}


/*********************************************************************
***
 * @fn      uartReadCallback
 *
 * @brief   UART read callback, notify the application it needs to
handle the data
 *
 * @param   handle  - UART handle
 *          *rxBuf  - buffer containing the incoming data.
 *          size    - length of the data buffer.
 *
 * @return  None.
 */
static void uartReadCallback(UART_Handle handle, void *rxBuf,
size_t size)
{
    // Pass the number of read bytes using the arg0 field
    SimpleSerialSocketServer_enqueueMsg(SSSS_OUTGOING_DATA, NULL,
NULL, size);
}

/*********************************************************************
***
 * @fn      uartWriteCallback
 *
 * @brief   UART write callback, perform additional writes if the
UART msg
 *          list is not empty
 *
 * @param   handle  - UART handle
 *          *txBuf  - buffer containing the written data.
```

```
 *          size    - length of the data buffer written.
 *
 * @return  None.
 */
static void uartWriteCallback(UART_Handle handle, void *txBuf,
size_t size)
{
    // Free the last printed buffer
    if (NULL != uartCurrentMsg)
    {
        ICall_free(uartCurrentMsg);
        uartCurrentMsg = NULL;
    }

    // If the list is not empty, start another write
    if (!List_empty(&uartWriteList)) {
        uartCurrentMsg = (uartMsg_t *) List_get(&uartWriteList);
        UART_write(uartHandle, uartCurrentMsg->buffer,
uartCurrentMsg->len);
    }
    else
    {
        uartWriteActive = 0;
    }
}

/*********************************************************************
***
 * @fn      SimpleStreamServer_cccUpdateCB
 *
 * @brief   Callback from SimpleStreamServer Profile indicating
state change
 *
 * @param   connHandle - connection handle tied to the state change
 *          state      - New state
 *
 * @return  None.
 */
static void SimpleStreamServer_cccUpdateCB(uint16_t value)
{
    // Notifications is active
    if (value == 0x0001) {
        // Indicate that the stream is enabled
        PIN_setOutputValue(ledPinHandle, Board_GLED, 1);
    }
    else
```

```
    {
        // Indicate that the stream is disabled
        PIN_setOutputValue(ledPinHandle, Board_GLED, 0);
    }
}

/*********************************************************************
***
 * @fn      SimpleStreamServer_incomingDataCB
 *
 * @brief   Callback from SimpleStreamServer Profile indicating
incoming data
 *
 * @param   paramId - parameter Id of the value that was changed.
 *          len     - length of the data buffer.
 *          *data - buffer containing the incoming data.
 *
 * @return  None.
 */
static void SimpleStreamServer_incomingDataCB(uint16_t connHandle,
                                              uint8_t paramID,
uint16_t len,
                                              uint8_t *data)
{
    // Try to allocate and store the data to our UART write queue
    uartMsg_t *newMsg;
    newMsg =
SimpleStreamServer_allocateWithHeadroom(sizeof(uartMsg_t) + len);

    if (newMsg)
    {
        newMsg->len = len;
        memcpy(newMsg->buffer, data, len);

        List_put(&uartWriteList, (List_Elem *) newMsg);

        // If no write is currently active, start a new one
        if (uartWriteActive == 0)
        {
            uartWriteActive = 1;
            uartCurrentMsg = (uartMsg_t *)
List_get(&uartWriteList);

            UART_write(uartHandle, uartCurrentMsg->buffer,
uartCurrentMsg->len);
        }
```

```
    }
}


/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_createTask
 *
 * @brief   Task creation function for the Simple Peripheral.
 *
 * @param   None.
 *
 * @return  None.
 */
void SimpleSerialSocketServer_createTask(void)
{
  Task_Params taskParams;

  // Configure task
  Task_Params_init(&taskParams);
  taskParams.stack = ssssTaskStack;
  taskParams.stackSize = SSSS_TASK_STACK_SIZE;
  taskParams.priority = SSSS_TASK_PRIORITY;

  Task_construct(&ssssTask, SimpleSerialSocketServer_taskFxn,
&taskParams, NULL);
}

/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_init
 *
 * @brief   Called during initialization and contains application
 *          specific initialization (ie. hardware
initialization/setup,
 *          table initialization, power up notification, etc), and
 *          profile initialization/setup.
 *
 * @param   None.
 *
 * @return  None.
 */
static void SimpleSerialSocketServer_init(void)
{
  //
*********************************************************************
```

```
  // N0 STACK API CALLS CAN OCCUR BEFORE THIS CALL TO
ICall_registerApp
  //
*****************************************************************
  // Register the current thread as an ICall dispatcher application
  // so that the application can send and receive messages.
  ICall_registerApp(&selfEntity, &syncEvent);

  // Create an RTOS queue for message from profile to be sent to
app.
  appMsgQueue = Util_constructQueue(&appMsg);

  // Initialize and open PIN driver to use LEDs
  PIN_init(ledPinTable);
  ledPinHandle = PIN_open(&ledPinState, ledPinTable);

  PIN_setOutputValue(ledPinHandle, Board_RLED, 0);
  PIN_setOutputValue(ledPinHandle, Board_GLED, 0);

  // Initialize UART
  UART_Params uartParams;
  UART_init();

  // Open UART in callback mode for both read and write
  UART_Params_init(&uartParams);
  uartParams.writeDataMode  = UART_DATA_BINARY;
  uartParams.readDataMode   = UART_DATA_BINARY;
  uartParams.readReturnMode = UART_RETURN_FULL;
  uartParams.readMode       = UART_MODE_CALLBACK;
  uartParams.writeMode      = UART_MODE_CALLBACK;
  uartParams.readCallback   = uartReadCallback;
  uartParams.writeCallback  = uartWriteCallback;
  //uartParams.readEcho       = UART_ECHO_OFF;
  uartParams.baudRate       = 250000;

  uartHandle = UART_open(Board_UART0, &uartParams);

  if (uartHandle == NULL) {
    // UART_open() failed
    while (1);
  }

  // Enable partial return
  UART_control(uartHandle, UARTCC26XX_CMD_RETURN_PARTIAL_ENABLE,
NULL);
```

```
  // Setup an initial read
  UART_read(uartHandle, uartReadBuffer, UART_MAX_READ_SIZE);
  //UART_write(uartHandle, uartReadBuffer, sizeof(uartReadBuffer));

  // Set GAP Parameters: After a connection was established, delay
in seconds
  // before sending when
GAPRole_SetParameter(GAPROLE_PARAM_UPDATE_ENABLE,...)
  // uses GAPROLE_LINK_PARAM_UPDATE_INITIATE_BOTH_PARAMS or
  // GAPROLE_LINK_PARAM_UPDATE_INITIATE_APP_PARAMS
  // For current defaults, this has no effect.
  GAP_SetParamValue(TGAP_CONN_PAUSE_PERIPHERAL,
DEFAULT_CONN_PAUSE_PERIPHERAL);

  // Setup the Peripheral GAPRole Profile. For more information see
the User's
  // Guide:
  // http://software-dl.ti.com/lprf/sdg-latest/html/
  {
    // Device starts advertising upon initialization of GAP
    uint8_t initialAdvertEnable = TRUE;

    // By setting this to zero, the device will go into the waiting
state after
    // being discoverable for 30.72 second, and will not being
advertising again
    // until re-enabled by the application
    uint16_t advertOffTime = 0;

    uint8_t enableUpdateRequest = DEFAULT_ENABLE_UPDATE_REQUEST;
    uint16_t desiredMinInterval =
DEFAULT_DESIRED_MIN_CONN_INTERVAL;
    uint16_t desiredMaxInterval =
DEFAULT_DESIRED_MAX_CONN_INTERVAL;
    uint16_t desiredSlaveLatency = DEFAULT_DESIRED_SLAVE_LATENCY;
    uint16_t desiredConnTimeout = DEFAULT_DESIRED_CONN_TIMEOUT;

    // Set the Peripheral GAPRole Parameters
    GAPRole_SetParameter(GAPROLE_ADVERT_ENABLED, sizeof(uint8_t),
                         &initialAdvertEnable);
    GAPRole_SetParameter(GAPROLE_ADVERT_OFF_TIME, sizeof(uint16_t),
                         &advertOffTime);

    GAPRole_SetParameter(GAPROLE_SCAN_RSP_DATA,
sizeof(scanRspData),
                         scanRspData);
```

```
    GAPRole_SetParameter(GAPROLE_ADVERT_DATA, sizeof(advertData),
advertData);

    GAPRole_SetParameter(GAPROLE_PARAM_UPDATE_ENABLE,
sizeof(uint8_t),
                         &enableUpdateRequest);
    GAPRole_SetParameter(GAPROLE_MIN_CONN_INTERVAL,
sizeof(uint16_t),
                         &desiredMinInterval);
    GAPRole_SetParameter(GAPROLE_MAX_CONN_INTERVAL,
sizeof(uint16_t),
                         &desiredMaxInterval);
    GAPRole_SetParameter(GAPROLE_SLAVE_LATENCY, sizeof(uint16_t),
                         &desiredSlaveLatency);
    GAPRole_SetParameter(GAPROLE_TIMEOUT_MULTIPLIER,
sizeof(uint16_t),
                         &desiredConnTimeout);
  }

  // Set the Device Name characteristic in the GAP GATT Service
  // For more information, see the section in the User's Guide:
  // http://software-dl.ti.com/lprf/sdg-latest/html
  GGS_SetParameter(GGS_DEVICE_NAME_ATT, GAP_DEVICE_NAME_LEN,
attDeviceName);

  // Set GAP Parameters to set the advertising interval
  // For more information, see the GAP section of the User's Guide:
  // http://software-dl.ti.com/lprf/sdg-latest/html
  {
    // Use the same interval for general and limited advertising.
    // Note that only general advertising will occur based on the
above configuration
    uint16_t advInt = DEFAULT_ADVERTISING_INTERVAL;

    GAP_SetParamValue(TGAP_LIM_DISC_ADV_INT_MIN, advInt);
    GAP_SetParamValue(TGAP_LIM_DISC_ADV_INT_MAX, advInt);
    GAP_SetParamValue(TGAP_GEN_DISC_ADV_INT_MIN, advInt);
    GAP_SetParamValue(TGAP_GEN_DISC_ADV_INT_MAX, advInt);
  }

  // Initialize GATT attributes
  GGS_AddService(GATT_ALL_SERVICES);           // GAP GATT Service
  GATTServApp_AddService(GATT_ALL_SERVICES);   // GATT Service
  DevInfo_AddService();                        // Device
Information Service
```

```
  // Initialize Simple data stream service
  SimpleStreamServer_AddService(GATT_ALL_SERVICES);

SimpleStreamServer_RegisterAppCBs(&SimpleStreamServer_SimpleStreamS
erverProfileCBs);
  // Make sure to leave at least MIN_HEAP_HEADROOM of heap
  // for the BLE stack application to operate.
  SimpleStreamServer_setHeadroomLimit(MIN_HEAP_HEADROOM);

  // Register to connection events

SimpleSerialSocketServer_RegisterToAllConnectionEvent(FOR_STREAM);

  // Start the Device
  VOID GAPRole_StartDevice(&SimpleSerialSocketServer_gapRoleCBs);

  // Register with GAP for HCI/Host messages. This is needed to
receive HCI
  // events. For more information, see the section in the User's
Guide:
  // http://software-dl.ti.com/lprf/sdg-latest/html
  GAP_RegisterForMsgs(selfEntity);

  // Register for GATT local events and ATT Responses pending for
transmission
  GATT_RegisterForMsgs(selfEntity);

#if !defined (USE_LL_CONN_PARAM_UPDATE)
  // Get the currently set local supported LE features
  // The HCI will generate an HCI event that will get received in
the main
  // loop
  HCI_LE_ReadLocalSupportedFeaturesCmd();
#endif // !defined (USE_LL_CONN_PARAM_UPDATE)
}

/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_taskFxn
 *
 * @brief   Application task entry point for the Simple Peripheral.
 *
 * @param   a0, a1 - not used.
 *
 * @return  None.
 */
```

```
static void SimpleSerialSocketServer_taskFxn(UArg a0, UArg a1)
{
  // Initialize application
  SimpleSerialSocketServer_init();

  // Application main loop
  for (;;)
  {
    uint32_t events;

    // Waits for an event to be posted associated with the calling
thread.
    // Note that an event associated with a thread is posted when a
    // message is queued to the message receive queue of the thread
    events = Event_pend(syncEvent, Event_Id_NONE, SSSS_ALL_EVENTS,
                        ICALL_TIMEOUT_FOREVER);

    if (events)
    {
      ICall_EntityID dest;
      ICall_ServiceEnum src;
      ICall_HciExtEvt *pMsg = NULL;

      // Fetch any available messages that might have been sent
from the stack
      if (ICall_fetchServiceMsg(&src, &dest,
                                (void **)&pMsg) ==
ICALL_ERRNO_SUCCESS)
      {
        uint8 safeToDealloc = TRUE;

        if ((src == ICALL_SERVICE_CLASS_BLE) && (dest ==
selfEntity))
        {
          ICall_Stack_Event *pEvt = (ICall_Stack_Event *)pMsg;

          if (pEvt->signature != 0xffff)
          {
            // Process inter-task message
            safeToDealloc =
SimpleSerialSocketServer_processStackMsg((ICall_Hdr *)pMsg);
          }
        }

        if (pMsg && safeToDealloc)
        {
```

```
          ICall_freeMsg(pMsg);
        }
      }

      // If RTOS queue is not empty, process app message.
      if (events & SSSS_QUEUE_EVT)
      {
        while (!Queue_empty(appMsgQueue))
        {
            ssssEvt_t *pMsg = (ssssEvt_t
*)Util_dequeueMsg(appMsgQueue);
          if (pMsg)
          {
            // Process message.
            SimpleSerialSocketServer_processAppMsg(pMsg);

            // Free the space from the message.
            ICall_free(pMsg);
          }
        }
      }
    }
  }
}

/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_processStackMsg
 *
 * @brief   Process an incoming stack message.
 *
 * @param   pMsg - message to process
 *
 * @return  TRUE if safe to deallocate incoming message, FALSE
otherwise.
 */
static uint8_t SimpleSerialSocketServer_processStackMsg(ICall_Hdr
*pMsg)
{
  uint8_t safeToDealloc = TRUE;

  switch (pMsg->event)
  {
    case GATT_MSG_EVENT:
      // Process GATT message
```

```
      safeToDealloc =
SimpleSerialSocketServer_processGATTMsg((gattMsgEvent_t *)pMsg);
      break;

    case HCI_GAP_EVENT_EVENT:
      {

        // Process HCI message
        switch(pMsg->status)
        {
          case HCI_COMMAND_COMPLETE_EVENT_CODE:
            // Process HCI Command Complete Event
            {

#if !defined (USE_LL_CONN_PARAM_UPDATE)
              // This code will disable the use of the
LL_CONNECTION_PARAM_REQ
              // control procedure (for connection parameter
updates, the
              // L2CAP Connection Parameter Update procedure will
be used
              // instead). To re-enable the LL_CONNECTION_PARAM_REQ
control
              // procedures, define the symbol
USE_LL_CONN_PARAM_UPDATE
              // The L2CAP Connection Parameter Update procedure is
used to
              // support a delta between the minimum and maximum
connection
              // intervals required by some iOS devices.

              // Parse Command Complete Event for opcode and status
              hciEvt_CmdComplete_t* command_complete =
(hciEvt_CmdComplete_t*) pMsg;
              uint8_t   pktStatus = command_complete-
>pReturnParam[0];

              //find which command this command complete is for
              switch (command_complete->cmdOpcode)
              {
                case HCI_LE_READ_LOCAL_SUPPORTED_FEATURES:
                  {
                    if (pktStatus == SUCCESS)
                    {
                      uint8_t featSet[8];
```

```
                        // Get current feature set from received
event (bits 1-9
                        // of the returned data
                        memcpy( featSet, &command_complete-
>pReturnParam[1], 8 );

                        // Clear bit 1 of byte 0 of feature set to
disable LL
                        // Connection Parameter Updates
                        CLR_FEATURE_FLAG( featSet[0],
LL_FEATURE_CONN_PARAMS_REQ );

                        // Update controller with modified features
                        HCI_EXT_SetLocalSupportedFeaturesCmd( featSet
);
                      }
                    }
                    break;

                  default:
                    //do nothing
                    break;
                }
#endif // !defined (USE_LL_CONN_PARAM_UPDATE)

              }
              break;

            case HCI_BLE_HARDWARE_ERROR_EVENT_CODE:
              AssertHandler(HAL_ASSERT_CAUSE_HARDWARE_ERROR,0);
              break;

            default:
              break;
          }
        }
        break;

        default:
          // do nothing
          break;

      }

  return (safeToDealloc);
}
```

```
/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_processGATTMsg
 *
 * @brief   Process GATT messages and events.
 *
 * @return  TRUE if safe to deallocate incoming message, FALSE
otherwise.
 */
static uint8_t
SimpleSerialSocketServer_processGATTMsg(gattMsgEvent_t *pMsg)
{
  // See if GATT server was unable to transmit an ATT response
    {
  if (pMsg->hdr.status == blePending)
  {
    // No HCI buffer was available. Let's try to retransmit the
response
    // on the next connection event.
    if(
SimpleSerialSocketServer_RegisterToAllConnectionEvent(FOR_ATT_RSP)
== SUCCESS)
      // First free any pending response
      SimpleSerialSocketServer_freeAttRsp(FAILURE);

      // Hold on to the response message for retransmission
      pAttRsp = pMsg;

      // Don't free the response message yet
      return (FALSE);
    }
  }

  // Free message payload. Needed only for ATT Protocol messages
  GATT_bm_free(&pMsg->msg, pMsg->method);

  // It's safe to free the incoming message
  return (TRUE);
}

/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_processConnEvt
 *
 * @brief   Process connection event.
```

```
 *
 * @param pReport pointer to connection event report
 */
static void
SimpleSerialSocketServer_processConnEvt(Gap_ConnEventRpt_t
*pReport)
{
  if( CONNECTION_EVENT_REGISTRATION_CAUSE(FOR_ATT_RSP))
  {
    // The GATT server might have returned a blePending as it was
trying
    // to process an ATT Response. Now that we finished with this
    // connection event, let's try sending any remaining ATT
Responses
    // on the next connection event.
    // Try to retransmit pending ATT Response (if any)
    SimpleSerialSocketServer_sendAttRsp();
  }

  if (CONNECTION_EVENT_REGISTRATION_CAUSE(FOR_STREAM))
  {
    // The stream is active, process the stream at every connection
event.
    bStatus_t status = SimpleStreamServer_processStream();

    // If status is successful there is no data pending for
processing
    if (status == SUCCESS)
    {

SimpleSerialSocketServer_UnRegisterToAllConnectionEvent(FOR_STREAM)
;
    }
  }
}

/*********************************************************************
***
 *
 * @brief   Send a pending ATT response message.
 *
 * @param   none
 *
 * @return  none
 */
static void SimpleSerialSocketServer_sendAttRsp(void)
```

```
{
  // See if there's a pending ATT Response to be transmitted
  if (pAttRsp != NULL)
  {
    uint8_t status;

    // Increment retransmission count
    rspTxRetry++;

    // Try to retransmit ATT response till either we're successful
or
    // the ATT Client times out (after 30s) and drops the
connection.
    status = GATT_SendRsp(pAttRsp->connHandle, pAttRsp->method,
&(pAttRsp->msg));
    if ((status != blePending) && (status != MSG_BUFFER_NOT_AVAIL))
    {
      // Disable connection event end notice
      SimpleSerialSocketServer_UnRegisterToAllConnectionEvent
(FOR_ATT_RSP);
      // We're done with the response message
      SimpleSerialSocketServer_freeAttRsp(status);
    }
  }
}

/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_freeAttRsp
 *
 * @brief   Free ATT response message.
 *
 * @param   status - response transmit status
 *
 * @return  none
 */
static void SimpleSerialSocketServer_freeAttRsp(uint8_t status)
{
  // See if there's a pending ATT response message
  if (pAttRsp != NULL)
  {
    // See if the response was sent out successfully
    if (status != SUCCESS)
    {
      // Free response payload
      GATT_bm_free(&pAttRsp->msg, pAttRsp->method);
```

```
    }

    // Free response message
    ICall_freeMsg(pAttRsp);

    // Reset our globals
    pAttRsp = NULL;
    rspTxRetry = 0;
  }
}

/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_processAppMsg
 *
 * @brief   Process an incoming callback from a profile.
 *
 * @param   pMsg - message to process
 *
 * @return  None.
 */
static void SimpleSerialSocketServer_processAppMsg(ssssEvt_t *pMsg)
{
  switch (pMsg->hdr.event)
  {
   // Outgoing data event
   case SSSS_OUTGOING_DATA:
     {
       // You could do processing of data here ...

       // For now, just Send the data read from UART over the air
       bStatus_t status = SimpleStreamServer_sendData(connHandle,
&uartReadBuffer, pMsg->arg0);

       // If status is not successful, register for connection
events for further processing
       if (status != SUCCESS) {

SimpleSerialSocketServer_RegisterToAllConnectionEvent(FOR_STREAM);
       }

       // Start another read
       UART_read(uartHandle, uartReadBuffer, UART_MAX_READ_SIZE);

       break;
     }
```

```
    // State change event
    case SSSS_STATE_CHANGE_EVT:
      {

SimpleSerialSocketServer_processStateChangeEvt((gaprole_States_t)pM
sg->
                                                        hdr.state);
        break;
      }

    // Connection event
    case SSSS_CONN_EVT:
      {
        SimpleSerialSocketServer_processConnEvt((Gap_ConnEventRpt_t
*)(pMsg->pData));

        ICall_free(pMsg->pData);
        break;
      }

    default:
      // Do nothing.
      break;
  }
}

/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_stateChangeCB
 *
 * @brief   Callback from GAP Role indicating a role state change.
 *
 * @param   newState - new state
 *
 * @return  None.
 */
static void SimpleSerialSocketServer_stateChangeCB(gaprole_States_t
newState)
{
  SimpleSerialSocketServer_enqueueMsg(SSSS_STATE_CHANGE_EVT,
newState, NULL, NULL);
}

/*********************************************************************
***
```

```
 * @fn      SimpleSerialSocketServer_processStateChangeEvt
 *
 * @brief   Process a pending GAP Role state change event.
 *
 * @param   newState - new state
 *
 * @return  None.
 */
static void
SimpleSerialSocketServer_processStateChangeEvt(gaprole_States_t
newState)
{
  switch ( newState )
  {
    case GAPROLE_STARTED:
      {
        uint8_t ownAddress[B_ADDR_LEN];
        uint8_t systemId[DEVINFO_SYSTEM_ID_LEN];

        GAPRole_GetParameter(GAPROLE_BD_ADDR, ownAddress);

        // use 6 bytes of device address for 8 bytes of system ID
value
        systemId[0] = ownAddress[0];
        systemId[1] = ownAddress[1];
        systemId[2] = ownAddress[2];

        // set middle bytes to zero
        systemId[4] = 0x00;
        systemId[3] = 0x00;

        // shift three bytes up
        systemId[7] = ownAddress[5];
        systemId[6] = ownAddress[4];
        systemId[5] = ownAddress[3];

        DevInfo_SetParameter(DEVINFO_SYSTEM_ID,
DEVINFO_SYSTEM_ID_LEN, systemId);

#ifdef SHOW_BD_ADDR
        // Display device address
        uint8_t printAddress[sizeof("Device Address: ") +
(2*B_ADDR_LEN)+3 + sizeof("\n\r")];
        uint8_t len = sprintf(printAddress, "Device Address:
%s\n\r", Util_convertBdAddr2Str(ownAddress));
        UART_write(uartHandle, printAddress, len);
```

```
#endif

      }
      break;

    case GAPROLE_CONNECTED:
      {
        uint8_t numActive = 0;

        numActive = linkDB_NumActive();

        // Use numActive to determine the connection handle of the
last
        // connection and save it
        if (connHandle == 0xFFFF)
        {
          connHandle = numActive - 1;
        }

        // Red LED indicates connection
        PIN_setOutputValue(ledPinHandle, Board_RLED, 1);

      }
      break;

    case GAPROLE_WAITING:
      {
        SimpleSerialSocketServer_freeAttRsp(bleNotConnected);

        // Clear connHandle
        connHandle = 0xFFFF;

        // "Disconnect" the stream as well
        SimpleStreamServer_disconnectStream();
      }
      break;

    case GAPROLE_WAITING_AFTER_TIMEOUT:
      {
        SimpleSerialSocketServer_freeAttRsp(bleNotConnected);

        // Clear connHandle
        connHandle = 0xFFFF;

        // "Disconnect" the stream as well
        SimpleStreamServer_disconnectStream();
```

```
        // Indicate disconnection by turning of the LEDs
        PIN_setOutputValue(ledPinHandle, Board_RLED, 0);
        PIN_setOutputValue(ledPinHandle, Board_GLED, 0);
      }
      break;

    default:
      // Default case
      break;
  }

}

/*********************************************************************
***
 * @fn      SimpleSerialSocketServer_connEvtCB
 *
 * @brief   Connection event callback.
 *
 * @param pReport pointer to connection event report
 */
static void SimpleSerialSocketServer_connEvtCB(Gap_ConnEventRpt_t
*pReport)
{
  // Enqueue the event for processing in the app context.
  if( SimpleSerialSocketServer_enqueueMsg(SSSS_CONN_EVT, 0
,(uint8_t *) pReport, NULL) == FALSE)
  {
    ICall_free(pReport);
  }
}

/*********************************************************************
***
 *
 * @brief   Creates a message and puts the message in RTOS queue.
 *
 * @param   event - message event.
 * @param   state - message state.
 * @param   pData - message data pointer.
 * @param   arg0  - generic argument.
 *
 * @return  TRUE or FALSE
 */
```

```
static uint8_t SimpleSerialSocketServer_enqueueMsg(uint8_t event,
uint8_t state,
                                                   uint8_t *pData, uint16_t
arg0)
{
  ssssEvt_t *pMsg = ICall_malloc(sizeof(ssssEvt_t));

  // Create dynamic pointer to message.
  if (pMsg)
  {
    pMsg->hdr.event = event;
    pMsg->hdr.state = state;
    pMsg->pData = pData;
    pMsg->arg0  = arg0;

    // Enqueue the message.
    return Util_enqueueMsg(appMsgQueue, syncEvent, (uint8_t
*)pMsg);
  }

  return FALSE;
}

/*********************************************************************
***
*********************************************************************
**/
```