POLITECNICO DI TORINO

MASTER's Degree in
MECHATRONIC ENGINEERING

MASTER's Degree Thesis

Firmware Implementation of a Frequency Response Analyzer for Battery Spectroscopy

Supervisor

Prof. BARTOLOMEO MONTRUCCIO

Candidate

SERGIO ANDRES ORTEGA PUENTE

Academic Year

2020-2021

Abstract

<< The proposal of BAT-MAN is to make significant technologic innovations (especially relative to the techniques of estimation and diagnostics), realizing at the same time a product idea (realizing a prototype) that, on one hand can offer immediate and large-scale feedback on the solutions developed, and on the other can act as a forerunner to a series of applications based on the same technologies, either in areas closely related to accumulation systems, or in areas where advanced diagnostic and estimation techniques can bring a significant added value. >> [1]

This thesis takes a step further in the development process of the industrial project BAT-MAN (Battery Management Device). Here, a new approach is taken to develop the firmware and software architecture using basic software to achieve a communication between the microcontroller abstraction layer and the application. BAT-MAN is a technology that provides State of Health estimation of a generic storage system (battery), on-line and real-time.

This study uses the CC2640R2F development board from Texas Instruments for running the developed firmware. An ADC channel reads periodically an external input signal. Read values represent the voltages in the time domain. Those values are then transformed into the frequency domain for the computation of the Power Spectrum. Lastly Standard Bluetooth Low Energy is used to transmit periodically the power spectrum of the signal to a mobile application for an easy visualization of the results.

It is concluded that, with the setup used, it is possible to study the spectral density of a signal and obtain relevant information required to apply electrochemical impedance spectroscopy techniques as the center of the BAT-MAN project for battery SoC and SoH estimation.

Acknowledgments

I cannot feel enough to thank the support that my family has given me throughout the training process through which I have passed these years. Especially my parents who have tried immensely to get me where I am right now. I am who I am because of them.

I thank my supervisor Giovanni Guida, who placed his trust in me and accepted me to carry out this thesis project. He was always available to clarify doubts and I learned a lot from him during the development of this project.

I thank my friends in Colombia and the new friends I made in Italy for all the moments we have shared. Thanks to Cesar Sierra for being a good friend. Thanks to my girlfriend, Mayra Bonfante, who has stood by my side supporting everything I do for over six years.

This achievement is not only mine, it belongs to everyone who was part of this adventure, and to each of you, I thank you.

Table of Content

Index of Figures

1. Introduction

Nowadays Energy Storage Devices (ESD) are a solution to a lot of problems and challenges in our society. In this sense, the participation of batteries stands out. Due to the relevance of these devices in the frame of the world market and the characteristics of the systems where they are used, knowing how these components behave in their applications is essential. For example, it is not possible to directly measure the energy they keep at a specific time, but it is possible to estimate that value from indirect measurements. The concept of accumulated energy in these devices is called State of Charge (SoC). It is also necessary to be aware of the State of Health (SoH), which is related to the maximum amount of load that the device can store, relative to the maximum conserved at the beginning of its life (BoL). Furthermore, the SoH is also related to how to determine the end of the battery life (EoL), which is a factor related to the efficiency of the battery.

The SoH of batteries is one of the more complex characteristics to estimate due to a number of factors. The variation of this phenomena in time is relatively short if compared with, for example, its SoC. Hence, having a larger volume of data to study the phenomena itself represents a complication. Aspects such as charge-discharge rate (C-rate), cell environment temperature, own temperature of the cell in operation, variably affect the SoH. In addition, the very definition of the SoH is related to the maximum stored charge, which cannot be measured, until a complete discharge of the cell, so the process of estimating this variable becomes somewhat more complicated to obtain. [2] [3]

Among the tools and techniques used for SoH estimation, the use of electrochemical impedance spectroscopy is highlighted (EIS), which consists of making small disturbances in a range of different discrete frequencies on the battery under study, to know the impedance resulting from each disturbance carried out, often using a frequency response analyzer, and thus have an idea of the behavior of the battery due to different stimuli. In this way, the response at a specific time in the life of the battery can be related to others in which the degradation of the battery itself occurs. [4]

1.1. Objective

The main objective is to make use of some of the Nonlinear Frequency Response Analysis (NFRA) techniques in order to set the bases for a new, low-cost, product. The actual implementation of the estimation of SoH and SoC of a battery is outside the scope of this project.

In order to meet the project goals, basic software is to be established, allocating algorithms capable of supporting the system requirements for the (also to be developed) embedded firmware. Entering into details, the possibility of performing spectral analysis on an input signal using an embedded system is to be studied whilst taking into account code complexity and memory management.

Finally, a mobile application is also to be developed. This may be useful for testing and data analysis and can also serve the purpose of a human-machine interface for future applications. The communication between the embedded system and the mobile shall use IoT data protocols (most probably BluetoothLE).

2. State of the Art

SoC and SoH estimation, each present complex and different challenges. Compared to the SoC estimation, which has progressed substantially to date, the SoH estimation is at a less mature stage. For this reason, it is important to know the state of the art of the different techniques and algorithms associated with estimating SoH.

At least two definitions of SoH are recognized in the literature. These definitions are related to the main factors that represent part of the degradation of a battery. These factors are the decay of the battery capacity (2.1) and the increase of the internal impedance of the battery (2.2).

$$SOH = \frac{Q_{Current}}{Q_{Nominal}} * 100 \tag{2.1}$$

$$SOH = \frac{R_{EOL} - R_{Current}}{R_{EOL} - R_{Nominal}} * 100 \tag{2.2}$$

These indices are not necessarily proportionally related, but it is easy to see how they relate to constants at different stages of the battery life, for instance, the nominal capacity at BoL and the value of the internal resistance at the EoL are needed, for which it is necessary to perform a complete charge-discharge cycle to a battery, which is an extensive and less practical process.

Capacity estimation techniques can be divided into two large groups: the electrochemical models and the equivalent RC circuit model. Electrochemical models allow the chemical reactions that take place inside the cell to be modeled employing partial derivative equations. Equivalent RC circuit models allow a generic model of the dynamics that occur in cells, using circuit elements such as resistors, capacitors, and inductances. In particular, RC circuit models are expressed as state-space models, which become more complex as the number of RC elements increases, and in the same way, the computational calculation also increases, making the implementation more complicated in real-time.

The internal resistance estimation techniques are generally divided into two groups. The first concentrates on studying the voltage drop that the battery presents to a certain current disturbance, which is directly related to the internal resistance, by Ohm's law. In this way, for specific states of charge, the voltage drop can be compared, comparing it with previous events in time, considering equality of conditions for temperature and current. Said evolution can be analyzed to determine the level of degradation present, but also the conditions required for the execution of the method can hinder its practical implementation. On the other hand, there are models based on electrochemical impedance spectroscopy measurements, which, in turn, allow us to determine part of the internal dynamics of the battery, using a small current or voltage stimulus.

2.1. Electrochemical Impedance Spectroscopy

Electrochemical impedance spectroscopy (EIS) is a non-destructive technique which is proven to provide a considerable amount of information in a relatively short space of time, while preserving the integrity of the battery [5].

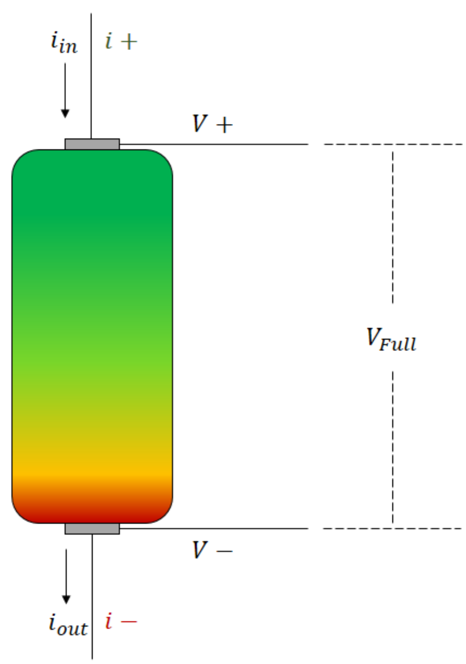

Figure 1. Wiring configuration for impedance measurement on batteries.

The EIS method can be performed both with a small excitation voltage (Potentiostatic) as with a small signal of excitation current (Galvanostatic). Generally, the excitation signals are small ac sinusoidal signals, usually in the mHz–MHz range [6].

Implementing a EIS measurement system in real-time is complex, apart from the cost of the necessary equipment, the time required to carry out measurements in a wide spectrum and with a large number of measurements at low frequencies, which naturally extend the measurement time, are particular problems that arise. For this, studies agree that frequencies between 50 Hz and 500 Hz are optimal for EIS. This allows estimating more practically the variables of interest, without having to measure such a wide range.

Batteries show a non-linear relationship between current and voltage, however, the EIS method applies to linear systems. For this reason, to find the frequency response in a battery, a small excitation signal is applied such that the entire non-linear current/voltage curve shows a pseudo-linear behavior, EIS usually is carried out with a small potentiostatic signal amplitude of 10 mV. In this small linear region, the non-linear properties of the battery can be approximated as a linear current/voltage curve. If the input signal is not small enough the system shows the non-linear characteristics and the steady-state condition is not reached. However, in the steady-state condition, the frequency of the small ac signal can be modified and the response can be saved to be analyzed and the complex impedance calculated.

There are two ways to graphically represent the impedance results obtained from an EIS test: Nyquist diagrams. Which is the most widely used representation system and the information obtained from it is based on the form that the spectra take. Bode diagrams, where the logarithm of the impedance module and the phase shift are represented as a function of the logarithm of the frequency. The information obtained from this type of representation is aimed above all at behavior as a function of frequency.

Two experimental techniques are used to perform EIS: the frequency-domain technique and the time-domain technique. The frequency-domain technique is the most widely used method, possibly due to the availability of commercial equipment. The time-domain technique uses a frequency-rich disturbance signal where current and voltage data measured in the time domain are converted to the frequency domain by a suitable transformation algorithm.

2.2. Frequency Response Analyzer

Frequency response is the measure of the output spectrum of a system in response to a stimulus. Frequency response analysis measures the magnitude and phase of the output as a function of frequency. Frequency Response Analyzers measure the response characteristics in magnitude and phase with respect to the frequency of the device or system under test with high precision and resolution.

The frequency spectrum of a wave phenomenon (sound, light, or electromagnetic), superposition of waves of various frequencies, is a measure of the amplitude distribution of each frequency. The graph of intensity versus frequency of a particular wave is also called a frequency spectrum. On the ordinate axis, the level in dBm (decibel-milliwatts) of the spectral content of the signal is usually presented on a logarithmic scale. The abscissa axis represents the frequency, on a scale that is a function of the temporal separation and the number of samples captured.

Spectrum analyzer types can be separated by means of the methods used to obtain the spectrum of a signal. There are swept-tuned, also called analog and fast Fourier transform (FFT) based spectrum analyzers.

An analog spectrum analyzer can be considered a frequency selective voltmeter, which responds to calibrated peaks in RMS values of the wave. Analog analyzers use a variable frequency band-pass filter whose center frequency is automatically tuned within a fixed range. A filter bank or a superheterodyne receiver can also be used where the local oscilloscope sweeps a range of frequencies.

A digital spectrum analyzer uses a mathematical process that transforms a signal into its spectral components. This analysis can be carried out for small time intervals, or less frequently for long intervals, or even spectral analysis of a given function can be carried out. Furthermore, the Fourier transform of a function not only allows a spectral decomposition of the formants of an oscillatory wave or signal but with the spectrum generated by the Fourier analysis it is even possible to reconstruct or synthesize the original function by means of the inverse transform. In order to do that, the transform not only contains information about the intensity of a certain frequency but also its phase. This information can be represented as a two-dimensional vector or as a complex number. In graphical representations, often only the modulus squared of that number is represented, and the resulting graph is known as a power spectrum or power spectral density.

Specialized equipment is responsible for carrying out both the signal generation and reception processes, as well as their processing and generation of graphics in such a way that comparison and diagnosis can be made on them. However, it is not easy to acquire equipment, in practice, due to its high cost. For this reason, with the understanding of the process of its operation, it is proposed to carry out a simile to said analysis, trying to approach an evaluation of functional frequency response analysis.

2.3. Devices Available in the Market

By studying the already existing devices that can be found in the market it is possible to study some of the features used in order to create a proposal for this specific project.

- **1260A Impedance Analyzer**

This device is positioned by many as the most popular impedance analyzer. It uses the techniques form frequency response analysis to measures the output spectrum of a system relative to a stimulus. Having a frequency range spanning 10 µHz to 32 MHz, 1260A delivers excellent coverage for virtually all chemical and molecular mechanisms - all in a single tool.

The signal generator is able to output only sine waveforms for both voltages and currents. The voltage generator is able to create 3 V RMS (≤10 MHz) or 1 V RMS (>10 MHz) and a maximum DC voltage equal to ±40.95 V. The current generator is a unique feature that Is able to generate 0 mA RMS (≤10 MHz), 20 mA RMS (>10 MHz).

The measurement system consists of 3 channels, 1 for current and 2 for voltages, able to measure up to 5 V peaks and 100 mA peak signals.

Presenting a great variety of functionalities, this device can be considered as an all-rounded for a big range of applications, in fact it is highly reliable and accurate. The only drawback this device presents is that it is not able to perform harmonic analysis, furthermore it low portability may be unfitting for some applications.

Figure 2. 1260A Impedance Analyer [7]

- **EmStat4S, USB Powered Potentiostat/Galvanostat with EIS**

EmStat4S brings portability to the table. It comes in two different versions regarding the current output ranges but, for both of them, an EIS/FRA can be configured up to a maximum frequency of 200 kHz.

Additionally, it comes with a software called PSTrace for windows machines. The device can hold up to 15 million data points that can be browsed and transferred to a computer by using the software (USB driven).

The measuring system consists of 2 mm banana connectors for working, counter, reference electrode and ground, allowing it to sample one signal at a time. It is able to carry out potentiostatic EIS by applying linear sweep popentiometry.

Whilst having its own advantages, the device loses on accuracy of measure. It can reach up to 10% accuracy for some frequency ranges this can be attributed to conditions on connections, the environment, and the device under testing.

Figure 3. EmStat4S with EIS [8]

- **Vector Network Analyzer - Bode 100**

Vector Network Analyzer Bode 100 presents itself as a middle ground to the already revised devices. It's high accuracy and great price-performance ratio, the Bode 100 is the best choice for industrial applications as well as research and educational labs. It excels as a frequency response analyzer while also being able to operate as a gain/phase meter or an impedance analyzer.

Presenting a signal generator and two channels for acquiring data it is able to compute the complex gain of active and passive circuits among others with a wide frequency range from 1 Hz to 50 MHz.

The device measurement system can be configured for a fixed frequency or a range or frequencies for its analysis. Furthermore, it has a very sensitive input sensor (superheterodyne receiver with 24 bit ADCs) which allows for high accuracy.

The Bode 100 is controlled via the Bode Analyzer Suite software, a GUI for Windows PCs.

Figure 4. Bode 100 – Vector Network Analyzer [9]

For the purpose of this project there are a lot of functionalities that can be studied from these devices in order to create a model that best fits the requirements. The main concern is to have simple functions in order to complete a specific task. Entering more into details, in one hand there is no need for having such wide ranges of frequencies for performing the analysis in the case studied, in the other accuracy is something to take into account.

3. Design and Implementation of a Proposed FRA

The proposal is based on the principles of FRA, that is, it seeks to use the methodologies of an FRA (technical analysis and structuring) with the premise of obtaining similar analysis results with affordable resources.

3.1. The Approach

The given context for a frequency response analyzer on the earlier chapters allows the definition of a coherent and easy-to-understand approach, as illustrated in figure 2.

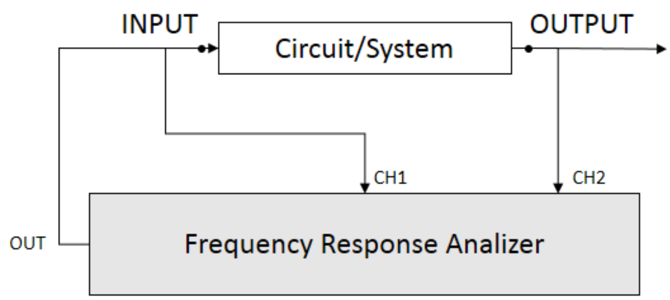

Figure 5. Connecting a FRA to a device under testing.

Two important components are highlighted for the modelling of a frequency response analyzer: a signal generator for the output signal and a receiver for both channels. A computer is also needed for the implementation of the Fourier transform algorithm. This method permits the determination of the frequency domain behavior for the device under testing.

3.2. The input signal

A great majority of the frequency response analyzers in the market utilize a sweep oscillator in order to generate the base input signal through a range of frequencies. Knowing the properties of the signals used to perform frequency response analysis and to simplify and give accessibility to the project, without losing reliability in the analysis processes, a PWM signal will be generated using an Arduino Uno board.

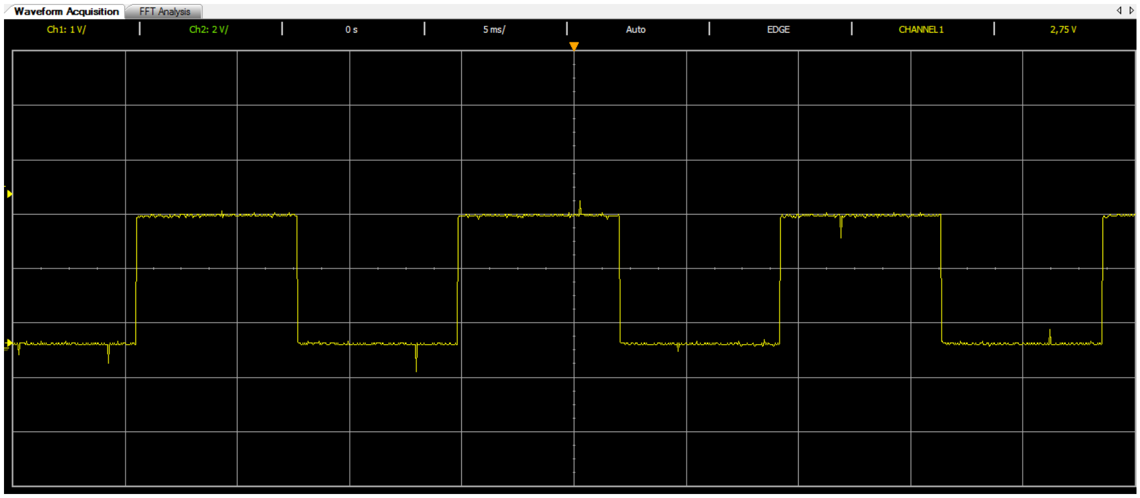

Figure 6. Example of a PWM signal generated by Arduino board.

Pulse width modulation is also a widely studied topic, which makes it possible to study the spectral density of the PWM signal more easily since every possible waveform and function in existence can be generated by just adding together sets of sine waves.

Choosing a PWM signal generated by an Arduino board allows a reliable and easy way of performing analysis on a wide range of frequency values. This can be achieved by simply changing the output frequency of the Arduino board pins or, even easier, changing the duty-cicle of the PWM signal. Figure 6 show an example of a PWM signal in the time domain generated by the Arduino with a frequency of 60.04 Hz and 50 % duty-cycle.

The spectrum of a PWM signal contains contributions from an infinite number of equally spaced frequencies as shown in Figure 7. These are the resonating frequencies also known as harmonics that make up the original signal. As this signal has frequency contributions in a wide range, its usage can facilitate the study of the spectrum of the device under testing and make it faster with respect to the same system being studied using pure sinusoidal inputs which only have one frequency contribution.

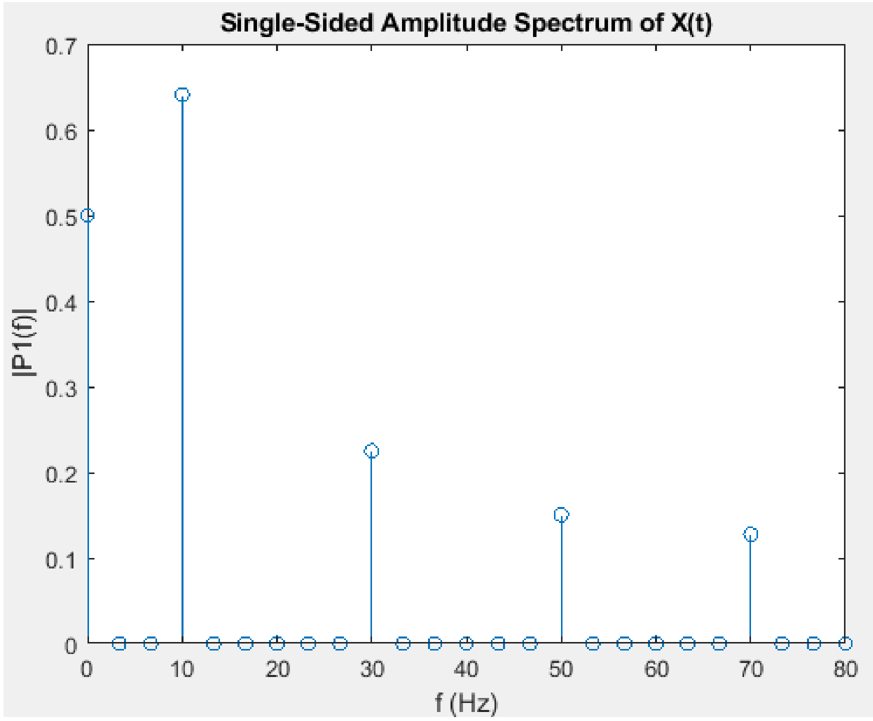

Figure 7. Example of a frequency spectrum of a PWM signal.

3.3. The Data Acquisition Device

The receiver is normally used in combination with an analog-to-digital converter along with a computer for spectra computation. They focus on acquiring and sample the data from the output of the device under testing in order to apply Fourier transform algorithm for the computation of the spectral density. CC2640R2F development board form Texas Instruments can, with no trouble, perform both tasks by making use of the built in ADC channels and timers.

According to the device datasheet, a 12-bit ADC with an 8-channel analog mux is available in order to perform the data acquisition. Correct data acquisition is relevant to be able to apply the Fourier transform algorithm, so calibration of the conversion parameters was carried out.

The most important feature to highlight of the microcontroller is the clock that controls the data acquisition time intervals, also known as the sampling period. This parameter determines what is known as the sampling frequency and is of great importance for the application of the discrete Fourier transform algorithm.

Clock characterization tests were carried out on the development board to evaluate the performance of the acquisition channels at different sample rates. For this, a led was programmed in such a way that its status would alternate each time data was acquired correctly through the ADC channel. The main idea was to check if the real sampling frequency matched the value input in software. Linear behavior was found for frequencies below 2 kHz. Figure 8 illustrates the signal from the toggling led for a sampling rate of 2 kHz, each rising/falling edge indicates that a sample was correctly acquired.

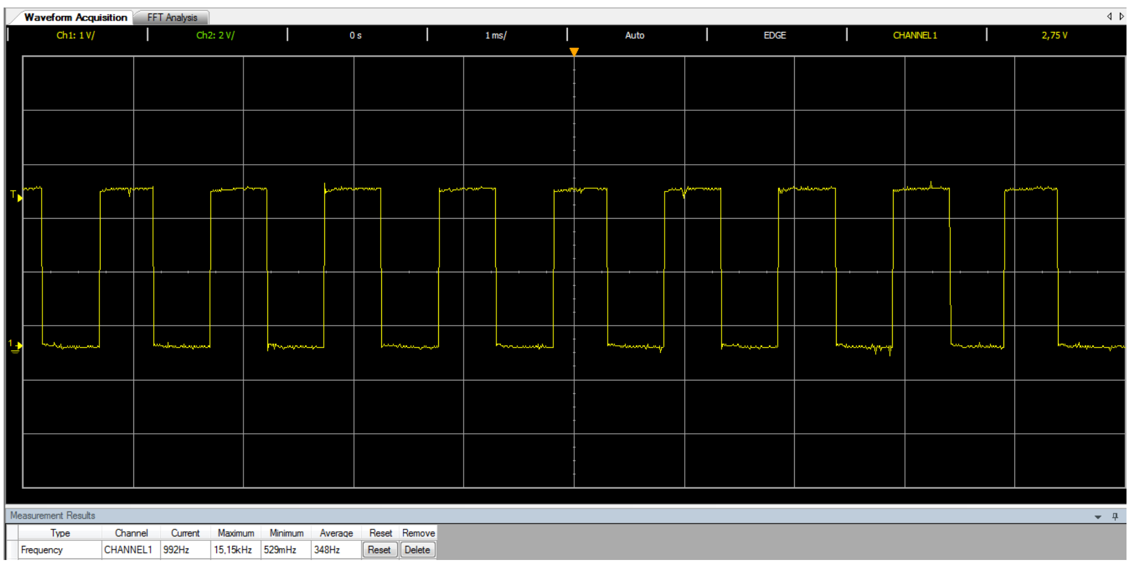

Figure 8. Sampling frequency test.

The computation of the power spectrum uses Fourier transform algorithms. This is a widely studied topic and is outside the scope of this paper. The only important thing to note is that the discrete Fourier transform algorithm will be applied due to simplicity, although it is not the most efficient algorithm in terms of computational speed.

3.4. Fourier Transform Implementation

One thing to note is that there is a great deal of very intricate mathematics surrounding the Fourier transform but for simply developing an understanding of how the technique works and what the Fourier transform is actually doing that is not particularly needed.

$$\hat{f}(k) = \int_{-\infty}^{\infty} f(x) exp^{\frac{-j2\pi}{N}kn}\ dx \tag{3.1}$$

The expression in (3.1) illustrates the continuous Fourier transform, this way of describing the Fourier transform is usually the one displayed in literature. Mathematically this is the best way of expressing it but it may lead to confusion because it is not necessarily intrinsically obvious to observe what is happening.

In this expression there is $f(x)$, which is the input signal, and there is also a complex exponential term. Both of this components are multiplied together this product is then integrated over all of X for any particular given value of K. in other words, for every value of K the signal is multiplied by the complex exponential for that value of K and this result is integrated between minus infinity and infinity repeating this for however many values of K are desired.

It is now easy to see how K, also known as wave number, represents the frequencies of interest. It is possible to pick specific number of frequencies components or a range of values of K to have a look to how the frequency content of the signal varies. Of course in this continuous form the chosen K can be continuous in some way which makes this expression of no use for programming purposes.

$$X_k = \sum_{n=0}^{N-1} x_n exp^{\frac{-j2\pi}{N}kn} \tag{3.2}$$

Equation (3.2) describes the discrete version of the Fourier transform, where the continuous signal $f(x)$ is replaced by x_n. The signal is defined having N samples and x_n describes each individual sample. The exponential term remains pretty much exactly the same.

The discrete Fourier transform takes the product of each sample with the corresponding sample of the complex exponential function repeating this for every value of n between 0 and $N-1$. By adding all those values together gives the Fourier transform coefficient for that particular value of K.

The key to understanding how the Fourier transform works is to understand what is really happening in terms of the complex exponential term. This exponential term can be conveniently expanded, obtaining a mathematically equivalent expression that is much easier to read.

$$X_k = \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi}{N}kn\right) - \mathrm{j}\sin\left(\frac{2\pi}{N}kn\right)\right] \tag{3.3}$$

Equation (3.3) shown how the discrete Fourier transform of a signal is composed of two parts. The first is obtained by adding together all the products of the sampled signal to the corresponding sample of the cosine term. The second part is obtained similarly by adding together all the products of the same sampled signal to the corresponding sample of the sine wave. This later term is also multiplied by the negative imaginary number -j. These two parts represent, respectively, the real and the imaginary components of the discrete Fourier transform which is represented as a complex number for each term X_k.

This way of describing the discrete Fourier transform makes the code implementation really easy. The things to take into account while doing this are:

- Determine the number of samples.
- Define K in order to evaluate as much values of the Fourier transform as there are samples of the signals (For example, a thousand samples signal is evaluated a thousand K number of times).

- Allocate memory for the internal cumulative sums of the real and imaginary parts as the algorithms loops through.
- Create a nested loop. This is because for every value of K the algorithms need to loop through all n samples, evaluating the real and imaginary parts and updating the cumulative sum.

Each time the inner loop ends it is imperative to reset the cumulative sum to zero since a new Kth component of the Fourier transform is to be computed. Also, when displaying the values of X_k it is a good idea to normalize by the number of samples otherwise the values of the Fourier transform will depend on the number of samples.

This implementation of the algorithm is not really for having the most efficient way possible but to have a better understanding of what the discrete Fourier transform works. In fact there is a version of the discrete Fourier transform called fast Fourier transform which is indeed faster and more computationally efficient compared to evaluating the discrete Fourier transform in this way.

4. Software Development

4.1. RTOS

A real-time operating system (RTOS) is an operating system used to facilitate multitasking and integration of tasks in time-constrained and resource-constrained designs, as is often the case in embedded systems. Furthermore, the term "real-time" indicates predictability/determinism in runtime rather than raw speed, so an RTOS can usually be shown to satisfy hard real-time requirements due to its determinism.

Applications can always be written in bare metal form, but as the complexity of the code increases, having some kind of structure will help to manage the different parts of the application, keeping them separate. By separating functions into different tasks, new functions can be easily added without breaking others, as long as the new function does not overload shared resources, such as CPU and peripherals. Development without RTOS will typically be in a large infinite loop where all functions are part of the loop. A change to any feature within the loop will impact other features, making the software difficult to modify and maintain.

An RTOS provides features such as scheduler, tasks, and inter-task communication RTOS objects, as well as communication stacks and drivers. It enables developers to focus on the application layer of embedded software and design multitasking software easily and quickly. However, like any other tool, it must be used properly to add value. To create safe and efficient embedded software, developers must know when to use the RTOS features and also how to configure them.

4.2. TI-RTOS

TI-RTOS is the operating environment developed by Texas Instruments for projects on CC2640R2F devices [10]. The TI-RTOS kernel operates as a real-time, preemptive, multi-threaded operating system with drivers, tools for synchronization and scheduling.

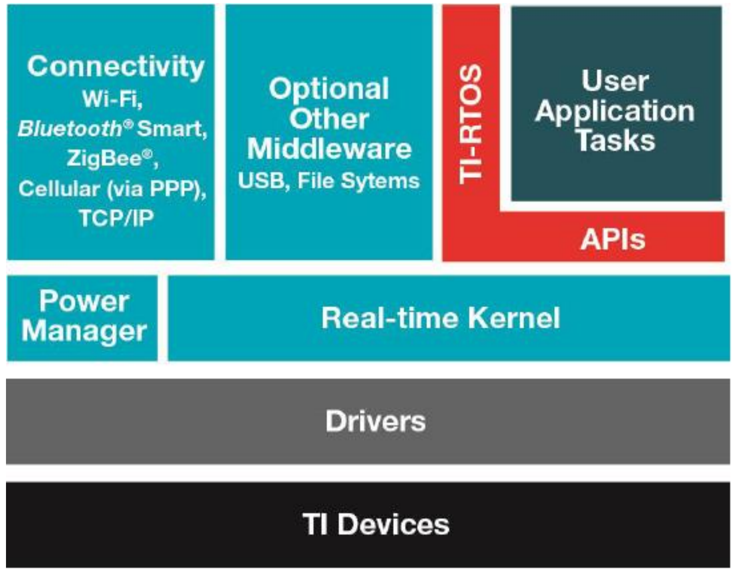

Figure 9. Structure of TI-RTOS components.

The main function of the kernel is the scheduler. The scheduler, which uses preemptive scheduling, is responsible for making sure the highest priority thread is running. Threads are independent functions that run in infinite loops, usually each responsible for one function [11]. The TI-RTOS kernel manages four distinct context levels of thread execution:

- Hwi or Hardware interrupt
- Swi or Software interrupt
- Tasks
- Idle task for Background functions

4.3. Software Implementation (Program Development)

Texas Instruments provides a series of example projects including BLE stack and task organization. It is much simpler to choose one of these projects to start with the development of the software than starting from scratch due to the complexity of the BLE stack implementation. The chosen example from the TI resource explorer is called “simple_pheripheral” which implements a simple Bluetooth low energy peripheral device with GATT services (more information on Low-Energy Bluetooth later). This example code needs to be adjusted and new modules were created in order to meet the project requirements.

4.3.1. Main Files and Functions

In this section the main source codes will be mentioned and a brief description on what task they perform and their content will be given. Note that for every source code (extension .c files) mentioned there exist a header file (extension .h files) containing declarations and macro definitions for their respective C file.

Main.c

Here, main in initialization of the operating system threads is performed. Additionally, for the realization of the application it was needed to initialize the thread running the application itself along with a clock instance and an event instance for managing the scheduling of the task.

A clock instance is a function that can be programmed to run after a certain amount of system ticks have passed. They can be one-shot functions or periodic functions and are started right after its creation. An event instance is a thread synchronization technique used to control the flow of threads during runtime.

During runtime, this code snippet runs only once before handing over the control of task scheduling to the RTOS kernel via BIOS_start() command.

Application.c

This source file contains the thread with all the functionalities for allowing the execution of the process. This file is of great importance because it contains the declaration of resources shared across all files (global variables).

The algorithm applying the data acquisition and the discrete Fourier transform is inside this source file. During runtime, the event instance controls whether the thread runs the acquisition or the discrete Fourier transform algorithms. The sampling frequency of the ADC channel is controlled by the first activation of the clock instance initialized on the main file which posts on the event START_AQUISITION. This event triggers the acquisition and re-starts the clock instance, this process is repeated until a number of predefined data have been acquired. When this happens, a post on the event START_DFT is performed which stops the data acquisition and starts the discrete Fourier transform and then power spectrum computation.

This thread is initialized with the same priority that the main thread in order not to interfere with BLE application and Bluetooth communication.

BATMAN_FT.c

The source code containing the main task loop and BLE callback functions. The task on this code snippet runs an infinite loop that receives messages from the BLE stack. It uses a module called Indirect Call Framework (ICall) which provides API's that allow the communication between the application and the BLE-Stack.

In order to achieve communication between application and BLE-Stack, the ICall module provides messaging and thread synchronization. BATMAN_FT.c features an event-driven task function. The task function enters an endless loop so that it is continuously processed as a separate task and does not run to completion. In this infinite loop, the task remains blocked and waits until the appropriate event flags indicate a new reason for processing. This event flags will be modified by BLE-Stack messages or by programmed code like hardware interrupts or callback functions.

4.3.2. Bluetooth Low Energy

BLE works using what is known as Generic Access Profile (GAP). GAP defines two roles: peripherals, which mobilize values through sensors and consume very little energy; central, which are devices that read/write data from/to the peripherals. Furthermore, GAT relies on two types of messages: advertisement and scan response requests.

For TI devices, BLE communication uses Generic Attribute Profile (GATT). GATT refers to the way peripherals will exchange data with central devices after a connection has been established. The association Bluetooth Special Interest Group describe how to implement GATT services and characteristics to achieve a certain application in documents called Profiles.

An Attribute is the smallest addressable unit of data used by GATT. All attributes are defined inside a table and they have a handle, a type (with a UUID) and a value.

Handle	UUID	Value
16 bits	16 or 128 bits	1 to 512 bytes.

Figure 10. Description of a GATT attribute.

A characteristic is an attribute with a handle, a type (a UUID that tells us that the attribute is a characteristic) and a value (the characteristic properties with a handle to the attribute value and so on). Inside the characteristic, you have an attribute that is the value of the characteristic and one or more descriptors that are themselves attributes.

Lastly, A Service is a collection of characteristics. The central element in a service is the Attribute Table which contains the information of attributes inside the definition of a service. A somewhat simplified graphic representation of an example of attribute table can be seen in Figure 11.

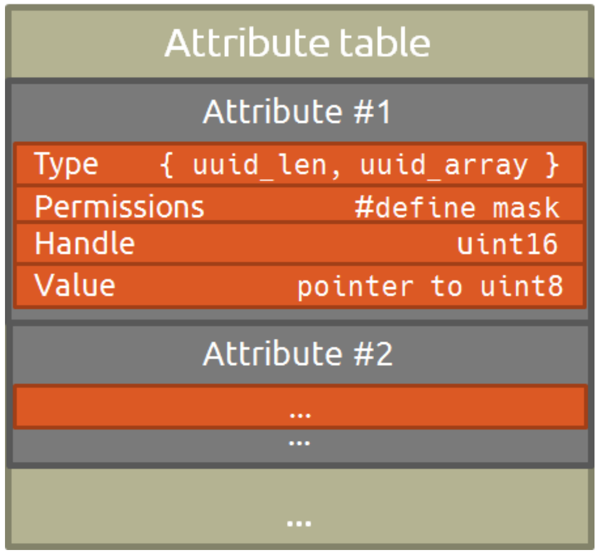

Figure 11. Example of an attribute table.

A Profile describes a collection of one or more services and how they can be used to allow an application. For this purpose a new profile was created called AppData.c by using the BLE profile generator provided by Texas Instruments. The information that this profile describes is shown in the following Figure.

NAME	UUID	LENGHT	PROPERTIES
APPDATA_APPDATAFS	0xAA01	4	GATT_PROP_READ
APPDATA_APPDATAFT	0xAA02	100	GATT_PROP_READ

Figure 12. Characteristics of project Service. (UUID: 0xAA00)

This service provide two characteristics that allow any connected device to only read their respective values as the user interface only displays the information of the power spectrum on a graphs and does not need to perform any configuration on the device.

The value of the APPDATA_APPDATAFT characteristic contains an array of twenty five floating point values. Each position of this array represents the magnitude of the power spectrum of the discretized input signal. As each floating point value occupies 4 bytes of memory, the total amount requires to store the value of this characteristic is 100 bytes.

When something is done to the Service over the air, the GATT Server will invoke the callback handlers. In this case there is only room for the Read Callback to be called, which send the characteristic value attribute. This value can be changed by the application task using the service's SetParameter function.

4.3.3. User Interface (Android App)

The idea of building a mobile application come from the need of visualizing the power spectrum which is the output of the algorithms implemented on the development board. To accomplish this in an easy manner the application will be developed using MTI App Inventor from Google Labs.

MIT App Inventor allows the use of extensions within the project that add functionalities to the application. BluetoothLE is an extension that adds Bluetooth Low Energy functionalities to the application. For BluetoothLE the UUID identifier that represents the device that intervenes in a connection with its roles, data exchange protocol, data type, etc., are represented in the services. These services are standardized. Likewise, the characteristics represent the data being exchanged. Obviously the peripherals should have software ready to receive these values and act accordingly.

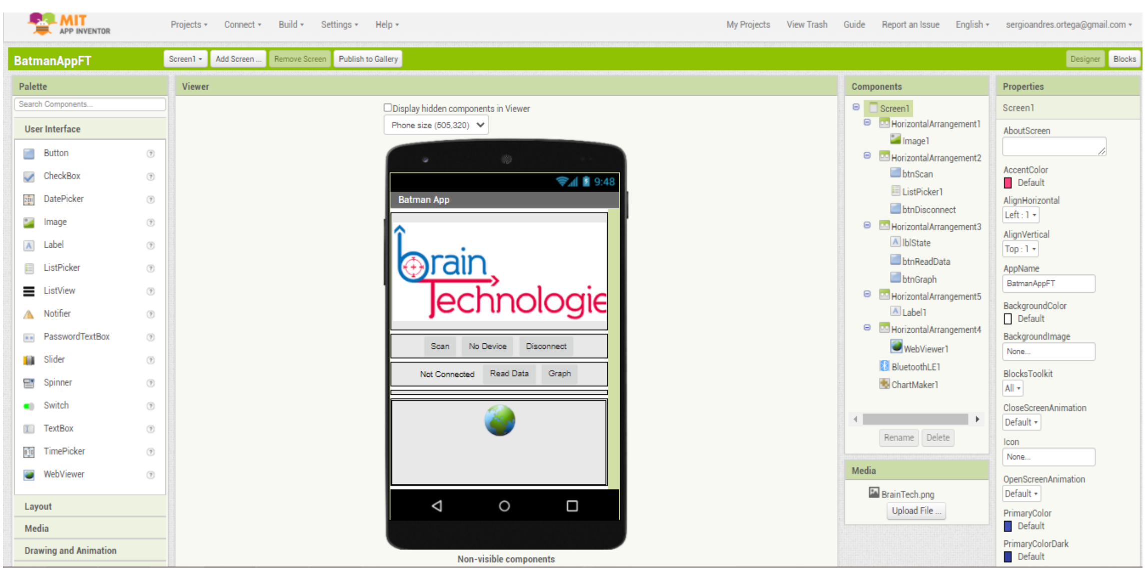

Figure 13. MIT App Invento designer view.

MIT App Inventor provides a palette with different components, in the designer view, that allows the creation of our visual interface, which will then be coded by using blocks, in the block view, to define how it behaves.

In this application, a Scan button is configured so that when pressed it will search for nearby devices. It will also generate a list of devices with which it is possible to initiate a connection. Through a list selector, an attempt will be made to establish said connection and, once connected, it will be possible to read the advertised data from the device. There will also be a disconnect button that finishes the connection between the devices.

The main blocks to be analyzed are those regarding the implementation of BLE processes and only those will be addressed in this discussion.

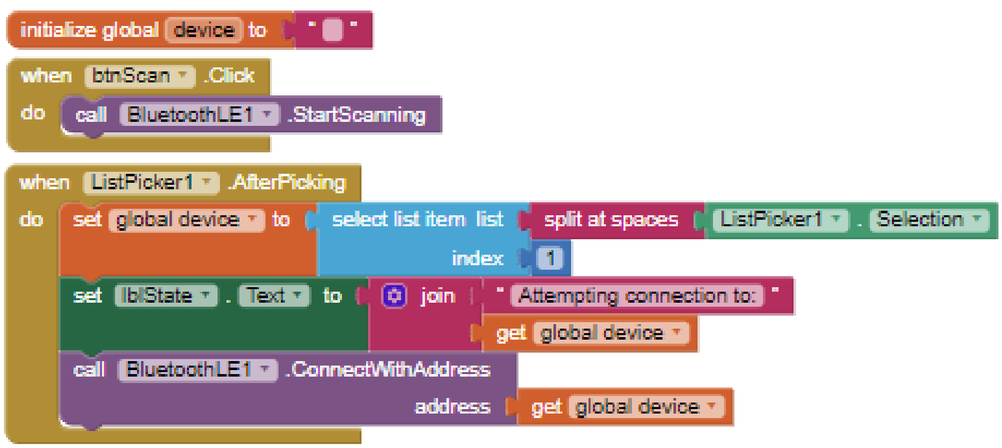

Figure 14. Scan and Connection blocks for mobile application.

The first block used is the StartScanning which starts scanning for Bluetooth Low Energy devices. Then, once the selection of a device occurs, the application need to call the ConnectWithAdress block, which connects to a specific Bluetooth low energy device if its Media Access Control (MAC) address is known. The MAC address can be easily obtained by splitting the selection from the List Picker (it contains both MAC address and device name) so that it can then be passed to the ConnectWithAdress block correctly.

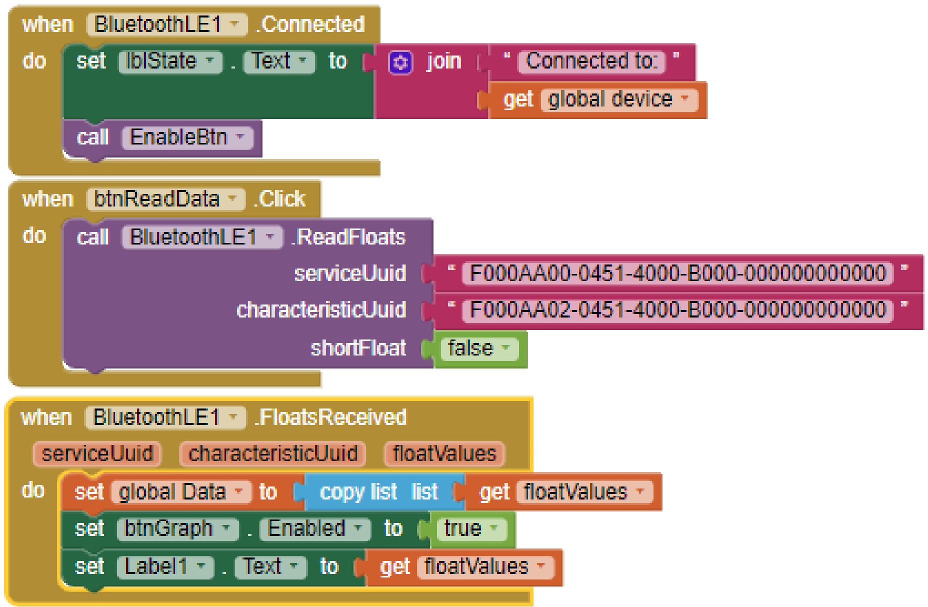

Figure 15. ReadFloats block from mobile application.

When the device is successfully connected the possibility to read the data from the device is enabled. ReadFloats Reads one or more IEEE 754 floating point numbers from a connected BluetoothLE device. Service Unique ID and Characteristic UniqueID are required. Here the UUID values from the created profile are introduced. The shortFloat parameter indicates whether the floats are either 16-bit half-precision floating point or 32-bit single precision floating point numbers.

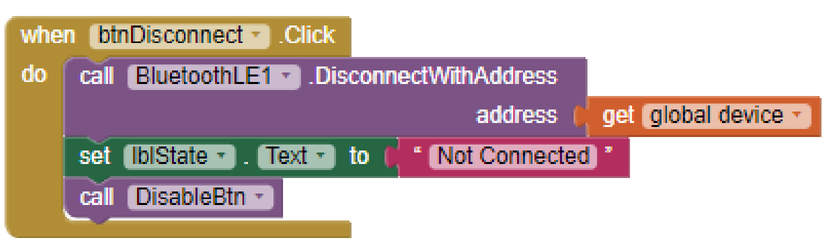

Figure 16. DisconnectWithAddress block from mobile application.

Lastly, there is a call to DiscconectWithAdress block when the Disconnect button is pressed. This Disconnects from a connected BluetoothLE device with the given address and terminates communication.

An additional extension is used for the displaying of the generated graphs. Chartmaker offers chart-making capabilities. Charts are displayed in a WebViewer using the visualization API for Google Charts. Data can be fed into each draw function in list form, with strings for labels and titles.

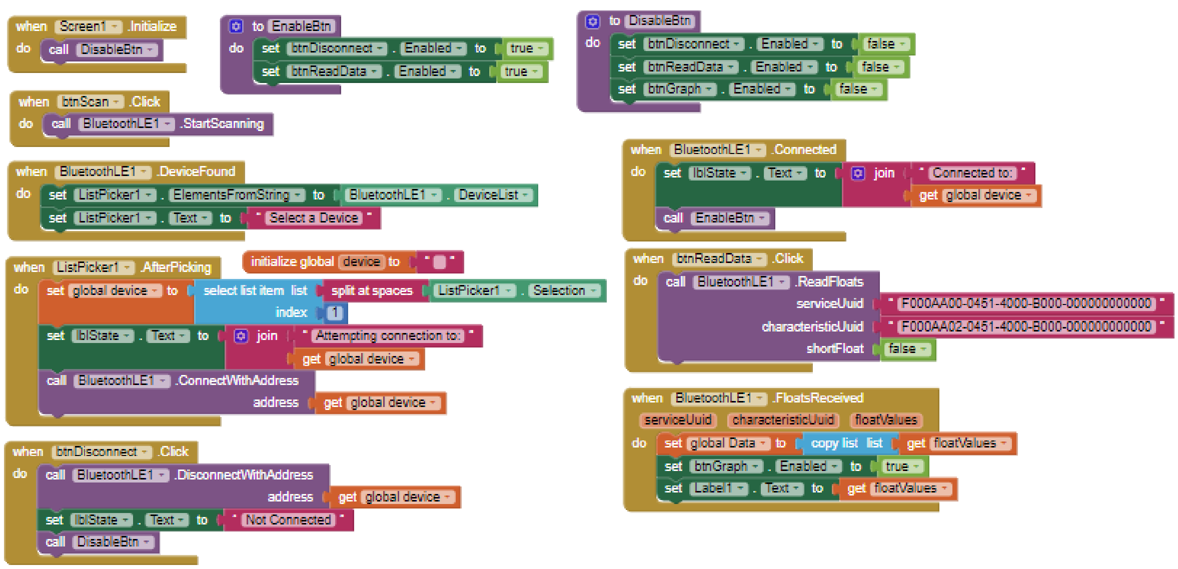

Figure 17. Complete block diagram for mobile application.

5. Tests and Results

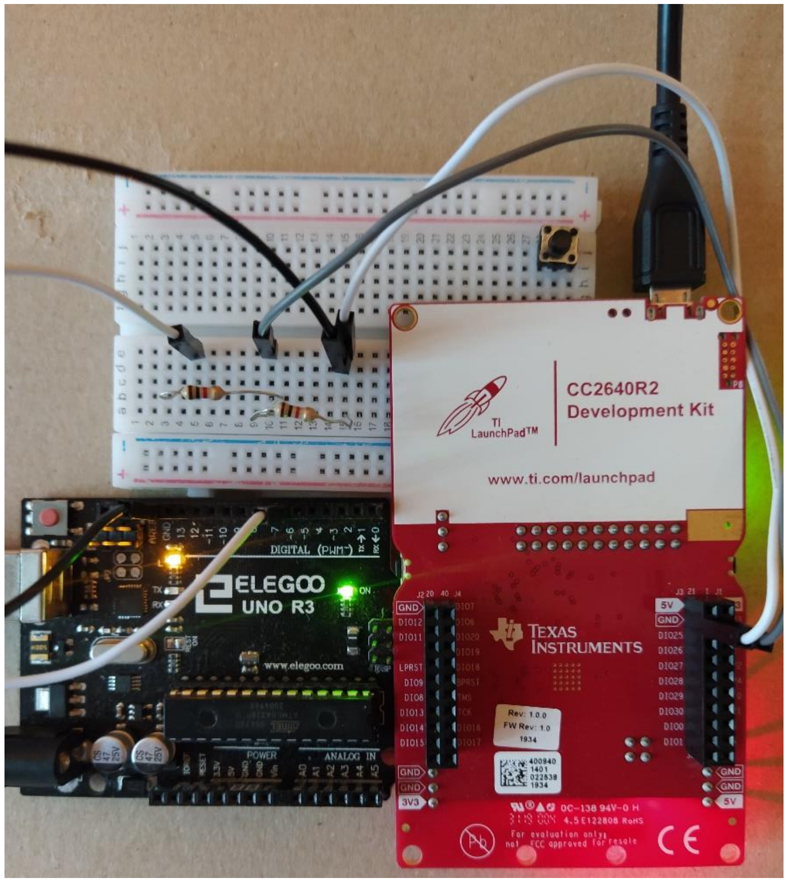

Figure 18. Testing equipment.

Tests were carried out throughout the entire development process in order to check for the proper functioning of the hardware and software.

As all the developed software architecture was applied on a development board. It was possible to perform code testing using the debug session provided by Texas Instruments on Code Composter Studio IDE. This allowed the supervision and control of some of the important variables during the process carried out by the microcontroller.

First, a signal with contributions at exactly two known frequencies was given as an input to the system. This can reveal any problems regarding the application of the discrete Fourier transform algorithm.

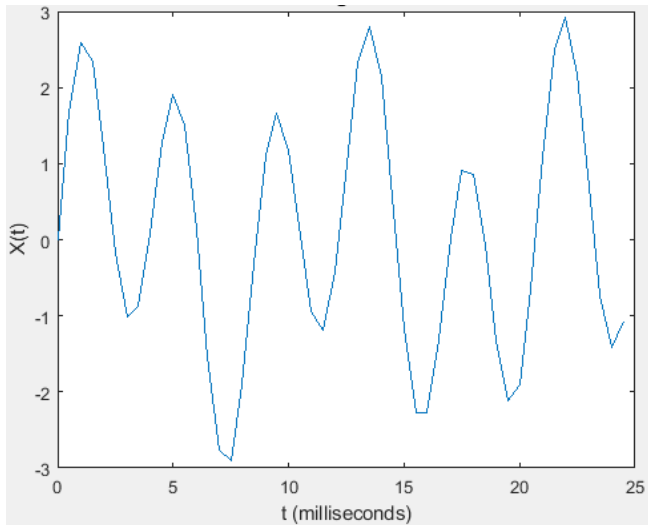

Figure 19. Input signal for discrete Fourier transform algorithm testing.

The resulting frequency spectrum shows the exact two contributions that make up the original signal at their respective frequencies. This proves a correct implementation of the discrete Fourier transform algorithm. The resulting spectrum can be observed using the mobile application as shown by Figure 20.

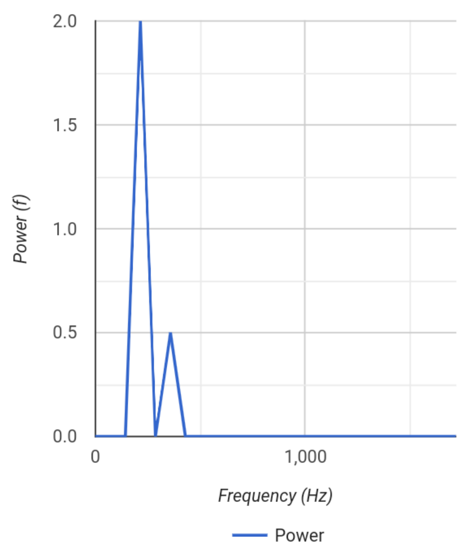

Figure 20. Power spectrum for the signal testing Fourier transform algorithm.

It is important to note that this is just one half of the power spectrum, which comprises only the magnitude. There is also a phase power spectrum but for display purposes this representation is easier to understand. Each resonating peak shows the amplitude and the frequency at with they contribute to the original signal.

Then, a selection of PWM signals with different frequencies was used as input to the system. Each of these signals was correctly measured with an oscilloscope as previously shown in section 3.2 of this work. The magnitude power spectrum of some of this selected frequencies are shown in the following figures.

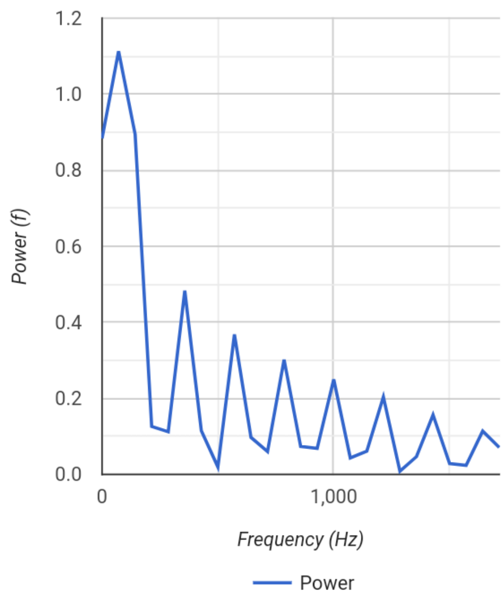

Figure 21. Power spectrum of a 60.01 Hz PWM signal.

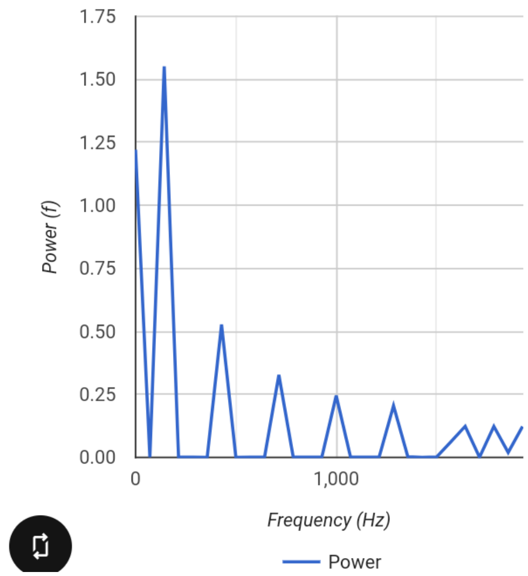

Figure 22. Power spectrum os a 71.42 Hz PWM signal.

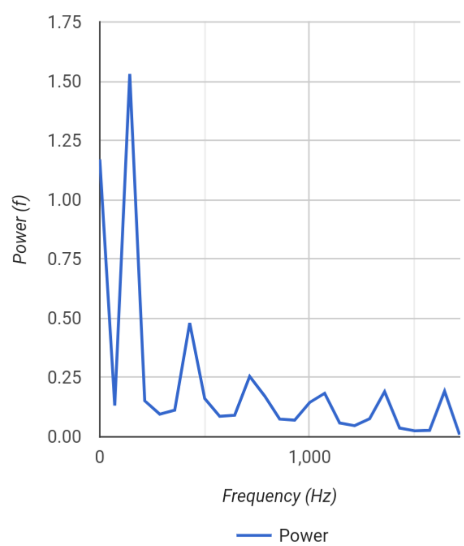

Figure 23. Power spectrum of a 83,33 Hz PWM signal.

It should be noted that all these signals have been sampled at a sampling frequency of 2 kHz, which means that the data collection is relatively fast as well as the processing time is short. In turn, it can be observed that efficiency is not lost when obtaining the frequency spectrum of the signals, giving the information necessary to recognize the type of signal supplied to the system.

6. Future Work

The discrete Fourier transform algorithm implemented takes the transform over the entire signal and relies on the assumption that the signal is stationary over its duration meaning that it doesn't see time varying frequency components and/or discontinuities on the signal. It might be useful to not only know the frequency content of the signal but also to get some information about when those particular frequencies occur.

The most obvious way to go about doing what previously stated is to window the signal which means simply to take short portions of the signal by multiplying it to a windowing function and take the Fourier transform of that short portions for all the signal and build those up. In practice this seems easy to implement, however doing this can bring up some other complications thus it is needed to evaluate whether the necessity of applying a window to the input signal is worth doing.

Regarding the data transmission, the sampling frequency of the device is set to 2 kHz which is more than enough for the purpose of this job but may be problematic for higher frequency input signals. Adding additional characteristics to the GATT service so that it is possible to change the sampling frequency of the device could be useful. By allowing this value to change, it is possible to find a better balance between accuracy a sampling time for any input signal. This can be easily implemented in two steps.

- Add a new characteristic for the service located at AppData.c profile. One way in which this can be achieved is by changing the properties of APPDATA_APPDATAFS characteristic to also allow for GATT_PROP_WRITE. In addition to this, new software has to be developed to properly manage this request.
- Add new blocks on the MIT App Investor environment for controlling the data transfer process. As it is the user who request for the data exchange, the user must send the value of the desired sampling frequency.

7. Conclusion

By performing simple analysis and choosing the correct equipment for the proposal of the project, it was possible to reach similar results to a commercial FRA that exist today, which have an extremely high cost due to the number of parameters that they implement and study.

The objectives of the project were achieved thanks to the use of the correct components for the system implementation. In fact, the PWM generated signal used as an input has numerous frequency contributions that will allow a further study on signal behavior. The chosen development board, CC2640R2F from Texas Instruments, suited the project necessities in term of computational power and specifications and was used for sampling, calculation and data transmission though BLE. Finally all this information was compacted to a graphical user interface in a designed mobile application.

All the efforts made during the trajectory of this project make it so that the final product is ready to be used in a battery without the need of major modifications. Therefore, SoC and SoH of the battery can be estimated using the core concept of Electrochemical Impedance Spectroscopy, where a pseudo Transfer Function in generated using both the input spectrum, which computation was already achieved in this work, and the output spectrum from the battery itself.

8. Appendix

8.1. Main.c

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

 @file  main.c

 @brief main entry of the BLE stack sample application.

 Group: WCS, BTS
 Target Device: cc2640r2


*************************************************************************
*****

 Copyright (c) 2013-2020, Texas Instruments Incorporated
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:

 *  Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 *  Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 *  Neither the name of Texas Instruments Incorporated nor the names of
    its contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS"
 AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS;
 OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


*************************************************************************
*****
```

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

/************************************************************************
*******
 * INCLUDES
 */
/* XDC module Headers */
#include <BATMAN_FT.h>
#include <xdc/std.h>
#include <xdc/runtime/System.h>

/* BIOS module Headers */
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/sysbios/knl/Event.h>

#include <ti/drivers/Board.h>
#include <ti/drivers/GPIO.h>
#include "Board.h"

#include <xdc/std.h>
#include <ti/sysbios/knl/Task.h>

#include <xdc/runtime/Error.h>

#include <ti/drivers/Power.h>
#include <ti/drivers/power/PowerCC26XX.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Clock.h>
#include <ti/display/Display.h>

#include <icall.h>
#include "hal_assert.h"
#include "bcomdef.h"
#include "peripheral.h"
#include <inc/hw_memmap.h>
#include <driverlib/vims.h>

#ifndef USE_DEFAULT_USER_CFG

#include "ble_user_config.h"

// BLE user defined configuration
#ifdef ICALL_JT
icall_userCfg_t user0Cfg = BLE_USER_CFG;
#else  /* ! ICALL_JT */
bleUserCfg_t user0Cfg = BLE_USER_CFG;
#endif /* ICALL_JT */

#endif // USE_DEFAULT_USER_CFG

#ifdef USE_FPGA
#include <inc/hw_prcm.h>
#endif // USE_FPGA
```

```
/***********************************************************************
*******
 * MACROS
 */

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

#if defined( USE_FPGA )
  #define RFC_MODE_BLE                  PRCM_RFCMODESEL_CURR_MODE1
  #define RFC_MODE_ANT                  PRCM_RFCMODESEL_CURR_MODE4
  #define RFC_MODE_EVERYTHING_BUT_ANT   PRCM_RFCMODESEL_CURR_MODE5
  #define RFC_MODE_EVERYTHING           PRCM_RFCMODESEL_CURR_MODE6
  //
  #define SET_RFC_BLE_MODE(mode) HWREG( PRCM_BASE + PRCM_O_RFCMODESEL ) =
(mode)
#endif // USE_FPGA

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

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

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

#ifdef CC1350_LAUNCHXL
#ifdef POWER_SAVING
// Power Notify Object for wake-up callbacks
Power_NotifyObj rFSwitchPowerNotifyObj;
static uint8_t rFSwitchNotifyCb(uint8_t eventType, uint32_t *eventArg,
                                uint32_t *clientArg);
#endif //POWER_SAVING

PIN_State  radCtrlState;
PIN_Config radCtrlCfg[] =
{
  Board_DIO1_RFSW   | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW  | PIN_PUSHPULL |
PIN_DRVSTR_MAX, /* RF SW Switch defaults to 2.4GHz path*/
  Board_DIO30_SWPWR | PIN_GPIO_OUTPUT_EN | PIN_GPIO_HIGH | PIN_PUSHPULL |
PIN_DRVSTR_MAX, /* Power to the RF Switch */
  PIN_TERMINATE
};
PIN_Handle radCtrlHandle;
#endif //CC1350_LAUNCHXL

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

```
 * EXTERNS
 */

extern void AssertHandler(uint8 assertCause, uint8 assertSubcause);

extern Display_Handle dispHandle;

//extern void myThread_create(void);

extern Void clk1Fxn(UArg arg0);

Clock_Struct clk1Struct;
Clock_Handle clk2Handle;

extern void taskFunction(UArg arg0, UArg arg1);

#define TASK_STACK_SIZE 512
uint8_t appTaskStack[TASK_STACK_SIZE];
Task_Struct task0;

Event_Handle event;
Event_Params eventParams;
Event_Struct structEvent;
/*************************************************************************
*******
 * @fn          Main
 *
 * @brief       Application Main
 *
 * input parameters
 *
 * @param       None.
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      None.
 */
int main()
{
#if defined( USE_FPGA )
  HWREG(PRCM_BASE + PRCM_O_PDCTL0) &= ~PRCM_PDCTL0_RFC_ON;
  HWREG(PRCM_BASE + PRCM_O_PDCTL1) &= ~PRCM_PDCTL1_RFC_ON;
#endif // USE_FPGA

  /* Register Application callback to trap asserts raised in the Stack */
  RegisterAssertCback(AssertHandler);

  Board_initGeneral();

#ifdef CC1350_LAUNCHXL
  // Enable 2.4GHz Radio
  radCtrlHandle = PIN_open(&radCtrlState, radCtrlCfg);

#ifdef POWER_SAVING
  Power_registerNotify(&rFSwitchPowerNotifyObj,
```

```
                       PowerCC26XX_ENTERING_STANDBY |
PowerCC26XX_AWAKE_STANDBY,
                       (Power_NotifyFxn) rFSwitchNotifyCb, NULL);
#endif //POWER_SAVING
#endif //CC1350_LAUNCHXL

#if defined( USE_FPGA )
  // set RFC mode to support BLE
  // Note: This must be done before the RF Core is released from reset!
  SET_RFC_BLE_MODE(RFC_MODE_BLE);
#endif // USE_FPGA

#ifdef CACHE_AS_RAM
  // retain cache during standby
  Power_setConstraint(PowerCC26XX_SB_VIMS_CACHE_RETAIN);
  Power_setConstraint(PowerCC26XX_NEED_FLASH_IN_IDLE);
#else
  // Enable iCache prefetching
  VIMSConfigure(VIMS_BASE, TRUE, TRUE);
  // Enable cache
  VIMSModeSet(VIMS_BASE, VIMS_MODE_ENABLED);
#endif //CACHE_AS_RAM

#if !defined( POWER_SAVING ) || defined( USE_FPGA )
  /* Set constraints for Standby, powerdown and idle mode */
  // PowerCC26XX_SB_DISALLOW may be redundant
  Power_setConstraint(PowerCC26XX_SB_DISALLOW);
  Power_setConstraint(PowerCC26XX_IDLE_PD_DISALLOW);
#endif // POWER_SAVING | USE_FPGA

#ifdef ICALL_JT
  /* Update User Configuration of the stack */
  user0Cfg.appServiceInfo->timerTickPeriod = Clock_tickPeriod;
  user0Cfg.appServiceInfo->timerMaxMillisecond  = ICall_getMaxMSecs();
#endif  /* ICALL_JT */
  /* Initialize ICall module */
  ICall_init();

  /* Start tasks of external images - Priority 5 */
  ICall_createRemoteTasks();

  /* Kick off profile - Priority 3 */
  GAPRole_createTask();

  SimplePeripheral_createTask();

  /* Call function to initialize adc threads */
  Board_init();
  GPIO_init();

  Clock_Params clkParams;
  Clock_Params_init(&clkParams);

  clkParams.period = 0;
  clkParams.startFlag = FALSE;

  /* Construct a one-shot Clock Instance */
```

```
  Clock_construct(&clk1Struct, (Clock_FuncPtr)clk1Fxn,
                        420/Clock_tickPeriod, &clkParams);  /*420 for
2k Hz, y = -0.00006x^2+0.971x-0.9383, should be 500*/
  clk2Handle = Clock_handle(&clk1Struct);
  Clock_start(clk2Handle);

  Event_Params_init(&eventParams);
  Event_construct(&structEvent, &eventParams);
  /* It's optional to store the handle */
  event = Event_handle(&structEvent);

  Task_Params taskParams;
  // Configure task
  Task_Params_init(&taskParams);
  taskParams.stack = appTaskStack;
  taskParams.stackSize = TASK_STACK_SIZE;
  taskParams.priority = 1;
  Task_construct(&task0, taskFunction, &taskParams, NULL);

  /* enable interrupts and start SYS/BIOS */
  BIOS_start();

  return 0;
}


/*************************************************************************
*******
 * @fn          AssertHandler
 *
 * @brief       This is the Application's callback handler for asserts
raised
 *              in the stack.  When EXT_HAL_ASSERT is defined in the
Stack
 *              project this function will be called when an assert is
raised,
 *              and can be used to observe or trap a violation from
expected
 *              behavior.
 *
 *              As an example, for Heap allocation failures the Stack
will raise
 *              HAL_ASSERT_CAUSE_OUT_OF_MEMORY as the assertCause and
 *              HAL_ASSERT_SUBCAUSE_NONE as the assertSubcause.  An
application
 *              developer could trap any malloc failure on the stack by
calling
 *              HAL_ASSERT_SPINLOCK under the matching case.
 *
 *              An application developer is encouraged to extend this
function
 *              for use by their own application.  To do this, add
hal_assert.c
 *              to your project workspace, the path to hal_assert.h (this
can
 *              be found on the stack side). Asserts are raised by
including
```

```
 *              hal_assert.h and using macro HAL_ASSERT(cause) to raise
an
 *              assert with argument assertCause.  the assertSubcause may
be
 *              optionally set by macro HAL_ASSERT_SET_SUBCAUSE(subCause)
prior
 *              to asserting the cause it describes. More information is
 *              available in hal_assert.h.
 *
 * input parameters
 *
 * @param       assertCause    - Assert cause as defined in hal_assert.h.
 * @param       assertSubcause - Optional assert subcause (see
hal_assert.h).
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      None.
 */
void AssertHandler(uint8 assertCause, uint8 assertSubcause)
{
#if !defined(Display_DISABLE_ALL)
  // Open the display if the app has not already done so
  if ( !dispHandle )
  {
    dispHandle = Display_open(Display_Type_LCD, NULL);
  }

  Display_print0(dispHandle, 0, 0, ">>>STACK ASSERT");
#endif // ! Display_DISABLE_ALL

  // check the assert cause
  switch (assertCause)
  {
    case HAL_ASSERT_CAUSE_OUT_OF_MEMORY:
#if !defined(Display_DISABLE_ALL)
      Display_print0(dispHandle, 0, 0, "***ERROR***");
      Display_print0(dispHandle, 2, 0, ">> OUT OF MEMORY!");
#endif // ! Display_DISABLE_ALL
      break;

    case HAL_ASSERT_CAUSE_INTERNAL_ERROR:
      // check the subcause
      if (assertSubcause == HAL_ASSERT_SUBCAUSE_FW_INERNAL_ERROR)
      {
#if !defined(Display_DISABLE_ALL)
        Display_print0(dispHandle, 0, 0, "***ERROR***");
        Display_print0(dispHandle, 2, 0, ">> INTERNAL FW ERROR!");
#endif // ! Display_DISABLE_ALL
      }
      else
      {
#if !defined(Display_DISABLE_ALL)
        Display_print0(dispHandle, 0, 0, "***ERROR***");
        Display_print0(dispHandle, 2, 0, ">> INTERNAL ERROR!");
```

```
#endif // ! Display_DISABLE_ALL
      }
      break;

    case HAL_ASSERT_CAUSE_ICALL_ABORT:
#if !defined(Display_DISABLE_ALL)
      Display_print0(dispHandle, 0, 0, "***ERROR***");
      Display_print0(dispHandle, 2, 0, ">> ICALL ABORT!");
#endif // ! Display_DISABLE_ALL
      HAL_ASSERT_SPINLOCK;
      break;

    case HAL_ASSERT_CAUSE_ICALL_TIMEOUT:
#if !defined(Display_DISABLE_ALL)
      Display_print0(dispHandle, 0, 0, "***ERROR***");
      Display_print0(dispHandle, 2, 0, ">> ICALL TIMEOUT!");
#endif // ! Display_DISABLE_ALL
      HAL_ASSERT_SPINLOCK;
      break;

    case HAL_ASSERT_CAUSE_WRONG_API_CALL:
#if !defined(Display_DISABLE_ALL)
      Display_print0(dispHandle, 0, 0, "***ERROR***");
      Display_print0(dispHandle, 2, 0, ">> WRONG API CALL!");
#endif // ! Display_DISABLE_ALL
      HAL_ASSERT_SPINLOCK;
      break;

  default:
#if !defined(Display_DISABLE_ALL)
      Display_print0(dispHandle, 0, 0, "***ERROR***");
      Display_print0(dispHandle, 2, 0, ">> DEFAULT SPINLOCK!");
#endif // ! Display_DISABLE_ALL
      HAL_ASSERT_SPINLOCK;
  }

  return;
}


/*************************************************************************
*******
 * @fn          smallErrorHook
 *
 * @brief       Error handler to be hooked into TI-RTOS.
 *
 * input parameters
 *
 * @param       eb - Pointer to Error Block.
 *
 * output parameters
 *
 * @param       None.
 *
 * @return      None.
 */
void smallErrorHook(Error_Block *eb)
```

```
{
  for (;;);
}

#if defined (CC1350_LAUNCHXL) && defined (POWER_SAVING)
/***********************************************************************
*******
 * @fn          rFSwitchNotifyCb
 *
 * @brief       Power driver callback to toggle RF switch on Power state
 *              transitions.
 *
 * input parameters
 *
 * @param    eventType - The state change.
 * @param    eventArg  - Not used.
 * @param    clientArg - Not used.
 *
 * @return  Power_NOTIFYDONE to indicate success.
 */
static uint8_t rFSwitchNotifyCb(uint8_t eventType, uint32_t *eventArg,
                                uint32_t *clientArg)
{
  if (eventType == PowerCC26XX_ENTERING_STANDBY)
  {
    // Power down RF Switch
    PIN_setOutputValue(radCtrlHandle, Board_DIO30_SWPWR, 0);
  }
  else if (eventType == PowerCC26XX_AWAKE_STANDBY)
  {
    // Power up RF Switch
    PIN_setOutputValue(radCtrlHandle, Board_DIO30_SWPWR, 1);
  }

  // Notification handled successfully
  return Power_NOTIFYDONE;
}
#endif //CC1350_LAUNCHXL || POWER_SAVING


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

8.2. Application.c

```
/*
 * taskFunction.c
 *
 *  Created on: 26/02/2021
 *      Author: Sergio Ortega
 */

#include <xdc/std.h>
#include <xdc/runtime/System.h>
```

```c
/* BIOS module Headers */
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Semaphore.h>
#include <ti/sysbios/knl/Event.h>

#include <ti/sysbios/knl/Clock.h>

#include <ti/drivers/Board.h>
#include <ti/drivers/GPIO.h>
#include <ti/drivers/ADC.h>
#include "Board.h"

#include <xdc/std.h>
#include <ti/sysbios/BIOS.h>
#include <ti/sysbios/knl/Task.h>

#include <math.h>

#include "AppData.h"

extern Clock_Struct clk1Struct;
extern Clock_Handle clk2Handle;

extern Event_Handle event;
extern Event_Params eventParams;
extern Event_Struct structEvent;

#define START_ACQUISITION         Event_Id_00
#define START_DFT                 Event_Id_01

#define PI 3.14159265

uint16_t adc1Value;

float myData[50];
float myFTabs[50];
float myFTpower[25];

void clk1Fxn(UArg arg0)
{
    /* Process this event */
    //GPIO_toggle(Board_GPIO_LED1);
    Event_post(event, START_ACQUISITION);
}

///* Task function */
void taskFunction(UArg arg0, UArg arg1)
{
    uint32_t count = 0;
    static int sample = 50;                         //amount of data points
taken
    static float Fs = 2000;                             //sampling
frequency in Hz

    ADC_Handle   adc1;
    ADC_Params   params;
```

```
    ADC_init();
    ADC_Params_init(&params);
    adc1 = ADC_open(Board_ADC0, &params);



    int_fast16_t res;
    uint32_t events;

    while(1)
        {
            events = Event_pend(event,
                                Event_Id_NONE,
                                START_ACQUISITION |
                                START_DFT,
                                BIOS_WAIT_FOREVER);

            clk2Handle = Clock_handle(&clk1Struct);
            Clock_start(clk2Handle);

            if (events & START_DFT){

                for (int k = 0; k < sample; k++){
                    float re = 0;
                    float im = 0;
                    for (int n = 0; n < sample; n++){
                        re += myData[n]*cos(2*PI*k*n/sample);
                        im -= myData[n]*sin(2*PI*k*n/sample);
                    }
                    re = re/sample;
                    im = im/sample;
                    myFTabs[k] = sqrt(re*re+im*im);
                }
                for(int i = 0; i < sample/2+1; i++){
                    myFTpower[i] = myFTabs[i];
                }
                for(int i = 0; i < sample/2; i++){
                    myFTpower[i+1] = 2*myFTpower[i+1];
                }

                AppData_SetParameter(APPDATA_APPDATAFS_ID,
APPDATA_APPDATAFS_LEN, &Fs);
                AppData_SetParameter(APPDATA_APPDATAFT_ID,
APPDATA_APPDATAFT_LEN, myFTpower);
                //AppData_SetParameter(APPDATA_APPDATAFT_ID,
APPDATA_APPDATAFT_LEN, test);
                count = 0;
                GPIO_toggle(Board_GPIO_LED0);
            }

            if (events & START_ACQUISITION)
            {
                if (count < sample){
                /* Process this event */
                //GPIO_toggle(Board_GPIO_LED0);
```

```
                res = ADC_convert(adc1, &adc1Value);
                if (res == ADC_STATUS_SUCCESS)
                {
                    //myData[count] = adc1Value;
                    myData[count] = (double) 4.3151*adc1Value/4096;   //
4.3151=Vd*4096/raw
                    //myData[count] =
2*cos(2*PI*count*3/sample+0)+0.5*cos(2*PI*count*5/sample+0);
                    //MyData_SetParameter(MYDATA_DATA_ID,
MYDATA_DATA_LEN, &myData);
                }

                count++;

                }else if (count == sample){
                    //count = 0;
                  Event_post(event, START_DFT);
                }
            }
        }
    }
```

8.3. AppData.c

```
/*
 * AppData.c
 *
 *  Created on: 9/05/2021
 *      Author: Sergio Ortega
 */

/*********************************************************************
**********************
 * Filename:       AppData.c
 *
 * Description:    This file contains the implementation of the service.
 *
 * Copyright (c) 2015-2020, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the
distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
```

```
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *

**************************************************************************
************************/


/*********************************************************************
 * INCLUDES
 */
#include <string.h>

#include <icall.h>

/* This Header file contains all BLE API and icall structure definition
*/
#include "icall_ble_api.h"

#include "AppData.h"

/*********************************************************************
 * MACROS
 */

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

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

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

// AppData Service UUID
CONST uint8_t AppDataUUID[ATT_UUID_SIZE] =
{
  TI_BASE_UUID_128(APPDATA_SERV_UUID)
};

// AppDataFs UUID
```

```
CONST uint8_t AppData_AppDataFsUUID[ATT_UUID_SIZE] =
{
  TI_BASE_UUID_128(APPDATA_APPDATAFS_UUID)
};
// AppDataFT UUID
CONST uint8_t AppData_AppDataFTUUID[ATT_UUID_SIZE] =
{
  TI_BASE_UUID_128(APPDATA_APPDATAFT_UUID)
};

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

static AppDataCBs_t *pAppCBs = NULL;

/*********************************************************************
* Profile Attributes - variables
*/

// Service declaration
static CONST gattAttrType_t AppDataDecl = { ATT_UUID_SIZE, AppDataUUID };

// Characteristic "AppDataFs" Properties (for declaration)
static uint8_t AppData_AppDataFsProps = GATT_PROP_READ;

// Characteristic "AppDataFs" Value variable
static uint8_t AppData_AppDataFsVal[APPDATA_APPDATAFS_LEN] = {0};
// Characteristic "AppDataFT" Properties (for declaration)
static uint8_t AppData_AppDataFTProps = GATT_PROP_READ;

// Characteristic "AppDataFT" Value variable
static uint8_t AppData_AppDataFTVal[APPDATA_APPDATAFT_LEN] = {0};

/*********************************************************************
* Profile Attributes - Table
*/

static gattAttribute_t AppDataAttrTbl[] =
{
  // AppData Service Declaration
  {
    { ATT_BT_UUID_SIZE, primaryServiceUUID },
    GATT_PERMIT_READ,
    0,
    (uint8_t *)&AppDataDecl
  },
    // AppDataFs Characteristic Declaration
    {
      { ATT_BT_UUID_SIZE, characterUUID },
      GATT_PERMIT_READ,
      0,
      &AppData_AppDataFsProps
    },
      // AppDataFs Characteristic Value
      {
        { ATT_UUID_SIZE, AppData_AppDataFsUUID },
```

```
        GATT_PERMIT_READ,
        0,
        AppData_AppDataFsVal
      },
    // AppDataFT Characteristic Declaration
    {
      { ATT_BT_UUID_SIZE, characterUUID },
      GATT_PERMIT_READ,
      0,
      &AppData_AppDataFTProps
    },
      // AppDataFT Characteristic Value
      {
        { ATT_UUID_SIZE, AppData_AppDataFTUUID },
        GATT_PERMIT_READ,
        0,
        AppData_AppDataFTVal
      },
};

/*********************************************************************
 * LOCAL FUNCTIONS
 */
static bStatus_t AppData_ReadAttrCB( uint16_t connHandle, gattAttribute_t
*pAttr,
                                           uint8_t *pValue, uint16_t
*pLen, uint16_t offset,
                                           uint16_t maxLen, uint8_t
method );
static bStatus_t AppData_WriteAttrCB( uint16_t connHandle,
gattAttribute_t *pAttr,
                                            uint8_t *pValue, uint16_t
len, uint16_t offset,
                                            uint8_t method );

/*********************************************************************
 * PROFILE CALLBACKS
 */
// Simple Profile Service Callbacks
CONST gattServiceCBs_t AppDataCBs =
{
  AppData_ReadAttrCB,  // Read callback function pointer
  AppData_WriteAttrCB, // Write callback function pointer
  NULL                       // Authorization callback function pointer
};

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

/*
 * AppData_AddService- Initializes the AppData service by registering
 *          GATT attributes with the GATT server.
 *
 */
extern bStatus_t AppData_AddService( uint8_t rspTaskId )
{
```

```
  uint8_t status;

  // Register GATT attribute list and CBs with GATT Server App
  status = GATTServApp_RegisterService( AppDataAttrTbl,
                                        GATT_NUM_ATTRS( AppDataAttrTbl ),
                                        GATT_MAX_ENCRYPT_KEY_SIZE,
                                        &AppDataCBs );

  return ( status );
}

/*
 * AppData_RegisterAppCBs - Registers the application callback function.
 *                    Only call this function once.
 *
 *    appCallbacks - pointer to application callbacks.
 */
bStatus_t AppData_RegisterAppCBs( AppDataCBs_t *appCallbacks )
{
  if ( appCallbacks )
  {
    pAppCBs = appCallbacks;

    return ( SUCCESS );
  }
  else
  {
    return ( bleAlreadyInRequestedMode );
  }
}

/*
 * AppData_SetParameter - Set a AppData parameter.
 *
 *    param - Profile parameter ID
 *    len - length of data to right
 *    value - pointer to data to write.  This is dependent on
 *            the parameter ID and WILL be cast to the appropriate
 *            data type (example: data type of uint16 will be cast to
 *            uint16 pointer).
 */
bStatus_t AppData_SetParameter( uint8_t param, uint16_t len, void *value
)
{
  bStatus_t ret = SUCCESS;
  switch ( param )
  {
    case APPDATA_APPDATAFS_ID:
      if ( len == APPDATA_APPDATAFS_LEN )
      {
        memcpy(AppData_AppDataFsVal, value, len);
      }
      else
      {
        ret = bleInvalidRange;
      }
      break;
```

```
    case APPDATA_APPDATAFT_ID:
      if ( len == APPDATA_APPDATAFT_LEN )
      {
        memcpy(AppData_AppDataFTVal, value, len);
      }
      else
      {
        ret = bleInvalidRange;
      }
      break;

    default:
      ret = INVALIDPARAMETER;
      break;
  }
  return ret;
}


/*
 * AppData_GetParameter - Get a AppData parameter.
 *
 *    param - Profile parameter ID
 *    value - pointer to data to write.  This is dependent on
 *          the parameter ID and WILL be cast to the appropriate
 *          data type (example: data type of uint16 will be cast to
 *          uint16 pointer).
 */
bStatus_t AppData_GetParameter( uint8_t param, uint16_t *len, void *value
)
{
  bStatus_t ret = SUCCESS;
  switch ( param )
  {
    default:
      ret = INVALIDPARAMETER;
      break;
  }
  return ret;
}


/*********************************************************************
 * @fn          AppData_ReadAttrCB
 *
 * @brief       Read an attribute.
 *
 * @param       connHandle - connection message was received on
 * @param       pAttr - pointer to attribute
 * @param       pValue - pointer to data to be read
 * @param       pLen - length of data to be read
 * @param       offset - offset of the first octet to be read
 * @param       maxLen - maximum length of data to be read
 * @param       method - type of read message
 *
 * @return      SUCCESS, blePending or Failure
```

```
 */
static bStatus_t AppData_ReadAttrCB( uint16_t connHandle, gattAttribute_t
*pAttr,
                                       uint8_t *pValue, uint16_t *pLen,
uint16_t offset,
                                       uint16_t maxLen, uint8_t method )
{
  bStatus_t status = SUCCESS;

  // See if request is regarding the AppDataFs Characteristic Value
if ( ! memcmp(pAttr->type.uuid, AppData_AppDataFsUUID, pAttr->type.len) )
  {
    if ( offset > APPDATA_APPDATAFS_LEN )  // Prevent malicious ATT
ReadBlob offsets.
    {
      status = ATT_ERR_INVALID_OFFSET;
    }
    else
    {
      *pLen = MIN(maxLen, APPDATA_APPDATAFS_LEN - offset);  // Transmit
as much as possible
      memcpy(pValue, pAttr->pValue + offset, *pLen);
    }
  }
  // See if request is regarding the AppDataFT Characteristic Value
else if ( ! memcmp(pAttr->type.uuid, AppData_AppDataFTUUID, pAttr-
>type.len) )
  {
    if ( offset > APPDATA_APPDATAFT_LEN )  // Prevent malicious ATT
ReadBlob offsets.
    {
      status = ATT_ERR_INVALID_OFFSET;
    }
    else
    {
      *pLen = MIN(maxLen, APPDATA_APPDATAFT_LEN - offset);  // Transmit
as much as possible
      memcpy(pValue, pAttr->pValue + offset, *pLen);
    }
  }
  else
  {
    // If we get here, that means you've forgotten to add an if clause
for a
    // characteristic value attribute in the attribute table that has
READ permissions.
    *pLen = 0;
    status = ATT_ERR_ATTR_NOT_FOUND;
  }

  return status;
}


/*********************************************************************
 * @fn      AppData_WriteAttrCB
 *
```

```
 * @brief   Validate attribute data prior to a write operation
 *
 * @param   connHandle - connection message was received on
 * @param   pAttr - pointer to attribute
 * @param   pValue - pointer to data to be written
 * @param   len - length of data
 * @param   offset - offset of the first octet to be written
 * @param   method - type of write message
 *
 * @return  SUCCESS, blePending or Failure
 */
static bStatus_t AppData_WriteAttrCB( uint16_t connHandle,
gattAttribute_t *pAttr,
                                        uint8_t *pValue, uint16_t len,
uint16_t offset,
                                        uint8_t method )
{
  bStatus_t status  = SUCCESS;
  uint8_t   paramID = 0xFF;

  // See if request is regarding a Client Characterisic Configuration
  if ( ! memcmp(pAttr->type.uuid, clientCharCfgUUID, pAttr->type.len) )
  {
    // Allow only notifications.
    status = GATTServApp_ProcessCCCWriteReq( connHandle, pAttr, pValue,
len,
                                             offset,
GATT_CLIENT_CFG_NOTIFY);
  }
  else
  {
    // If we get here, that means you've forgotten to add an if clause
for a
    // characteristic value attribute in the attribute table that has
WRITE permissions.
    status = ATT_ERR_ATTR_NOT_FOUND;
  }

  // Let the application know something changed (if it did) by using the
  // callback it registered earlier (if it did).
  if (paramID != 0xFF)
    if ( pAppCBs && pAppCBs->pfnChangeCb )
    {
      uint16_t svcUuid = APPDATA_SERV_UUID;
      pAppCBs->pfnChangeCb(connHandle, svcUuid, paramID, len, pValue); //
Call app function from stack task context.
    }
  return status;
     }
```

8.4. BATMAN_FT.c

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

 @file  simple_peripheral.c
```

```
 @brief This file contains the Simple Peripheral sample application for
use
        with the CC2650 Bluetooth Low Energy Protocol Stack.

 Group: WCS, BTS
 Target Device: cc2640r2


****************************************************************************
*****

 Copyright (c) 2013-2020, Texas Instruments Incorporated
 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions
 are met:

 *  Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 *  Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 *  Neither the name of Texas Instruments Incorporated nor the names of
    its contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

 THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
IS"
 AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS;
 OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


****************************************************************************
*****



****************************************************************************
****/

/**************************************************************************
 * INCLUDES
 */
#include <string.h>
#include <ti/sysbios/knl/Task.h>
```

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

#if defined( USE_FPGA ) || defined( DEBUG_SW_TRACE )
#include <driverlib/ioc.h>
#endif // USE_FPGA | DEBUG_SW_TRACE

#include <icall.h>
#include "util.h"
#include "att_rsp.h"

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

#include "peripheral.h"

#ifdef USE_RCOSC
#include "rcosc_calibration.h"
#endif //USE_RCOSC

#include "board_key.h"
#include "board.h"
#include <BATMAN_FT.h>

#include "AppData.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

// How often to perform periodic event (in msec)
#define SBP_PERIODIC_EVT_PERIOD               5000

// Application specific event ID for HCI Connection Event End Events
#define SBP_HCI_CONN_EVT_END_EVT              0x0001

// Type of Display to open
#if !defined(Display_DISABLE_ALL)
  #if defined(BOARD_DISPLAY_USE_LCD) && (BOARD_DISPLAY_USE_LCD!=0)
    #define SBP_DISPLAY_TYPE Display_Type_LCD
  #elif defined (BOARD_DISPLAY_USE_UART) && (BOARD_DISPLAY_USE_UART!=0)
    #define SBP_DISPLAY_TYPE Display_Type_UART
  #else // !BOARD_DISPLAY_USE_LCD && !BOARD_DISPLAY_USE_UART
    #define SBP_DISPLAY_TYPE 0 // Option not supported
  #endif // BOARD_DISPLAY_USE_LCD && BOARD_DISPLAY_USE_UART
#else // BOARD_DISPLAY_USE_LCD && BOARD_DISPLAY_USE_UART
  #define SBP_DISPLAY_TYPE 0 // No Display
#endif // !Display_DISABLE_ALL

// Task configuration
#define SBP_TASK_PRIORITY                     1

#ifndef SBP_TASK_STACK_SIZE
#define SBP_TASK_STACK_SIZE                   644
#endif

// Application events
#define SBP_STATE_CHANGE_EVT                  0x0001
#define SBP_CHAR_CHANGE_EVT                   0x0002
#define SBP_PAIRING_STATE_EVT                 0x0004
#define SBP_PASSCODE_NEEDED_EVT               0x0008
#define SBP_CONN_EVT                          0x0010

// Internal Events for RTOS application
#define SBP_ICALL_EVT                         ICALL_MSG_EVENT_ID //
Event_Id_31
#define SBP_QUEUE_EVT                         UTIL_QUEUE_EVENT_ID //
Event_Id_30
#define SBP_PERIODIC_EVT                      Event_Id_00

// Bitwise OR of all events to pend on
#define SBP_ALL_EVENTS                        (SBP_ICALL_EVT        | \
                                               SBP_QUEUE_EVT        | \
                                               SBP_PERIODIC_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 )

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

// App event passed from profiles.
typedef struct
{
  appEvtHdr_t hdr;  // event header.
  uint8_t *pData;  // event data
} sbpEvt_t;

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

// Display Interface
Display_Handle dispHandle = NULL;

extern double myFTpower;
/*********************************************************************
 * LOCAL VARIABLES
 */

// 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;

// Clock instances for internal periodic events.
static Clock_Struct periodicClock;

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

// Task configuration
Task_Struct sbpTask;
Char sbpTaskStack[SBP_TASK_STACK_SIZE];

// Scan response data (max size = 31 bytes)
```

```
static uint8_t scanRspData[] =
{
  // complete name
  10,   // length of this data
  GAP_ADTYPE_LOCAL_NAME_COMPLETE,
  'B',
  'A',
  'T',
  'M',
  'A',
  'N',
  '.',
  'F',
  'T',


  // 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
  0x03,   // length of this data
  GAP_ADTYPE_16BIT_MORE,      // some of the UUID's, but not all
  LO_UINT16(SIMPLEPROFILE_SERV_UUID),
  HI_UINT16(SIMPLEPROFILE_SERV_UUID)
};

// GAP GATT Attributes
static uint8_t attDeviceName[GAP_DEVICE_NAME_LEN] = "BATMAN.FT";


/*********************************************************************
 * LOCAL FUNCTIONS
 */
```

```
static void SimplePeripheral_init( void );
static void SimplePeripheral_taskFxn(UArg a0, UArg a1);

static uint8_t SimplePeripheral_processStackMsg(ICall_Hdr *pMsg);
static uint8_t SimplePeripheral_processGATTMsg(gattMsgEvent_t *pMsg);
static void SimplePeripheral_processAppMsg(sbpEvt_t *pMsg);
static void SimplePeripheral_processStateChangeEvt(gaprole_States_t
newState);
static void SimplePeripheral_processCharValueChangeEvt(uint8_t paramID);
static void SimplePeripheral_performPeriodicTask(void);
static void SimplePeripheral_clockHandler(UArg arg);

static void SimplePeripheral_passcodeCB(uint8_t *deviceAddr,
                                        uint16_t connHandle,
                                        uint8_t uiInputs, uint8_t
uiOutputs,
                                        uint32_t numComparison);
static void SimplePeripheral_pairStateCB(uint16_t connHandle, uint8_t
state,
                                         uint8_t status);
static void SimplePeripheral_processPairState(uint8_t state, uint8_t
status);
static void SimplePeripheral_processPasscode(uint8_t uiOutputs);

static void SimplePeripheral_stateChangeCB(gaprole_States_t newState);
static void SimplePeripheral_charValueChangeCB(uint8_t paramID);
static uint8_t SimplePeripheral_enqueueMsg(uint8_t event, uint8_t state,
                                              uint8_t *pData);

static void SimplePeripheral_connEvtCB(Gap_ConnEventRpt_t *pReport);
static void SimplePeripheral_processConnEvt(Gap_ConnEventRpt_t *pReport);


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

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

// Peripheral GAPRole Callbacks
static gapRolesCBs_t SimplePeripheral_gapRoleCBs =
{
  SimplePeripheral_stateChangeCB     // GAPRole State Change Callbacks
};

// GAP Bond Manager Callbacks
// These are set to NULL since they are not needed. The application
// is set up to only perform justworks pairing.
static gapBondCBs_t simplePeripheral_BondMgrCBs =
{
  SimplePeripheral_passcodeCB,  // Passcode callback
  SimplePeripheral_pairStateCB  // Pairing / Bonding state Callback
};
```

```c
// Simple GATT Profile Callbacks
static simpleProfileCBs_t SimplePeripheral_simpleProfileCBs =
{
  SimplePeripheral_charValueChangeCB // Simple GATT Characteristic value
change callback
};

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

/*********************************************************************
 * The following typedef and global handle the registration to connection
event
 */
typedef enum
{
   NOT_REGISTER       = 0,
   FOR_AOA_SCAN       = 1,
   FOR_ATT_RSP        = 2,
   FOR_AOA_SEND       = 4,
   FOR_TOF_SEND       = 8
}connectionEventRegisterCause_u;

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

/*********************************************************************
 * @fn      SimplePeripheral_RegistertToAllConnectionEvent()
 *
 * @brief   register to receive connection events for all the connection
 *
 * @param connectionEventRegisterCause represents the reason for
registration
 *
 * @return @ref SUCCESS
 *
 */
bStatus_t SimplePeripheral_RegistertToAllConnectionEvent
(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(SimplePeripheral_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      SimplePeripheral_UnRegistertToAllConnectionEvent()
 *
 * @brief   Unregister connection events
 *
 * @param connectionEventRegisterCause represents the reason for
registration
 *
 * @return @ref SUCCESS
 *
 */
bStatus_t SimplePeripheral_UnRegistertToAllConnectionEvent
(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(SimplePeripheral_connEvtCB,
GAP_CB_UNREGISTER, LINKDB_CONNHANDLE_ALL);
  }

  return(status);
}

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

  // Configure task
  Task_Params_init(&taskParams);
  taskParams.stack = sbpTaskStack;
  taskParams.stackSize = SBP_TASK_STACK_SIZE;
  taskParams.priority = SBP_TASK_PRIORITY;

  Task_construct(&sbpTask, SimplePeripheral_taskFxn, &taskParams, NULL);
}

/*********************************************************************
 * @fn      SimplePeripheral_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 SimplePeripheral_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);

#ifdef USE_RCOSC
  RCOSC_enableCalibration();
#endif // USE_RCOSC

#if defined( USE_FPGA )
  // configure RF Core SMI Data Link
  IOCPortConfigureSet(IOID_12, IOC_PORT_RFC_GPO0, IOC_STD_OUTPUT);
  IOCPortConfigureSet(IOID_11, IOC_PORT_RFC_GPI0, IOC_STD_INPUT);

  // configure RF Core SMI Command Link
  IOCPortConfigureSet(IOID_10, IOC_IOCFG0_PORT_ID_RFC_SMI_CL_OUT,
IOC_STD_OUTPUT);
  IOCPortConfigureSet(IOID_9, IOC_IOCFG0_PORT_ID_RFC_SMI_CL_IN,
IOC_STD_INPUT);

  // configure RF Core tracer IO
  IOCPortConfigureSet(IOID_8, IOC_PORT_RFC_TRC, IOC_STD_OUTPUT);
#else // !USE_FPGA
  #if defined( DEBUG_SW_TRACE )
    // configure RF Core tracer IO
    IOCPortConfigureSet(IOID_8, IOC_PORT_RFC_TRC, IOC_STD_OUTPUT |
IOC_CURRENT_4MA | IOC_SLEW_ENABLE);
  #endif // DEBUG_SW_TRACE
#endif // USE_FPGA

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

  // Create one-shot clocks for internal periodic events.
  Util_constructClock(&periodicClock, SimplePeripheral_clockHandler,
                      SBP_PERIODIC_EVT_PERIOD, 0, false,
SBP_PERIODIC_EVT);

  dispHandle = Display_open(SBP_DISPLAY_TYPE, NULL);

  // 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/
  {
    // 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;

    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);
  }

  // Setup the GAP Bond Manager. For more information see the section in
the
  // User's Guide:
  // http://software-dl.ti.com/lprf/sdg-latest/html/
  {
    // Don't send a pairing request after connecting; the peer device
must
    // initiate pairing
    uint8_t pairMode = GAPBOND_PAIRING_MODE_WAIT_FOR_REQ;
    // Use authenticated pairing: require passcode.
    uint8_t mitm = TRUE;
    // This device only has display capabilities. Therefore, it will
display the
    // passcode during pairing. However, since the default passcode is
being
    // used, there is no need to display anything.
    uint8_t ioCap = GAPBOND_IO_CAP_DISPLAY_ONLY;
    // Request bonding (storing long-term keys for re-encryption upon
subsequent
    // connections without repairing)
    uint8_t bonding = TRUE;
    // Whether to replace the least recently used entry when bond list is
full,
    // and a new device is bonded.
    // Alternative is pairing succeeds but bonding fails, unless
application has
    // manually erased at least one bond.
    uint8_t replaceBonds = FALSE;

    GAPBondMgr_SetParameter(GAPBOND_PAIRING_MODE, sizeof(uint8_t),
&pairMode);
    GAPBondMgr_SetParameter(GAPBOND_MITM_PROTECTION, sizeof(uint8_t),
&mitm);
    GAPBondMgr_SetParameter(GAPBOND_IO_CAPABILITIES, sizeof(uint8_t),
&ioCap);
    GAPBondMgr_SetParameter(GAPBOND_BONDING_ENABLED, sizeof(uint8_t),
&bonding);
    GAPBondMgr_SetParameter(GAPBOND_LRU_BOND_REPLACEMENT,
sizeof(uint8_t), &replaceBonds);
  }

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

```
  AppData_AddService( selfEntity );


  // Setup the SimpleProfile Characteristic Values
  // For more information, see the sections in the User's Guide:
  // http://software-dl.ti.com/lprf/sdg-latest/html/
  {
    uint8_t charValue1 = 1;
    uint8_t charValue2 = 2;
    uint8_t charValue3 = 3;
    uint8_t charValue4 = 4;
    uint8_t charValue5[SIMPLEPROFILE_CHAR5_LEN] = { 1, 2, 3, 4, 5 };

    SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR1, sizeof(uint8_t),
                               &charValue1);
    SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR2, sizeof(uint8_t),
                               &charValue2);
    SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR3, sizeof(uint8_t),
                               &charValue3);
    SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR4, sizeof(uint8_t),
                               &charValue4);
    SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR5,
SIMPLEPROFILE_CHAR5_LEN,
                               charValue5);
    /* Add your new characteristic to the service. These names may vary
*/
    // Initalization of characteristics in AppData that are readable.
    uint8_t AppData_AppDataFs_initVal[APPDATA_APPDATAFS_LEN] = {0};
    AppData_SetParameter(APPDATA_APPDATAFS_ID, APPDATA_APPDATAFS_LEN,
AppData_AppDataFs_initVal);
    uint8_t AppData_AppDataFT_initVal[APPDATA_APPDATAFT_LEN] = {0};
    AppData_SetParameter(APPDATA_APPDATAFT_ID, APPDATA_APPDATAFT_LEN,
AppData_AppDataFT_initVal);

  }

  // Start the Device:
  // Please Notice that in case of wanting to use the
GAPRole_SetParameter
  // function with GAPROLE_IRK or GAPROLE_SRK parameter - Perform
  // these function calls before the GAPRole_StartDevice use.
  // (because Both cases are updating the gapRole_IRK & gapRole_SRK
variables).
  VOID GAPRole_StartDevice(&SimplePeripheral_gapRoleCBs);

  // Register callback with SimpleGATTprofile
  SimpleProfile_RegisterAppCBs(&SimplePeripheral_simpleProfileCBs);

  // Start Bond Manager and register callback
  VOID GAPBondMgr_Register(&simplePeripheral_BondMgrCBs);

  // 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);

  //Set default values for Data Length Extension
  {
    //Set initial values to maximum, RX is set to max. by default(251
octets, 2120us)
    #define APP_SUGGESTED_PDU_SIZE 251 //default is 27 octets(TX)
    #define APP_SUGGESTED_TX_TIME 2120 //default is 328us(TX)

    //This API is documented in hci.h
    //See the LE Data Length Extension section in the BLE-Stack User's
Guide for information on using this command:
    //http://software-dl.ti.com/lprf/sdg-latest/html/cc2640/index.html
    //HCI_LE_WriteSuggestedDefaultDataLenCmd(APP_SUGGESTED_PDU_SIZE,
APP_SUGGESTED_TX_TIME);
  }

#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)

  Display_print0(dispHandle, 0, 0, "BLE Peripheral");
}

/*********************************************************************
 * @fn      SimplePeripheral_taskFxn
 *
 * @brief   Application task entry point for the Simple Peripheral.
 *
 * @param   a0, a1 - not used.
 *
 * @return  None.
 */
static void SimplePeripheral_taskFxn(UArg a0, UArg a1)
{
  // Initialize application
  SimplePeripheral_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, SBP_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 = SimplePeripheral_processStackMsg((ICall_Hdr
*)pMsg);
          }
        }

        if (pMsg && safeToDealloc)
        {
          ICall_freeMsg(pMsg);
        }
      }

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

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

      if (events & SBP_PERIODIC_EVT)
      {
        Util_startClock(&periodicClock);

        // Perform periodic application task
        SimplePeripheral_performPeriodicTask();
      }
    }
  }
```

```
}

/*********************************************************************
 * @fn      SimplePeripheral_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 SimplePeripheral_processStackMsg(ICall_Hdr *pMsg)
{
  uint8_t safeToDealloc = TRUE;

  switch (pMsg->event)
  {
    case GATT_MSG_EVENT:
      // Process GATT message
      safeToDealloc = SimplePeripheral_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      SimplePeripheral_processGATTMsg
 *
 * @brief   Process GATT messages and events.
 *
 * @return  TRUE if safe to deallocate incoming message, FALSE otherwise.
 */
```

```
static uint8_t SimplePeripheral_processGATTMsg(gattMsgEvent_t *pMsg)
{
  // See if GATT server was unable to transmit an ATT response
  if (attRsp_isAttRsp(pMsg))
  {
    // No HCI buffer was available. Let's try to retransmit the response
    // on the next connection event.
    if( SimplePeripheral_RegistertToAllConnectionEvent(FOR_ATT_RSP) ==
SUCCESS)
    {
      // Don't free the response message yet
      return (FALSE);
    }
  }
  else if (pMsg->method == ATT_FLOW_CTRL_VIOLATED_EVENT)
  {
    // ATT request-response or indication-confirmation flow control is
    // violated. All subsequent ATT requests or indications will be
dropped.
    // The app is informed in case it wants to drop the connection.

    // Display the opcode of the message that caused the violation.
    Display_print1(dispHandle, 5, 0, "FC Violated: %d", pMsg-
>msg.flowCtrlEvt.opcode);
  }
  else if (pMsg->method == ATT_MTU_UPDATED_EVENT)
  {
    // MTU size updated
    Display_print1(dispHandle, 5, 0, "MTU Size: %d", pMsg-
>msg.mtuEvt.MTU);
  }

  // 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      SimplePeripheral_processConnEvt
 *
 * @brief   Process connection event.
 *
 * @param pReport pointer to connection event report
 */
static void SimplePeripheral_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)
    if (attRsp_sendAttRsp() == SUCCESS)
```

```
    {
        // Disable connection event end notice
        SimplePeripheral_UnRegistertToAllConnectionEvent (FOR_ATT_RSP);
    }
  }

}

/*********************************************************************
 * @fn      SimplePeripheral_processAppMsg
 *
 * @brief   Process an incoming callback from a profile.
 *
 * @param   pMsg - message to process
 *
 * @return  None.
 */
static void SimplePeripheral_processAppMsg(sbpEvt_t *pMsg)
{
  switch (pMsg->hdr.event)
  {
    case SBP_STATE_CHANGE_EVT:
      {
        SimplePeripheral_processStateChangeEvt((gaprole_States_t)pMsg->
                                                hdr.state);
      }
      break;

    case SBP_CHAR_CHANGE_EVT:
      {
        SimplePeripheral_processCharValueChangeEvt(pMsg->hdr.state);
      }
      break;

    // Pairing event
    case SBP_PAIRING_STATE_EVT:
      {
        SimplePeripheral_processPairState(pMsg->hdr.state, *pMsg->pData);

        ICall_free(pMsg->pData);
        break;
      }

    // Passcode event
    case SBP_PASSCODE_NEEDED_EVT:
      {
        SimplePeripheral_processPasscode(*pMsg->pData);

        ICall_free(pMsg->pData);
        break;
      }

      case SBP_CONN_EVT:
      {
        SimplePeripheral_processConnEvt((Gap_ConnEventRpt_t *)(pMsg-
>pData));
```

```
        ICall_free(pMsg->pData);
        break;
        }

    default:
      // Do nothing.
      break;
  }
}

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

/*********************************************************************
 * @fn      SimplePeripheral_processStateChangeEvt
 *
 * @brief   Process a pending GAP Role state change event.
 *
 * @param   newState - new state
 *
 * @return  None.
 */
static void SimplePeripheral_processStateChangeEvt(gaprole_States_t
newState)
{
#ifdef PLUS_BROADCASTER
  static bool firstConnFlag = false;
#endif // PLUS_BROADCASTER

  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);

        // Display device address
        Display_print0(dispHandle, 1, 0,
Util_convertBdAddr2Str(ownAddress));
        Display_print0(dispHandle, 2, 0, "Initialized");

        // Device starts advertising upon initialization of GAP
        uint8_t initialAdvertEnable = TRUE;
        // Set the Peripheral GAPRole Parameters
        GAPRole_SetParameter(GAPROLE_ADVERT_ENABLED, sizeof(uint8_t),
                             &initialAdvertEnable);
      }
      break;

    case GAPROLE_ADVERTISING:
      Display_print0(dispHandle, 2, 0, "Advertising");
      break;

#ifdef PLUS_BROADCASTER
    // After a connection is dropped, a device in PLUS_BROADCASTER will
continue
    // sending non-connectable advertisements and shall send this change
of
    // state to the application.  These are then disabled here so that
sending
    // connectable advertisements can resume.
    case GAPROLE_ADVERTISING_NONCONN:
      {
        uint8_t advertEnabled = FALSE;

        // Disable non-connectable advertising.
        GAPRole_SetParameter(GAPROLE_ADV_NONCONN_ENABLED,
sizeof(uint8_t),
                              &advertEnabled);

        advertEnabled = TRUE;

        // Enabled connectable advertising.
        GAPRole_SetParameter(GAPROLE_ADVERT_ENABLED, sizeof(uint8_t),
                              &advertEnabled);

        // Reset flag for next connection.
        firstConnFlag = false;

        attRsp_freeAttRsp(bleNotConnected);
      }
      break;
#endif //PLUS_BROADCASTER
```

```
    case GAPROLE_CONNECTED:
      {
        linkDBInfo_t linkInfo;
        uint8_t numActive = 0;

        Util_startClock(&periodicClock);

        numActive = linkDB_NumActive();

        // Use numActive to determine the connection handle of the last
        // connection
        if ( linkDB_GetInfo( numActive - 1, &linkInfo ) == SUCCESS )
        {
          Display_print1(dispHandle, 2, 0, "Num Conns: %d",
(uint16_t)numActive);
          Display_print0(dispHandle, 3, 0,
Util_convertBdAddr2Str(linkInfo.addr));
        }
        else
        {
          uint8_t peerAddress[B_ADDR_LEN];

          GAPRole_GetParameter(GAPROLE_CONN_BD_ADDR, peerAddress);

          Display_print0(dispHandle, 2, 0, "Connected");
          Display_print0(dispHandle, 3, 0,
Util_convertBdAddr2Str(peerAddress));
        }

        #ifdef PLUS_BROADCASTER
          // Only turn advertising on for this state when we first
connect
          // otherwise, when we go from connected_advertising back to
this state
          // we will be turning advertising back on.
          if (firstConnFlag == false)
          {
            uint8_t advertEnabled = FALSE; // Turn on Advertising

            // Disable connectable advertising.
            GAPRole_SetParameter(GAPROLE_ADVERT_ENABLED, sizeof(uint8_t),
                                 &advertEnabled);

            // Set to true for non-connectable advertising.
            advertEnabled = TRUE;

            // Enable non-connectable advertising.
            GAPRole_SetParameter(GAPROLE_ADV_NONCONN_ENABLED,
sizeof(uint8_t),
                                 &advertEnabled);
            firstConnFlag = true;
          }
        #endif // PLUS_BROADCASTER
      }
      break;

    case GAPROLE_CONNECTED_ADV:
```

```c
      Display_print0(dispHandle, 2, 0, "Connected Advertising");
      break;

    case GAPROLE_WAITING:
      {
        uint8_t advertReEnable = TRUE;

        Util_stopClock(&periodicClock);
        attRsp_freeAttRsp(bleNotConnected);

        // Clear remaining lines
        Display_clearLines(dispHandle, 3, 5);

        GAPRole_SetParameter(GAPROLE_ADVERT_ENABLED, sizeof(uint8_t),
&advertReEnable);
        Display_print0(dispHandle, 2, 0, "Advertising");
      }
      break;

    case GAPROLE_WAITING_AFTER_TIMEOUT:
      attRsp_freeAttRsp(bleNotConnected);

      Display_print0(dispHandle, 2, 0, "Timed Out");

      // Clear remaining lines
      Display_clearLines(dispHandle, 3, 5);

      #ifdef PLUS_BROADCASTER
        // Reset flag for next connection.
        firstConnFlag = false;
      #endif // PLUS_BROADCASTER
      break;

    case GAPROLE_ERROR:
      Display_print0(dispHandle, 2, 0, "Error");
      break;

    default:
      Display_clearLine(dispHandle, 2);
      break;
  }

}

/*********************************************************************
 * @fn      SimplePeripheral_charValueChangeCB
 *
 * @brief   Callback from Simple Profile indicating a characteristic
 *          value change.
 *
 * @param   paramID - parameter ID of the value that was changed.
 *
 * @return  None.
 */
static void SimplePeripheral_charValueChangeCB(uint8_t paramID)
{
  SimplePeripheral_enqueueMsg(SBP_CHAR_CHANGE_EVT, paramID, 0);
```

```
}

/*********************************************************************
 * @fn      SimplePeripheral_processCharValueChangeEvt
 *
 * @brief   Process a pending Simple Profile characteristic value change
 *          event.
 *
 * @param   paramID - parameter ID of the value that was changed.
 *
 * @return  None.
 */
static void SimplePeripheral_processCharValueChangeEvt(uint8_t paramID)
{
  uint8_t newValue;

  switch(paramID)
  {
    case SIMPLEPROFILE_CHAR1:
      SimpleProfile_GetParameter(SIMPLEPROFILE_CHAR1, &newValue);

      Display_print1(dispHandle, 4, 0, "Char 1: %d", (uint16_t)newValue);
      break;

    case SIMPLEPROFILE_CHAR3:
      SimpleProfile_GetParameter(SIMPLEPROFILE_CHAR3, &newValue);

      Display_print1(dispHandle, 4, 0, "Char 3: %d", (uint16_t)newValue);
      break;

    default:
      // should not reach here!
      break;
  }
}

/*********************************************************************
 * @fn      SimplePeripheral_performPeriodicTask
 *
 * @brief   Perform a periodic application task. This function gets
called
 *          every five seconds (SBP_PERIODIC_EVT_PERIOD). In this
example,
 *          the value of the third characteristic in the
SimpleGATTProfile
 *          service is retrieved from the profile, and then copied into
the
 *          value of the the fourth characteristic.
 *
 * @param   None.
 *
 * @return  None.
 */
static void SimplePeripheral_performPeriodicTask(void)
{
  uint8_t valueToCopy;
```

```
  // Call to retrieve the value of the third characteristic in the
profile
  if (SimpleProfile_GetParameter(SIMPLEPROFILE_CHAR3, &valueToCopy) ==
SUCCESS)
  {
    // Call to set that value of the fourth characteristic in the
profile.
    // Note that if notifications of the fourth characteristic have been
    // enabled by a GATT client device, then a notification will be sent
    // every time this function is called.
    SimpleProfile_SetParameter(SIMPLEPROFILE_CHAR4, sizeof(uint8_t),
                               &valueToCopy);
  }
}

/*********************************************************************
 * @fn      SimplePeripheral_pairStateCB
 *
 * @brief   Pairing state callback.
 *
 * @return  none
 */
static void SimplePeripheral_pairStateCB(uint16_t connHandle, uint8_t
state,
                                            uint8_t status)
{
  uint8_t *pData;

  // Allocate space for the event data.
  if ((pData = ICall_malloc(sizeof(uint8_t))))
  {
    *pData = status;

    // Queue the event.
    SimplePeripheral_enqueueMsg(SBP_PAIRING_STATE_EVT, state, pData);
  }
}

/*********************************************************************
 * @fn      SimplePeripheral_processPairState
 *
 * @brief   Process the new paring state.
 *
 * @return  none
 */
static void SimplePeripheral_processPairState(uint8_t state, uint8_t
status)
{
  if (state == GAPBOND_PAIRING_STATE_STARTED)
  {
    Display_print0(dispHandle, 2, 0, "Pairing started");
  }
  else if (state == GAPBOND_PAIRING_STATE_COMPLETE)
  {
    if (status == SUCCESS)
    {
      Display_print0(dispHandle, 2, 0, "Pairing success");
```

```
    }
    else
    {
      Display_print1(dispHandle, 2, 0, "Pairing fail: %d", status);
    }
  }
  else if (state == GAPBOND_PAIRING_STATE_BONDED)
  {
    if (status == SUCCESS)
    {
      Display_print0(dispHandle, 2, 0, "Bonding success");
    }
  }
  else if (state == GAPBOND_PAIRING_STATE_BOND_SAVED)
  {
    if (status == SUCCESS)
    {
      Display_print0(dispHandle, 2, 0, "Bond save success");
    }
    else
    {
      Display_print1(dispHandle, 2, 0, "Bond save failed: %d", status);
    }
  }
}

/*********************************************************************
 * @fn      SimplePeripheral_passcodeCB
 *
 * @brief   Passcode callback.
 *
 * @return  none
 */
static void SimplePeripheral_passcodeCB(uint8_t *deviceAddr,
                                        uint16_t connHandle,
                                        uint8_t uiInputs,
                                        uint8_t uiOutputs,
                                        uint32_t numComparison)
{
  uint8_t *pData;

  // Allocate space for the passcode event.
  if ((pData = ICall_malloc(sizeof(uint8_t))))
  {
    *pData = uiOutputs;

    // Enqueue the event.
    SimplePeripheral_enqueueMsg(SBP_PASSCODE_NEEDED_EVT, 0, pData);
  }
}

/*********************************************************************
 * @fn      SimplePeripheral_processPasscode
 *
 * @brief   Process the Passcode request.
 *
 * @return  none
```

```
 */
static void SimplePeripheral_processPasscode(uint8_t uiOutputs)
{
  // This app uses a default passcode. A real-life scenario would handle
all
  // pairing scenarios and likely generate this randomly.
  uint32_t passcode = B_APP_DEFAULT_PASSCODE;

  // Display passcode to user
  if (uiOutputs != 0)
  {
    Display_print1(dispHandle, 4, 0, "Passcode: %d", passcode);
  }

  uint16_t connectionHandle;
  GAPRole_GetParameter(GAPROLE_CONNHANDLE, &connectionHandle);

  // Send passcode response
  GAPBondMgr_PasscodeRsp(connectionHandle, SUCCESS, passcode);
}

/*********************************************************************
 * @fn      SimplePeripheral_clockHandler
 *
 * @brief   Handler function for clock timeouts.
 *
 * @param   arg - event type
 *
 * @return  None.
 */
static void SimplePeripheral_clockHandler(UArg arg)
{
  // Wake up the application.
  Event_post(syncEvent, arg);
}

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

}

/*********************************************************************
 *
 * @brief   Creates a message and puts the message in RTOS queue.
```

```c
 *
 * @param   event - message event.
 * @param   state - message state.
 * @param   pData - message data pointer.
 *
 * @return  TRUE or FALSE
 */
static uint8_t SimplePeripheral_enqueueMsg(uint8_t event, uint8_t state,
                                           uint8_t *pData)
{
  sbpEvt_t *pMsg = ICall_malloc(sizeof(sbpEvt_t));

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

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

  return FALSE;
}
/*********************************************************************
*********************************************************************/
```

9. Bibliography

[1] B. T. "Preliminery Analysis of BAT-MAN Project," 2019.

[2] B. Mo, J. Yu, D. Tang and H. Liu, "A remaining useful life prediction approach for lithium-ion batteries using kalman filter and an improved particle filter.," IEEE International Conference, 2016, pp. 1-5.

[3] Q. Miao, L. Xie, H. Cui, W. Liang and M. Pecht, Remaining useful life prediction of lithium-ion battery with unscented particle filter technique, Microelectronics Reliability, 2013.

[4] T. Stockley, K. Thanapalan, M. Bowkett, J. Williams and M. Hathway, Advanced eis techniques for performance evaluation of li-ion cells, 2014.

[5] T. U, K. O and T. H-R, Characterizing aging effectes of litium ion batteries by impedance spectroscopy, Electrochim Acta, 2006.

[6] H. F, "A review of impedance measurements for determination of state-of-charge or state-of-health of secondary batteries," J Power Sources, 1998, pp. 59-69.

[7] "1260A Impedance Analyzer is the Cornerstone of FRA technology.," AMETEK Scientific Instruments, [Online]. Available: https://www.ameteksi.com/products/frequency-response-analyzers/1260a-impedance-gain-phase-analyzer. [Accessed 06 2021].

[8] "EmStat4S, USB powered potentiostat / galvanostat with EIS," PalmSens Compact Electrochemical Interfaces, [Online]. Available: https://www.palmsens.com/product/emstat4s/. [Accessed 06 2021].

[9] "Vector Network Analyzer - Bode 100," OMICRON LAB, [Online]. Available: https://www.omicron-lab.com/products/vector-network-analysis/bode-

100/?gclid=Cj0KCQjwraqHBhDsARIsAKuGZeGq0VLvXBcmhCGsxQWNXrEgvN2zSoG2zHsY3fBMfi451T0b7cWZkTYaAsnbEALw_wcB#. [Accessed 06 2021].

[10] "TI-RTOS (RTOS Kernel) Overview," Texas Instruments, [Online]. Available: http://software-dl.ti.com/simplelink/esd/simplelink_cc2640r2_sdk/3.20.00.21/exports/docs/proprietary-rf/proprietary-rf-users-guide/proprietary-rf-guide/tirtos-index.html. [Accessed 06 2021].

[11] "TI-RTOS: Real-Time Operating System (RTOS) for Microcontrollers (MCU)," Texas Instruments, [Online]. Available: https://www.ti.com/tool/TI-RTOS-MCU. [Accessed 06 2021].