PipelineBLE Documentation

University of Pittsburgh

Samuel Peterson

[Sdp38@pitt.edu](mailto:Sdp38@pitt.edu)

636-489-9930

Table of Contents

[1.0 Overview 2](#_Toc27649045)

[2.0 Requirements 3](#_Toc27649046)

[2.1 The app shall be able to connect to multiple sensors. 3](#_Toc27649047)

[2.1.1 The app shall be able to disconnect from specific sensors. 3](#_Toc27649048)

[2.1.2 The app shall be able to store data about each sensor. 3](#_Toc27649049)

[2.1.3 The app shall be able to detect other non-connected sensors. 3](#_Toc27649050)

[2.2 The app shall be able to communicate with the sensors. 3](#_Toc27649051)

[2.2.1 The app shall be able to request specific information from the sensors. 3](#_Toc27649052)

[2.2.2 The app shall be able to record data received from the sensors. 3](#_Toc27649053)

[2.2.3 The app shall be able to plot data that is being measured. 4](#_Toc27649054)

[2.2.4 The app shall be able to send commands to the sensor. 4](#_Toc27649055)

[2.3 The app shall be able to export data that it has received. 4](#_Toc27649056)

[3.0 Code Review 4](#_Toc27649057)

[3.1 Controllers 4](#_Toc27649058)

[3.1.1 AppDelegate.swift 4](#_Toc27649059)

[3.1.2 MainPageViewControllers 5](#_Toc27649060)

[3.1.3 ConnectedDeviceViewControllers 9](#_Toc27649061)

[3.2 Common 20](#_Toc27649062)

[3.2.1 ExportData.swift 21](#_Toc27649063)

[3.2.2 DataFormatter.swift 21](#_Toc27649064)

[3.2.3 PersistentData 21](#_Toc27649065)

[3.2.4 AdafruitKit 23](#_Toc27649066)

[4.0 Conclusion 24](#_Toc27649067)

[5.0 References 24](#_Toc27649068)

# 1.0 Overview

This documentation will review the PipelineBLE iOS application and the requirements used when designing. The main purpose of this app is to communicate with multiple Bluetooth Low Energy (BLE) devices at one time. This communication was primarily enabled through UART.

# 2.0 Requirements

Below are the requirements that were used when designing the app. These requirements changed throughout as feedback was received.

## 2.1 The app shall be able to connect to multiple sensors.

The app will be able to connect to multiple sensors. These sensors are defined to be UART capable devices. Communication with non-UART capable devices is not guaranteed.

### 2.1.1 The app shall be able to disconnect from specific sensors.

The app shall allow the user to be able to connect/disconnect from specific devices. Connecting and disconnecting from devices should be intuitive and easy.

### 2.1.2 The app shall be able to store data about each sensor.

The app shall be able to store specific data about each sensor so that it is intuitive when delineating sensors from each other.

### 2.1.3 The app shall be able to detect other non-connected sensors.

Although communication is only guaranteed with UART capable devices, the app shall be able to detect other sensors that are not connected and non-UART capable.

## 2.2 The app shall be able to communicate with the sensors.

Communication between the app and the sensors is vital. This communication shall be possible so long as the sensors are UART capable.

### 2.2.1 The app shall be able to request specific information from the sensors.

The type of information requested can be configured by the user and saved for future use.

### 2.2.2 The app shall be able to record data received from the sensors.

The data communicated between the app and the sensor maybe important and it shall be possible to save this data. This data will be available to review at a later point.

### 2.2.3 The app shall be able to plot data that is being measured.

The app will be able to show the user the data that it is current receiving from the sensors. Data from multiple sensors will be displayed on the same graphs.

### 2.2.4 The app shall be able to send commands to the sensor.

Commands will be able to be sent to the sensors. These commands will be able to be saved to be used for later use as well.

## 2.3 The app shall be able to export data that it has received.

The data that is received from the sensors and sent from the app will be able to be exported. This data includes both UART data as well as graph data.

2.3.1 The app shall be able to export data in txt format.

In order to view the data on any machine, the data will be able to be exported in txt format. Both graph data and UART data will be able to be exported in txt format.

# 3.0 Code Review

This section will review each file and the process through each file. For many files, it will be impossible to follow the flow due to the nature of the application. The application must handle asynchronous actions and react accordingly. It is impossible to know how exactly the app will execute, across many threads, in these cases.

There will be a number of files that will not be discussed in great detail. These files were used from the open source Bluefruit\_LE\_Connect\_v2 repository on GitHub, written by Adafruit [1]. A few of the most important files and functions within those files will be reviewed, but most of the other files will not be reviewed.

Other files that will not be reviewed are Podfiles. A Pod is essentially a third-party package that is imported into the application. The list of the Pods that were used can be found in the Podfile. All these Pods are open source and free to use.

## 3.1 Controllers

The files contained in this section are primarily responsible for handling the content that is displayed to the users and handling the interactions with the user. The two groups of controllers are MainPage controllers and ConnectedDevice controllers. MainPage controllers are used for when there is no device connected and the application is just starting up. ConnectedDevice controllers are used when there are device(s) connected and the user needs to interact with those device(s).

### 3.1.1 AppDelegate.swift

This is the file that gets called when the app is first starting up. The only function that’s worth noting in this file is the application method.

**application():**

This method is used to set up a number of configurations at start up. First, it uses the Preferences object to set some parameters about the Bluetooth connection. To view these parameters, look at the ~/Resources/DefaultPreferences.plist. Since the application’s GUI is configured programmatically instead of with the storyboards, the initial view controller that is shown on startup is set here. The view controller that is set as the initial view is the MainPageTabBarController.

### 3.1.2 MainPageViewControllers

The controllers that are present in this section are used to control the three view controllers that are accessible on the programs start up. The three different view controllers can be seen at the bottom of the screen: Available Devices, Saved Devices, and Past Data.

#### 3.1.2.1 MainPageTabBarController.swift

Although this is the initial view that is opened on startup, it really doesn’t do much. All it does is initialize the three views and encapsulates them to make it easy to navigate between them.

**createNavController():**

This method initializes the give view controller with the same parameters. It will create each view controller and set the title and image of each controller in the tab bar.

#### 3.1.2.2 AvailableDevicesControllers

These controllers are responsible for scanning for devices and connecting to those devices. The devices will be presented to the user to either communicate with those device(s) or continue to search for other devices.

##### 3.1.2.2.1 AvailableDevicesViewController.swift

As mentioned above, this file will scan and connected with devices. The methods will be discussed in the order they would most likely be called.

**viewDidLoad():**

The first thing the app tries to do is get all the previously saved peripherals. How this is done will be discussed further in **getSavedPeripherals()**. Next, a **PeripheralList** object will be created to store all of the scanned peripherals. Next, a **refreshController** will be created and added to the **tableView**. This **refreshController** will call the **onTableRefresh()** method whenever there is a change to the **tableView**. Next, the **nextPageButton** is created, which will call the **nextPage()** method whenever it is pressed from the top right corner in the navigation controller.

**viewWillAppear():**

This method will be called anytime the view is going to appear for the user. The first thing it does is call the **didUpdateBleState()** method to see if there were any changes to the peripherals. Next, it will call the **registerNotifications()** methods to add notifications. The app will then disconnect from all currently connected peripherals. Lastly, the app will begin scanning for peripherals and update the **tableView** to reflect the scanned peripherals.

**viewWillDisappear()**:

This method will be called whenever the app is about to be removed from view. When this happens, the BLE module will be told to stop scanning. Next, the notifications that were enabled on startup will be turned off. The peripherals in the **peripheralList** will then be cleared. Finally, **dirtyData** is then set to true so that if the view is reopened, it knows to rescan for savedPeripherals.

**tableView(numberOfRowsInSection):**

This method will tell the app how many rows to have in each section. For the **availableDevices** section, it will count the number of devices that are not connected or are disconnecting.

**tableView(cellForRowAt):**

This method will determine what content should be displayed in the given cell. For the **connectedDevices** section, this will be the peripherals that the app is connected to. For the **availableDevices** section, this will be the next available device that is not connected. Depending on whether or not the device has been saved before or not, the app will display the saved name or the advertised name.

**tableView(didSelectRowAt):**

If a device is selected in the **availableDevices** section, then the app will save information about it if necessary and then connect. If the information is already saved, then it will just connect.

**tableView(commit, forRowAt):**

If a device is deleted from the **connectedDevices** section, then the device will be disconnected from.

**getSavedPeripherals():**

This method will use **NSFetchRequest** and **PersistenceService** to grab all saved peripherals. All these saved peripherals will then be saved in the **savedDevices** array. It will then mark that the data has been updated recently.

**nextPage():**

This function is used once there are device(s) connected to the app. When this method is called, it will get the connected peripherals and their **SavedPeripheral** objects to send to the **ConnectedDeviceViewController**. This new view will then be pushed to the screen.

**discoverServices():**

This method is used for determining what services a peripheral has. Some peripherals that it would be looking for include UART.

**registerNotifications():**

This function is used for adding notifications for certain actions. These notifications are added when the view appears and disappears. Notification are enabled for when: a device’s state has changed, a new device has been discovered, a device will be connected to, did connect to a device, did disconnect from a device, and name of a device has changed.

**didUpdateBleState():**

This method will be called if the state of a device has changed. If it changes to a state that makes it unavailable, then the reason will be displayed to the screen.

**willConnectToPeripheral():**

This method will be called when a device is being connected to. It will update the screen to let the user know a device is being connected to.

**connect():**

This function is used to connect to BLE devices. It will enable the **nextButton** so the user can interact with the device. It will also update the table to remove it from **availableDevices**.

**disconnectAll()**:

This function will disable the **nextButton** since no devices will be connected. It will also disconnect from all peripherals and then update the table to reflect the change.

**didConnectToPeripheral():**

This function will check that the peripheral was connected successfully. It will then discover what services are available from that device.

**didDisconnectFromPeripheral():**

This function will clear any updates that are waiting to show up for a device. It will also reload the table to make sure the info is up to date.

**onFirmwareUpdateAvailable():**

This function will display information regarding whether or not there is an available update for the BLE device. This is the extent of updating. Most of the other firmware updating has been stripped from the app.

**savePeripheralPrompt():**

This function is used for saving information about a device prior to connected. This function will ask the user what name they would like the device to be saved under. Depending on their response, the app will either save the new name or the advertised name using the **saveAndConnect** function or will cancel and not continue.

**saveAndConnect():**

This function will take the name that is passed in and the **BlePeripheral** object and save them. This function does this by creating a new **SavedPeripheral** object and setting some preferences. After that, it is added to the **savedDevices** and saved using the **PersistenceService**. The device is then connected to.

##### 3.1.2.2.2 AvailableDevicesTableViewCell.swift

This file is used for setting some preferences for how the available and connected devices will be presented in the table of **AvailableDevicesViewController.** The object this file contains is a **TableViewCell**. The design of the cell contains a main title and a subtitle underneath. The constraints for this object are set up in the **SetUpConstraints** method.

#### 3.1.2.3 SavedDevicesControllers

The controller that is within this folder is responsible for the saved devices tab in the **MainPageTabBarController**.

##### 3.1.2.3.1 SavedDevicesViewController.swift

**viewWillAppear():**

Every time this page is opened up, the app will update the **savedDevices** to see if there have been any changes.

**viewWillDisappear():**

This function will mark the data as dirty whenever it is navigated away from. This should ensure that the data is always up to date.

**getSavedPeripherals():**

This function will use a **NSFetchRequest** and the **PersistenceService** to gather all the saved devices and save it in **savedDevices**. It will also reload the **tableView**.

**tableView(cellForRowAt):**

This function will generate the cell for each saved device. If there are no saved devices, it will just display a cell indicating that this is the case. The table cell is just a generic **UITableViewCell**.

**tableView(commit, forRowAt):**

Whenever a device is deleted from the **tableView**, it’s saved data will also be deleted. **PersistenceService** will remove the saved data and the instance will be removed from the **savedDevices**.

**tableView(didSelectRowAt):**

Whenever a device is selected, a **DeviceInfoViewController** will be created and displayed. The intricacies of how this controller displays information will be discussed later.

#### 3.1.2.4 PastDataControllers

This controller is used in combination with other controllers in **ConnectedDeviceViewControllers** to display previously saved data.

##### 3.1.2.4.1 PastDataViewController.swift

**viewDidLoad():**

Whenever this view is opened, it will gather past data, get saved devices, and set up the UI.

**gatherData():**

This function will gather past UART and Plot data. Both sets of data are gathered using **NSFetchRequest** and **PersistenceService**. Once the data has been collected, I iterate through both sets of data and add any UUID of a device to the list **uuids**. This data will let me know what devices to display to the user.

**getSavedPeripherals():**

This function will gather all previously saved devices and their data. This function does the same as seen in **SavedDevicesViewController.**

**tableView(cellForRowAt):**

This function will set the cell for a given row in the **tableView.** If there is no data, then it will tell the user that. If there is data to display, then by using **uuids**, the name of the device from **savedDevices** is found and displayed for the user.

**tableView(didSelectRowAt):**

If a device is selected, information about this device will be gathered from the **uuids.** This data will then be sent to the newly created **SavedDataViewController** prior to displaying this new controller to the screen. The data displayed in this controller will be discussed later.

### 3.1.3 ConnectedDeviceViewControllers

The view controllers in this section are responsible for all options when device(s) are connected to the app.

#### 3.1.3.1 ConnectedDeviceViewController.swift

This view controller controls how the app will interact when device(s) have been selected, are connected to, and now the user wants to interact with.

**viewDidLoad():**

When the view is about to display, the status of UART across the connected devices is checked, the UI is set, and table cells are registered. Registering table cells makes it easy to set a consistent style for a given type of data. This will be discussed later.

**viewWillAppear():**

This function also does some initialization. It will first see how many devices are connected. Then, it will register some notifications that will be triggered on given events. Next, **rssiRefreshTimer** is created so that the status of the RSSI between the app and the device can be updated on a set schedule. Finally, the **tableView** is updated.

**viewWillDisappear():**

This function is called whenever the view is going to disappear. It will deactivate notifications and will disable the **rssiRefreshTimer** so that it no longer continues to update.

**rssiRefreshFired():**

This method is called by the refresh timer. It will go through all the connected devices and update the RSSI values. If a value of RSSI changes, then **peripheralDidUpdateRssi** will be called.

**checkUart():**

This method is used for checking to see whether or not all devices are UART capable. Different modes will be available depending on whether or not UART is available for all devices.

**DefineModes():**

This method is responsible for determining what modes will be available to the user according to whether or not UART is available and whether there are multiple devices connected.

**tableView(titleForHeaderInSection):**

This method will choose what header to display in a given section of the **tableView**. **LocalizationManager** is used to display a given string according to what language the device is in. The key will choose what language to display the given message in. More info on how the **LocalizationManager** does this is ignored because it will always display in English.

**tableView(numberOfRowsInSection):**

This function will determine the number of rows in each section. For the **device** section, it will be the number of devices connected. For the **modules** section, it will be whatever modes are available.

**tableView(cellForRowAt):**

This function will only choose what cell to display. The information for each cell will be determined later.

**tableView(willDisplay):**

This function will determine what information to display for each cell. If it is a **device** cell, then it will display information about a device that is currently connected. If it is a **module** cell, then it will just display the module that is available.

**tableView(didSelectRowAt):**

This function will handle what happens when a cell is selected. If it is a **module** that is selected, then it will see what module was selected, gather the necessary information, and then will display the modules view controller.

**registerNotifications():**

This function will enable/disable notifications for when a peripherals RSSI is updated and when a peripheral disconnects.

**peripheralDidUpdateRssi():**

Whenever a peripherals RSSI is updated, this method is triggered. It will then trigger the **tableView** to reload. This will update the RSSI image that is displayed with each connected device.

**didDisconnectFromPeripheral():**

This function handles whenever a device is disconnected. If only a single peripheral was connected, then the app returns back to the peripheral scanning view controller. If multiple peripherals were connected and there are peripheral(s) still connected, then it will remain on the page. If it going to remain on the page, then UART capability is checked again and then the **tableView** is reloaded.

#### 3.1.3.4 DeviceInfoController

These files are responsible for displaying information about a saved device.

##### 3.1.3.4.1 PeripheralInfoSelectorViewController.swift

This file is used when there are multiple devices connected and the user needs to select what device they want to either see information about or see past data from.

**tableView(cellForRowAt):**

This function will simply just display the connected devices that were sent to the controller from the **ConnectedDeviceViewController.**

**tableView(didSelectRowAt):**

If the **ConnectedDeviceViewController** set the **PeripheralInfoSelectorViewController** in **infoMode**, then when a cell is selected, then a **DeviceInfoViewController** will be created so that specific information about the device can be displayed. If it is not in **infoMode,** then it will create a **SavedDataViewController** so that it can display past data.

##### 3.1.3.4.2 DeviceInfoViewController.swift

This controller is used by multiple other view controllers in order to display and edit specific information about a saved device.

**genericConstraints():**

This function is used by **setupUI** to produce generic constraints for many of the visual elements. All it does is set flexible constraints for each of the elements. These constraints are buffers on either side of the elements and keeping the width a constant size.

**save():**

This button is connected to the **saveButton** that appears at the bottom of the screen. Whenever the button is pressed, it will save the name and the notes sections. **PersistenceService** is used to save the new information for the **savedPeripheral**.

**onKeyboardPositionChanged():**

This function is called whenever the keyboard is either opened or closed. Whenever the function is called, it will alter the constraints of the view that contains all the visual elements. If the keyboard is opened, then the size of the view is shrunk so that all the elements are still visible. If the keyboard is closed, then it will set the views size back to the entire screen.

#### 3.1.3.5 ButtonsControllers

This controller is responsible for the buttons module. At the time of the writing, it is only supported under single peripheral mode.

##### 3.1.3.5.1 ButtonsViewController.swift

This view controller is used for creating buttons to easily send the same message to a device repeatedly.

**viewDidLoad():**

Whenever the view will be loaded, the first thing done is retrieve all saved commands from the **savedPeripheral**. Next, the UI is setup and a **UartPacketManager** is created to handle the sending and receiving of any data.

**viewWillAppear():**

Right before view is set to appear, this function will setup UART for any available devices and setup notifications.

**viewWillDisappear():**

Whenever the view is going to be removed, all commands are saved and the notification are disabled.

**setupUART():**

This function is used for enabling UART for one or more devices. If it is connected to multiple peripherals, then for each peripheral that has UART, it will call **uartEnable** on the **BlePeripheral** object. It will also set **onUartPacket** to handle any responses from the device. If these are completed without error, the next peripheral will be processed. If there is an error, then it will alert the user that there was an error setting up UART. The same process is followed for a single peripheral.

**send():**

This method will send a message to all connected peripherals. This is done using the **UartPacketManager** and the **send** command.

**onClickAdd():**

This function is called whenever the **addCommand** button in the **navigationController** is pressed. It will first prompt the user to enter a name for the command and the actual command using a **UIAlertController**. Once the new command and name have been entered, it will then add it to the list of commands and update the **tableView** of commands.

**save():**

This method is used to save the list of commands. The commands will be saved with the current **savedPeripheral.** Saving is done with the **PersistenceService**.

**registerNotifications():**

This function is used to set up a notification. This notification is set up when the view is opening and shut down when the view is being removed. The notification will be triggered whenever user preferences are changed. When this occurs, it will just reload the **tableView**.

**tableView(cellForRowAt):**

This method will set the configuration for each cell in the **tableView.** Each cell is one of the saved commands. The cell is configured to be a **subtitle** cell with a main label and a smaller label underneath. This type of cell is a default type of cell. The name of the command is set to be the main text and the actual command is set to be underneath.

**tableView(didSelectRowAt):**

Whenever a user selects a given row, this function will find what command was selected and then it will send the saved command message. Before the message is send, an end of line character is attached to the end of the message.

**tableView(commit):**

This function is called whenever a cell in the **tableView** is being deleted. Whenever the command is being deleted, it will be removed from the list of **commands** and then the table will then be reloaded.

**onUartPacket():**

This function is used for whenever a message is received from the device. This function will be called but won’t do anything. I left the stub there for future use.

#### 3.1.3.6 SavedDataControllers

This set of controllers are used for displaying previously saved data. Currently, the controllers can display both UART and Plot data.

##### 3.1.3.6.1 SavedDataViewController.swift

This view controller is used for displaying what data is available for the user to view. It will show the available Plot data and UART data for the given peripheral.

**viewDidLoad():**

The first things the controller does is get the set peripherals UUID. Next it gathers the saved data for that peripheral. Finally, it sets up the UI.

**gatherData():**

Similar to other functions previously described, **NSFetchRequest** and **PersistenceService** are used to get the UART and Plot data. Once the data has been gathered, it is filtered to only contain the data with the matching UUID as the desired peripheral.

**tableView(cellForRowAt):**

This method will build a **subtitle** cell for the given row. For both types of data, the title will be the name given to the data when saved and the subtitle will be the date it was saved.

**cantDisplay():**

If a cell is selected, but for some reason the data can’t be displayed, then this function will use a **UIAlertController** to let the user know that the data is corrupted.

**tableView(didSelectRowAt):**

If a cell is selected, then this function will find the data that was selected and then open a view to display the saved data. If the data is **plot** data, then the name of the data, the plot data, and the data in the form of a string are all passed to the new controller. If the data is **uart** data, then the string, name, and date of the data is all collected and passed to the new controller. For both types of data, a **DisplayDataViewController** is used.

**tableView(commit, forRowAt):**

If a cell row is deleted from either section, then the data needs to be deleted from the saved data. For both **plot** and **uart** data, the data is taken from either **plotData** or **uartData** and then deleted using **PersistenceService**. The data is then removed from the lists they were found in. That row is then deleted from the **tableView** and the **context** is then saved using **PersistenceService**.

##### 3.1.3.6.2 DisplayDataViewController.swift

This view controller is used to display saved data. This controller will either show plot data or UART data.

**setUpUI():**

This function is used for setting up the visual aspects of the view. First, an export button is added to the top right of the **navigationItem**. Whenever this button is pressed, it will call the **onClickExport** function. Next, the data component will be created and displayed. In the case of **plot** data, a **PlotPagesView** will be created. This view will be initialized with plot data. If the data is **uart** data, then a **UITextView** will be created to display the UART communication as a string in the view.

**onClickExport():**

This method is called any time that the **exportButton** is pressed. This will use the static class **ExportData** to export the data.

#### 3.1.3.7 DataStreamControllers

These controllers are used for displaying any plot data, whether it is being streamed or it is from saved data.

##### 3.1.3.7.1 DataStreamContainerViewController.swift

This function is used for streaming data from peripheral(s) and displaying the data on graph(s).

**viewDidLoad():**

When the view is going to load, this method will do a lot of initialization. First it will set up the UI. Next, it will create a **UartDataManager** to manage communication between peripherals. It will then take a snapshot of the peripherals that are connected on initialization. The scroll bar and slider will be initialize and targets added to react for when these are being interacted with. Initial configurations for these UI components will then be set.

**viewWillAppear():**

When the view is about to appear, UART with the peripherals will be set up.

**removeOldDataOnMemoryWarning():**

If the app receives a message about a memory warning, then this function will clear some old data to make some space.

**configureUI():**

This function sets up the UI for this view. Multiple buttons are initialized. The **saveButton** is used for when the user clicks the button to save received data. The **onClickSave** handles the saving. The **startCommandButton** is set up and used to send start commands to the peripherals. The **stopCommandButton** is used to send stop commands to the peripherals. The **exportButton** is used to export all the current data. Next, the **barButtons** function is called to set the bar buttons appropriately. Finally, the new **PlotPages** view is initialized and set up in the middle of the screen.

**barButtons():**

This function is used to determine what bar buttons to display at a given time. If the data is current being streamed, then **stopCommandButton** will be shown. If it is not streaming, then the **startCommandButton** will be displayed.

**setUpUART():**

This function will initialize UART for all UART capable devices. It will use the **uartEnable** function from **BlePeripheral** to initialize UART. If there is an error, the error will be displayed to the user. Also, a line dash will be assigned to each peripheral so that it can be delineated from other peripherals. Lastly, **dataCounter, currentPlot,** and **basicDataSet** are all initialized for the peripheral. The same is done for either multiple peripherals or single peripherals.

**addEntry():**

This function is responsible for adding single data point that is received from a peripheral. A **ChartDataEntry** is first created for the new data point. Next, the function will check whether or not this data point is being added to an existing data set or if a new data set needs to be created. A single data set corresponds to a single peripherals data for a graph. There are multiple data sets for a graph if there are multiple peripherals.

If the data point is being added to an existing data set, then it is just appended to the appropriate data set for the corresponding peripheral. If a new data set needs to be created, then **addDataSet** is called. Then the new data set is added to the basic data set of only doubles. If it is a new data set, then the new dataset is added to **plots**. Lastly, for both cases, the **lastUpdatedData** will be updated so it can be updated from the **plots.**

**addDataSet():**

This function creates a new data set if the data point received does not already belong to a data set. The entry is added to a newly created **LineChartDataSet** object. Next, some settings are set for the data set and a color is assigned to the dataset. Finally, this data set is added to the dictionary of data sets for the peripheral.

**onXScaleValueChanged():**

This function is called whenever the slider is changed by the user. When the slider is changed, a new visible interval is calculated using the value and the **plots** are updated accordingly. Finally, **notifyDataSetChanged** is called on the main thread.

**onAutoScrollChanged():**

This function is called whenever the auto scroll button has been toggled. The function will notify the **plots** that this has changed, and the update will be propagated over the main thread.

**onClickSave():**

Whenever the save button has been pressed, this function will be called. It will then attempt to save all data that has been received thus far. The data will be saved to each peripheral and not altogether. So, an alert is presented to the user asking them to name the data to be recognized later. Once the name is set, then each peripheral will be iterated through, a new **PlotData** object will be created and will be saved using **PersistenceService.**

**onClickStart():**

This function is used to let the peripherals know that they need to start sending data to the app. The app will first ask how many runs and how many samples should be collected. This input will then be validated and if it is not acceptable, then **invalidInput** will then be called with a specific message. Then the **autoScroll** button is then enabled and the **plots** initialized. Finally, messages are sent to every peripheral that is connected to the app. The format for these messages are: “t<samples>” for the number of samples and then “r<runs>” for the number of runs.

**invalidInput():**

This function is used to display an alert to the user with a given message that is passed in. The only option is for the user to select dismiss.

**onClickStop():**

If the user wants to stop the peripherals from sending any more data, then they will click the **stopButton**. This button will trigger this method. If the user confirms that they want to stop reading in data, then “s” will be sent to all devices to let them know that they need to stop gathering data. Bar buttons are then adjusted to reflect the stop.

**onUartRx():**

This function is called whenever data is received from a peripheral. The data is first gathered from the **dataManager**. The data is then filtered to retrieve only the viable pieces of information. Then on the main thread, every line that is received is processed. So, each data point received is paired with a value in the order it is received for that data set. This will evenly space out each data point and will put it along the X axis starting at 0. Next, the data point that is received will be checked. In this context, a value of 0 is used to delimit each plot send by the peripheral. Next, if the value is 0 and there has yet to be any other data received yet, then the app won’t do anything and just proceed to the next. If the data received is after when the last data point should have been received, then it will just return. If it is a zero and other data has been received previously, then **currentPlot** is incremented and **dataCounter** is reset to zero. Also, if the new plot that it is going to move to is farther along than any other plot, then the app will shift the **plots** over. If the data point received is any other value, then **addEntry** is called to add it to the data set. Then, if a new data set is going to be created, it is added to the **basicDataSet.** Finally, **enh\_throttledReloadData** is called. This will call a reload data on the plot. This method is called instead of reload data because if data is coming in too fast, then it will cause the app to crash due to too many reloads. This will throttle the reloads down to a level that the phone can handle.

**send(message):**

This function will send a specific message to a single peripheral. The peripheral it will send it to is the peripheral that is set as **blePeripheral** at the very start.

**send(message, peripheral):**

This function is used to send data as well. The only difference is that the peripheral that the message needs to be sent to is specified as an argument.

##### 3.1.3.7.2 PlotPagesView.swift

This view controller is used to control the different plots that are needed for the **DataStreamContainerViewController.** A **UIScrollView** and **UIPageControl** are used to manage flipping through plots.

**init():**

This init method is used when there isn’t any data yet and a message that there is no data needs to be displayed.

**initialize(count):**

This function will first clear any items from the view and then will set up the UI with the given number of plots.

**initialize(data):**

This function will first clear all the data from the view. Next, it will add preloaded plots to the view. This method would be called by a saved data view controller to display data that has already been collected.

**clean():**

This function will remove all necessary subviews from the current view. This is necessary if new data is going to come in or if the empty plots need to be added.

**setupUI():**

This function will set up the plot pages whether or not there is data to initialize it with. It will also initialize the **pageControl**, which will control changing the pages when the next plot has been reached.

**addPlots():**

This function adds plots to the **scrollView**. First, a **LineChartView** is formatted and returned using the **formattedPlot** function. This plot is then added, visually, to the end of the existing plots. Finally, if there is data available, then it will be added to the plot by using **createPlotDataSet**.

**createPlotDataSet():**

This function takes in the data set and creates **ChartDataEntry**s for each and adds it to a **LineChartDataSet**. Some preferences are then set for the data set and it is added to the plots.

**changePage():**

When this function is called, it signals that the page needs to be changed. The next page is calculated and then the **scrollView** is shifted so it is visible to the user.

**notifyDataSetChanged():**

When this function is called, it signals that the data has been changed and that the plot needs to be updated. So all the data and the visible range is reloaded to be up to date. Finally, if autoscroll is enabled, then the plot will be shifted over accordingly.

**addDataSet():**

This function will add data to a given plot. This function would be used if there is new data received that needs to be put onto a new plot.

**changePage():**

This method will move the page over one to the right. This is useful when a new plot of data is being received and the new data needs to be shown to the user.

**changePage(toPage):**

This method is similar to the one above; however, this one will change to the specific page.

#### 3.1.3.8 UARTControllers

These controllers are used for managing the UART module between the app and the device.

##### 3.1.3.8.1 UARTBaseViewController.swift

This controller outlines how the UART module shall be ran. Lot of the details are left for the **UARTViewController** to specify.

**viewWillAppear():**

This method is called before the view is set to appear. It will enable notifications, configure the UI, and setup UART.

**viewDidDisappear():**

This function will disable notifications just before the view is set to be removed from the users view.

**reloadDataUI():**

Similar to the data stream, the incoming data can cause the app to crash. So, throttling the reloading of data is necessary. This function is called when the UI is updating. It will format the strings and will add it to the **UITextView.**

**onClickSend():**

This method reacts whenever the send button is pressed. It will take the string that is in the text box, append an end of line character, and it will send it to all the peripheral(s). Finally, it will close the keyboard if it is open.

**onClickClear():**

This will clear all UART packets and will erase all the data from the **UITextView.**

**onInputTextFieldEditingDidEndOnExit():**

This method is called whenever the send button on the keyboard is pressed by the user. This is different from the button that is present on screen. This will send the message in the same manner as the **onClickSend** method.

**onClickSave():**

This method is called whenever the save button is pressed by the user. All the data that is present in the **UITextView** will be saved to the corresponding peripheral(s). A **UIAlertController** is used to get a new for the data. Once the name is entered, it will save it using the **save** method.

**save():**

This method will create a **UARTData** object and save it to the peripheral. This saving is done using **PersistenceService**.

**onKeyboardPositionChanged():**

Similar to other views in the app, the visual view needs to be shifted whenever the keyboard is opened. This will adjust the frame size of the view so that it allows the user to still see every component on the view even when the keyboard is open.

**onUartPacket():**

This function is called whenever data is received from a peripheral. A throttled reload is also called to make sure the data is added to the view.

**onUartPacketText():**

This function is responsible for formatting the data whenever it is received. This function will change the font and color depending on whether it is received or sent and what peripheral it was received from. After this formatting is done, it is added to the view.

##### 3.1.3.8.2 UARTViewController.swift

This view controller lays out more of the specifics that the above view controller doesn’t specifiy.

**viewDidLoad():**

Whenever the view loads, this function will initialize the **UartPacketManager**, which will manage the incoming messages. Also, the **sendButton** and **clearButton** will be initialized.

**setupUart():**

The setup of UART is identical to what was done in the **ButtonsViewController** in section 3.1.3.5.1. The only difference here is for each peripheral, a different color is assigned so that is it easily recognized from the others.

**send():**

This function is used to send messages to all peripherals that are currently connected. It will keep track of what messages are sent and what info is added to the **UITextView** so that messages to multiple peripherals don’t show up multiple times in the view.

## 3.2 Common

The files contained in this section are used in multiple different areas of the app. They are kept in here so that it is clear that these are necessary for not just one controller.

### 3.2.1 ExportData.swift

This file is responsible for exporting data to a file. Currently, the only format supported is txt file.

**exportData(view, button, data):**

This method is called whenever a view controller is trying to export data. This function will present an alert to the user and ask the user to provide a filename. Once the filename has been provided, then the function below will be called.

**exportData(view, button, data, filename):**

Now that this function has a filename, it will attempt to save the information. So, the function will ask the user what format the user would like to save in (only txt right now). Once the format is selected, it will parse the data and then export it.

**parseData():**

This function will take in the data to export and will format it to the specific type. Once it figures out what type of data has been passed through, it will format it. **PlotData** has it’s own formatter and **UARTData** is just string so it doesn’t need to be formatted. If there are multiple peripherals with **PlotData**, then the data will be put in one file with the device name separating the data.

**exportWithView():**

This function uses **NSTemporaryDirectory** in order to allow the user to choose where to save or send the exported data. Once the path has been created, it will open a **UIActivityViewController** to allow the user to choose where to save the data.

### 3.2.2 DataFormatter.swift

This file is primarily used as a helper for the UART communications. It is responsible for handling the data before it is presented to the user.

**stringFromData():**

This method is responsible for filter the data received from the peripherals. It will go through the characters received and it will filter out any of the unreadable values. It will make sure that all values present are readable.

**attributedStringFromData():**

This method will build a string with different color and font characteristics so that it is easy to determine what peripheral sent what on the view. Again, this is easiest to see when using the UART functionality.

### 3.2.3 PersistentData

These files are used for saving data even when the app has been closed. This was made possible by using **CoreData**. **CoreData** is necessary in any file that wants to access saved data.

#### 3.2.3.1 PersistentData.swift

This file is responsible for managing some of the saving of data. This helps with making sure data is available even when the app closes.

**encode():**

This function is just used to prepare an object to be saved. Most likely, the **PersistenceService** encoder is being passed through to encode the object. Once it is encode, it can be saved and retrieved using the **PersistenceService**.

**saveData():**

This function will just save a blank version of the data. It will create a **SavedData** object and then save it with the rest of the data in the array.

**save():**

The **NSKeyedArchiver** is used to save the array of **SavedData.** If there is an error with saving the data, then it is logged using **DLog.**

#### 3.2.3.2 PersistenceService.swift

This file handles bulk of the saving of data. Frequently it is mentioned for saving or retrieving saved data.

**persistentContainer:**

This **persistentContainer** is responsible for grabbing the location for storing data. This container is where new data is put prior to being saved. The specific container it grabs is specific to the app.

**saveContext():**

This method will save any new changes that have been made to the context, or any of the saved data in general. It will check that changes have been made, and then it will save it.

#### 3.2.3.3 SavedData\*.swift

This is a parent class to both **PlotData** and **UARTData.** This object just outlines some of the components a saved piece of data must have if it is going to be saved. These values are necessary when trying to determine on piece of saved data from another.

**setup():**

This function is designed to make it easy for the app to create new saved data objects. Just be getting a name, or id, for the data and the **BlePeripheral** object, it is able to parse all the required information.

#### 3.2.3.4 SavedPeripheral\*.swift

**SavedData** and **SavedPeripherals** are saved separately. Even though they are separate, they share some common attributes so they can be associated. This file outlines the attributes that should be set prior to saving.

#### 3.2.3.5 PlotData\*.swift

This object is a child class of **SavedData**. It has almost all the same attributes, except for its data is in the format of an **NSObject.** This is because the data needs to be stored as a 3D array, instead of a String in **UARTData**.

**dataToString():**

This function is used to make it easy for the developer to get the String representation of the data. It will just put each data point on a new line and if it hits a new plot, it will separate it with a 0.

#### 3.2.3.6 UARTData\*.swift

This file is a subclass of **SavedData.** It is used to represent the communication between peripheral and app. Only note about this app is that the data is a String type. This is different from **PlotData** and needed to be differentiated.

### 3.2.4 AdafruitKit

These files were used from the Adafruit open source app. These files came in handy when trying to communicate back and forth with the device. Two of the main files are briefly described below.

#### 3.2.4.1 Ble

##### 3.2.4.1.1 BleCentralMode

###### 3.2.4.1.1.1 BleManager.swift

This object was helpful in managing all the communications between the peripherals and the app. In particular, it was helpful for scanning devices, connecting to devices, and viewing what peripherals were connected. The main functions that can be used and are most helpful are: **connect, disconnect, refreshPeripherals, startScan,** and **stopScan.** These can easily be used without knowledge of the underlying implementation.

###### 3.2.4.1.1.2 BlePeripheral.swift

This object was used to represent a BLE peripheral. This object encapsulates all the advertised information about the device. It was extremely helpful when discovering what characteristics were available, whether it had UART, what the RSSI value was, and actually setting up UART. Similar to the **BleManager**, this object and its methods abstracted many of the fine details and made it very easy to use.

# 4.0 Conclusion

This documentation is designed to bridge any gap between developers. The explanation of each important file and the important methods within each should hopefully bring some clarity to the design flow. If questions do arise, an excellent resource is the Adafruit BLE app [1]. If questions can’t be answered by this resource, feel free to reach out via email at [sdp38@pitt.edu](mailto:sdp38@pitt.edu).

# 5.0 References

[1] https://github.com/adafruit/Bluefruit\_LE\_Connect\_v2