

# Multi Comm Terminal

Hobbyist Terminal Program to communicate with Embedded systems

By Michael Roop

## Table of Contents

Introduction .....	4
Supported Mediums .....	4
Data Storage.....	5
Lists of Commands .....	5
Serial Configurations .....	5
Terminators.....	6
WIFI Credentials.....	6
Languages .....	6
Settings.....	7
User Interface .....	7
Buttons.....	7
Wordless Buttons.....	7
Run Window.....	7
Elements .....	8
Main Window.....	10
Elements: .....	10
Menu .....	11
Bluetooth Run Page .....	11
Info .....	12
Settings.....	13
BLE Run Page (Bluetooth Low Energy).....	15
Info .....	15
Settings.....	19
Special Note .....	22
WIFI Run Page .....	23
Info .....	23
Settings.....	24
USB Run Page .....	26
Info .....	26
Settings.....	27
Ethernet Run Page (TBD) .....	29
Info .....	29

Settings.....	30
Terminators.....	31
Adding a terminator sequence set.....	32
Edit Terminator set .....	33
Deleting a Terminator set .....	33
Commands .....	35
Adding a command list .....	35
Language .....	37
Code Samples.....	38
Appendix .....	39
Introduction to Embedded Messaging .....	39
Arduino Bluetooth Sample.....	39
Arduino BLE Sample .....	42
Arduino WIFI Sample .....	47
Arduino USB Sample .....	50
Arduino USB Sample 2 for AT Commands .....	52
Arduino BLE Sample .....	54

## Introduction

This program will allow users to communicate with embedded devices such as Arduinos, Raspberry Pie, and others using different communication mediums.

## Supported Mediums

- Bluetooth Classic:
  - Connects to a device which implements a Bluetooth serial access point.
  - Tested against an Arduino Uno with an ITEA Bluetooth shield with an HC-05 module.
  - An Arduino code sample is provided
- BLE (Bluetooth Low Power):
  - Connects to a device which implements BLE Characteristics for inputs and outputs.
  - Tested against an Arduino WIFI Rev2. With forked ArduinoBLE branch
    - <https://github.com/unknownconstant/ArduinoBLE>
  - An Arduino code sample is provided.
- WIFI:
  - Connects to a device which implements a WIFI Access Point.
  - The access point shows up as an SSID like any router.
  - The device also must provide a socket address and port.
  - Tested against an Arduino WIFI Rev2.
  - An Arduino code sample is provided.
- USB:
  - Connection via hard wired USB port
  - Tested against an Arduino Uno
  - An Arduino code sample is provided
- Ethernet
  - Connection via hard wired Ethernet cable
  - Tested against a Sunfounder Ethernet shield on an Arduino Uno

## Data Storage

Data storage root depends on if running as a Windows Application or a Win App Desktop Application (From the App Store). Currently I am only releasing via the Windows App store so it should not be a problem

- Windows Desktop Application: `C:\Users\UserName\AppData\Local\MultiCommSerialTerminal`
- Win App:  
`C:\Users\Michael\UserName\Local\Packages\LongGUIDSstring\LocalCache\Local\MultiCommSerialTerminal`

An example of the long GUID string would be: `1c29e46d-d1d0-4f21-acbf-d63f3611c3a2_k00brw8eakxmj`

Data is stored in various directories under that root. The only time you might ever want to access it directly is to view the debug logs to report errors

All contents are in plain text in JSON format. Except for the credentials files which are encrypted. Do NOT change the file names as you will lose access to the data.

An interesting side note is that if the Windows Desktop directory tree is already created, the Win App will store the data in those directories instead of the normal Win App directory. Also, the data will not be removed if the App is deleted. I have noticed some bizarre side effects with this so if you ever get a desktop version you would do well to delete the App first and vice versa, making sure to delete the desktop storage tree

These are the current sub directories

- Documents: Has a copy of the User Manual. You can access it through the App
- Logs: Up to 10 log files that contain data from running the App
- Scripts: Command sets
- SerialConfigurations: The serial configuration for USB connections
- Settings: Current App Settings
- WifiCredentials: WIFI connection credentials and parameters

## Lists of Commands

The user can create, store, and retrieve lists of ASCII based messages. The messages can be sent to the embedded device once connection is established. The embedded device can interpret those character strings as a command to launch certain actions. This allows the user to control the device remotely. For more information see the Intro to embedded messaging in the appendix

## Serial Configurations

On first connection to a USB device on COM port  $n$ , with a USB vendor id of  $n$  and a USB product Id of  $n$ , a dialog comes with the configuration of the device as read from the device. The user can modify those and save the results in the serial configurations directory

## Terminators

The user can create, store, and retrieve end of message terminator sequences. The terminator(s) will be automatically added to the end of the outgoing message. The terminators are non-printable characters which tell the embedded device that the full message has been received. The device must also add those terminators to the end of any response or other synchronous messages from the device.

## WIFI Credentials

On first connection, the WIFI password, as well as the host name (or IP address) and service port are entered and saved if the connection is successful. There is functionality to go back and edit or delete those stored credentials. They are encrypted

## Languages

The App currently supports 21 languages in most areas of the UI:

- Arabic
- Bengali (India)
- Chinese (Simplified)
- Czech
- Dutch
- English
- French
- German
- Hindi
- Indonesian
- Italian
- Japanese
- Korean
- Polish
- Portuguese
- Romanian
- Russian
- Spanish
- Turkish
- Ukrainian
- Vietnamese

I followed a simple approach of single word descriptions for headers and buttons. I used the Microsoft Technical Language Portal to get the translation. See: <https://www.microsoft.com/en-us/language/Search?&searchTerm=Automatic&langID=303&Source=true&productid=0>. This allows for a high precision in rendering of the languages.

## Settings

The file contains, among other things, currently selected preferences so that the App starts with the same selections as last time. Currently those preferences are Language, Terminators, Command list. More could be added over time

## User Interface

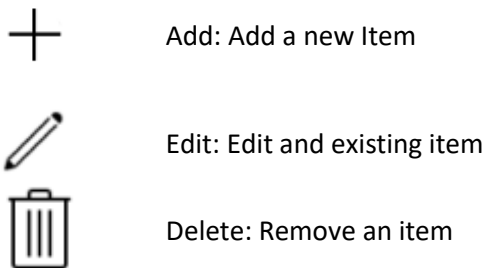
### Buttons



Buttons have both an icon and text. So, if you inadvertently switch to a language you do not understand you can still navigate back to change it.

### Wordless Buttons

These are small and used throughout. The usage should be self-evident

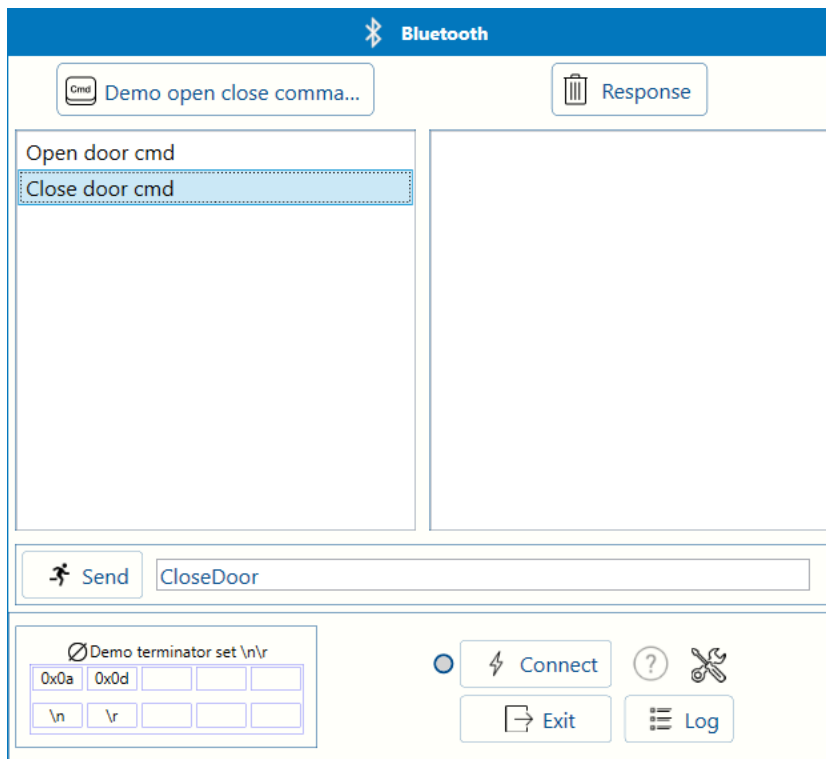


## Run Window

This is a common page format to run commands through different communication mediums. You will get this by clicking on the Menu for Bluetooth, WIFI, USB, and Ethernet. BLE has its own page format. Not all the controls are enabled on every page.

You can have one window of each communication medium open at the same time. The exception is Bluetooth and BLE. Since they both use the same radio, only one of the two can be open. If you try to open the second Bluetooth type, the first opened type will close.

I am considering allowing multiple USB, Ethernet pages open at the same time if there is a demand for such



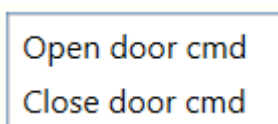
## Elements



Title. Before any device selection the name of the medium is posted. After you do a discovery and select a device, its name will be posted here next to the medium type icon.



The currently selected Command list. The name of the list is displayed. If the name is too long, it will be cut short with ellipses. If you tap on this it will bring up a selection dialog where you can change the list



Commands from the selected commands list. When you tap one, its command is posted in the Send box



This will send the command in the Send box to the device if you have established a connection. You can also free edit the contents of the Send box and send that text



Tapping on this will clear all the responses from the response list below



⊘ this is a very long name to sh...

0x04	0x02			
EOT	STX			

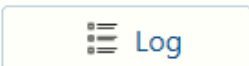
The shows the currently selected Terminators set. Tapping on this will bring up a select dialog to change the set



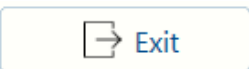
Opens a selection dialog to allow you to connect to a device. The light to the left turns green on successful connection. When you are connected the *Connect* button is replaced by the *Disconnect* button



Break any current connection. The green light shows is it currently connected



Opens a pane at the bottom to show scrolling debug information



Close the page

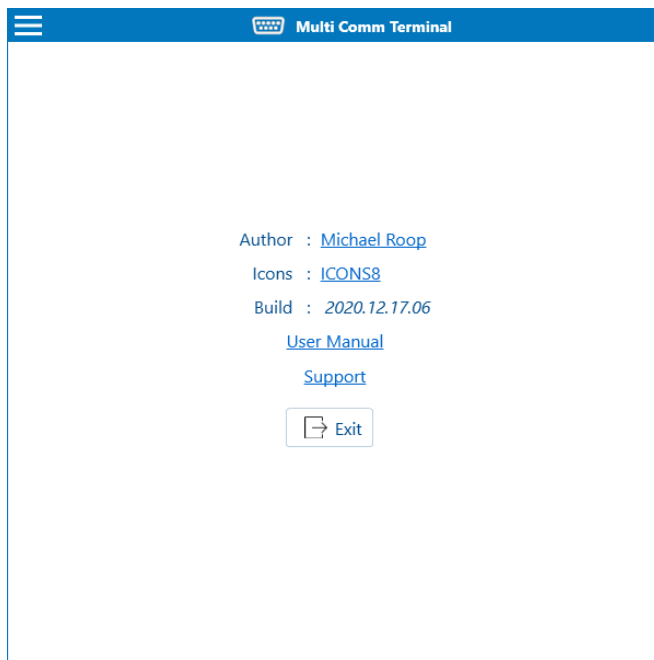


Provides information on selected device. Enabled only if connected.



Settings. Varies per connection medium

## Main Window



### Elements:



‘Hamburger’ icon on title bar. Click on it to open drop down menu where you can set language, etc.

Author : [Michael Roop](#)

Link to my LinkedIn profile

Icons : [ICONS8](#)

Link to the Icons 8 which provided the icons

Build : 2020.12.17.06

Build number for reporting errors

[User Manual](#)

Link to this user manual in PDF format










[Support](#)

Link to an email address for support



Exit the App

## Menu

	Bluetooth
	WIFI
	USB
	Ethernet
	BLE
	Terminators
	Command
	Language
	Code Samples

Select the run page for various communication mediums general settings. We will start with the general settings

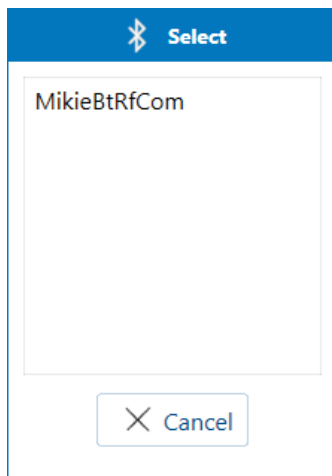
## Bluetooth Run Page

Click on the menu item

 Bluetooth

It will bring up the run page. General functionality of the run page is explained above.

Clicking on the Discover button (or info Icon if nothing yet selected) will bring up the Select dialog for paired Bluetooth Classic devices

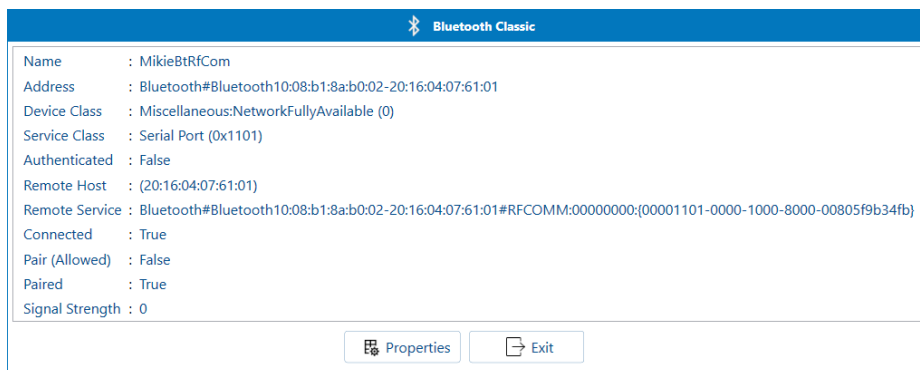


Clicking on a selection will close the dialog and set it as your selected device. You will notice that the device name is now listed on the title bar of the Run page

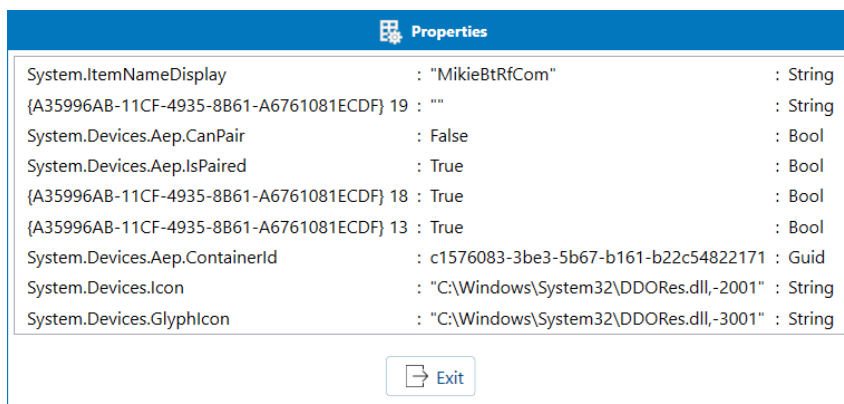


Info

Click on the info icon

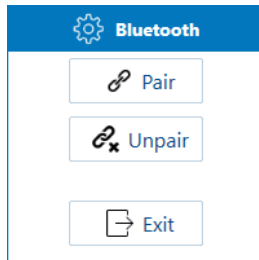


Click on exit to close, or the Properties button to show properties

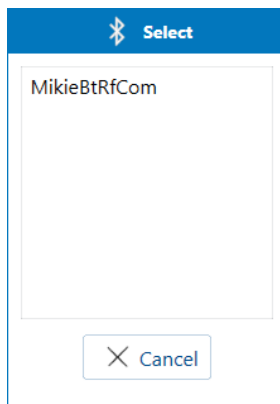


## Settings

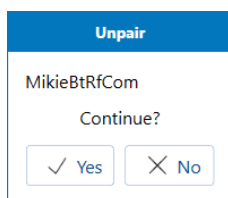
Click on the *Settings* icon



If you click on the *Pair* button you will get a list of non-paired devices. Clicking on the *Unpair* button will bring up a list of paired devices.



Clicking on a selection will bring up a decision dialog to determine whether to continue the operation. Note that the selection dialog for unpaired devices takes some time to load as it scans for any available device



The title shows you if you are going to pair or unpair a device. The name of the device is posted next, the *Continue* message. Clicking on Yes will complete the operation.

When pairing you enter your PIN

Pair Bluetooth (MikieBtRfCom)

Enter PIN

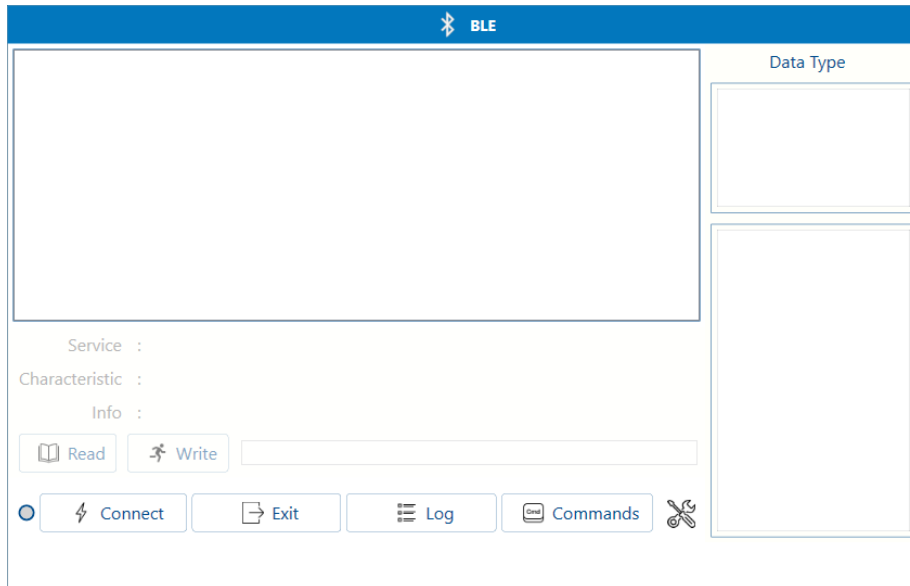
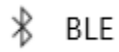
✓ OK

✕ Cancel

## BLE Run Page (Bluetooth Low Energy)

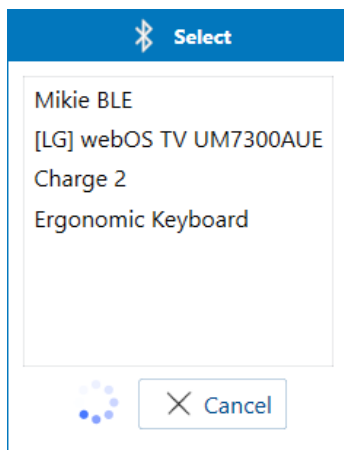
### Info

Select BLE from the menu

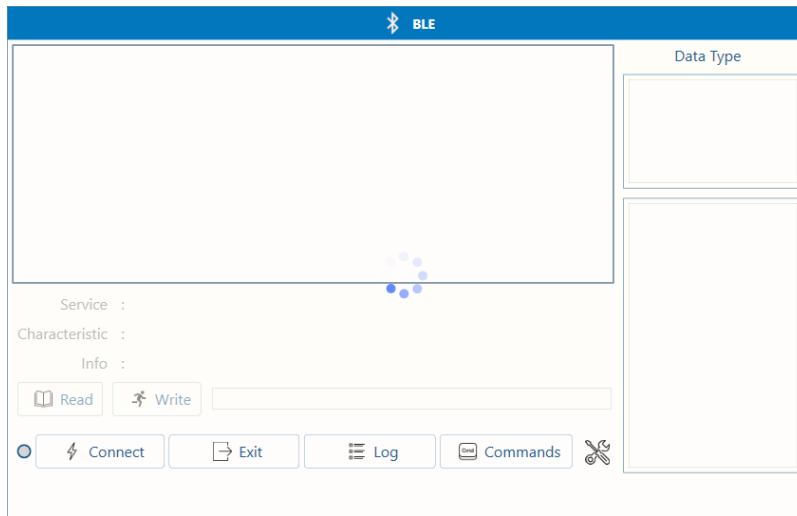


This page format differs from the other device pages. You can read and write values to and from a BLE device. See the appendix for a BLE Arduino sample.

Click on Connect o bring up the selection dialog.

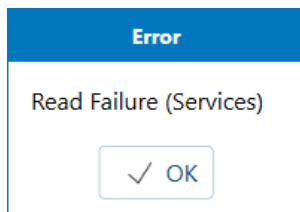


The busy spinner shows on the left of the Cancel button. It will disappear when the scan is completed. Or you can select any device showing on the list at any time to stop the scan and go back to the view page.

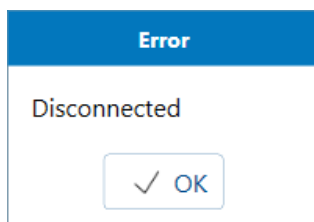


The spinner will show on the view page while the connection is completed, and all the services are loaded.

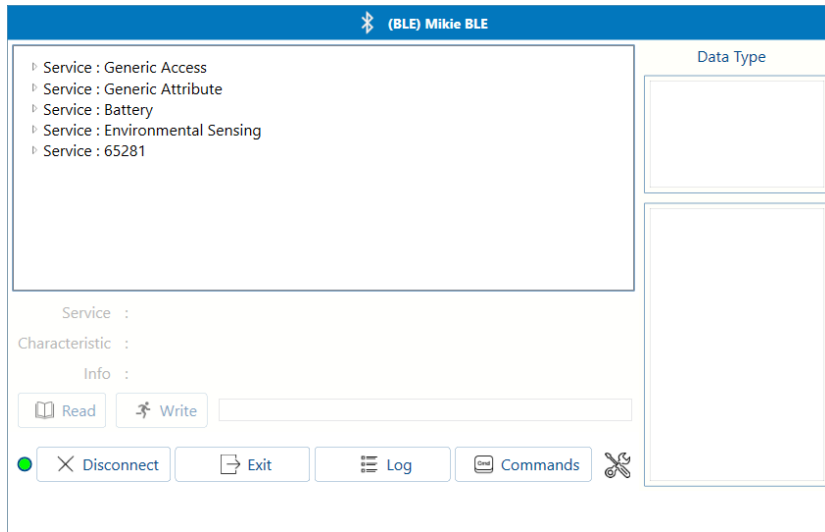
It is possible to get an error when loading the services and their characteristics. Trying again will often work.



You can also have a device disconnect at any time when connected

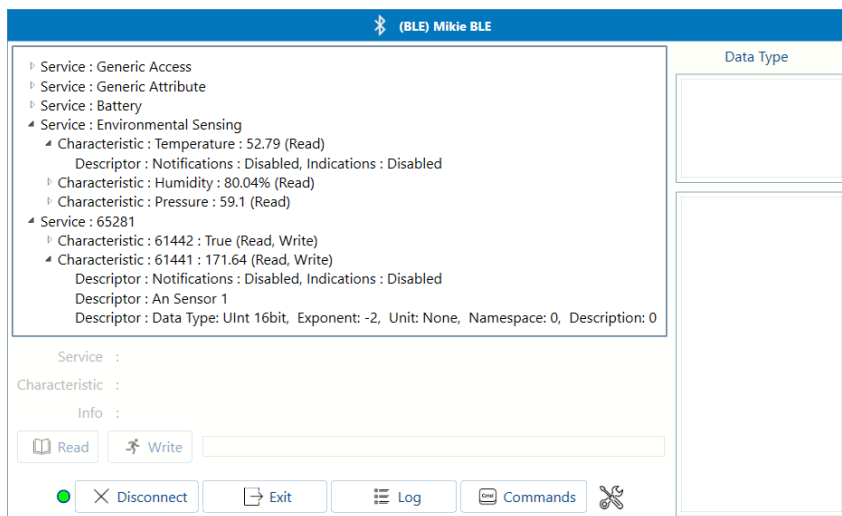




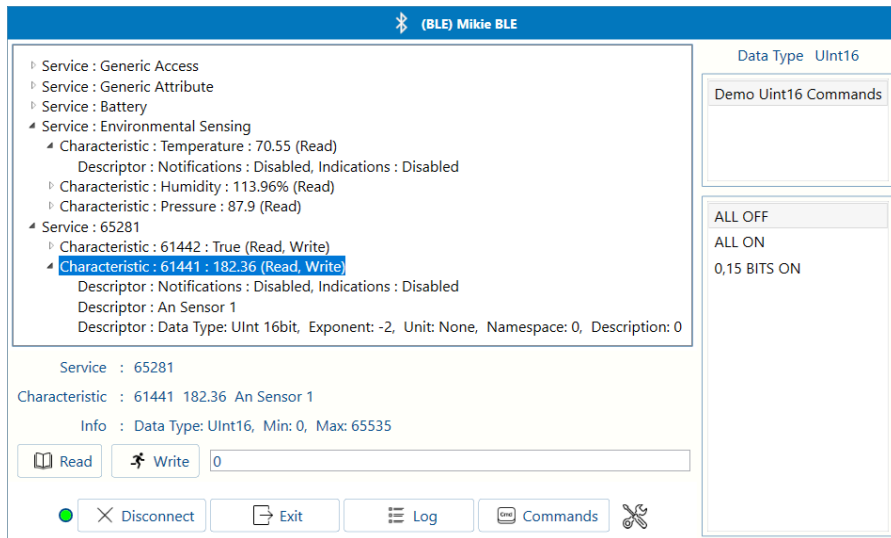


On successful completion you will get a tree of Services -> Characteristics -> Descriptors. You can drill down to look at the contents

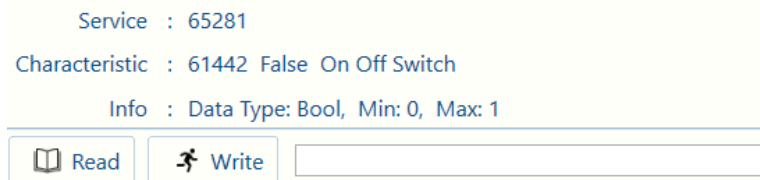
The App will subscribe to all Characteristics that allow Notifications. The display will show the data as it is received from the device.



Clicking on one of the Characteristics will select it. If it has read or write characteristics it will appear in the Read Write section below



If you have Write enabled characteristics on the device, the Write button will be enabled. If the characteristic is Read enabled, the read button will be enabled. In this example, both read and write are enabled. When you select a characteristic as above, any command sets for the same data type will be loaded in the right Command panes. You can hide the pane by clicking on the *Commands* button

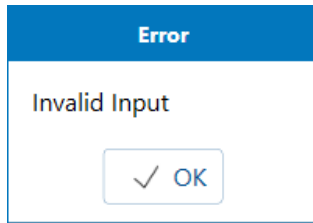


In this example a Bool data type that is Read and Write enabled was selected. The following information is posted

- Service:
  - Name, or in this case the GUID since it is not a BLE standard Service
- Characteristic:
  - Name, or in this case the GUID since it is not a BLE standard Characteristic
  - Value, in this case *False*
  - User description if provided in the descriptor
- Info:
  - Data type
  - Data values range

To write to the BLE, enter text in the send box. Or, if there is a command defined for that data type, select it and the command value will be inserted in the send box. Click on *Write* to send the command.

If you attempt to send a value out of range, you will get an error

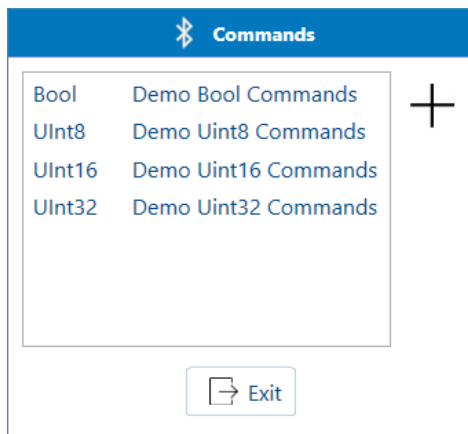


If the value is valid you will not be prompted

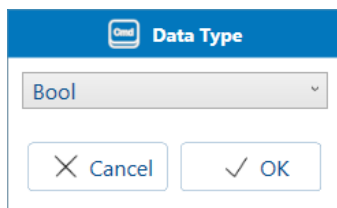
## Settings



Click on the Settings icon you will get a list of existing BLE commands.



In the left column you have the data type the commands send. On the right is the name of the list. To create a new command set, click on the Add icon. It brings up the Data Type select dialog to set for the new set. The data type cannot be changed after creation



You have the choice of Bool and unsigned integers 8bits, 16bits, 32bits. The allows you to send commands that are values or bit patterns which you can bit mask on your device. Select type and click OK. A dialog will come up with one sample command in the set.

Commands

Data Type : UInt16

Name

Open 1

+  
✎  
🗑

You need to edit the name to your liking. Nothing is saved until you click on Ok. You can click on the Add icon to create a new command or you can select the sample for edit.

Edit

UInt16, Min: 0, Max: 65535

Name

Command

Dec

Hex 0x

Bin

The name is what shows in the command pane on the run screen. The Command value (dec) is what show when you select it for send.

At the top you can see the data type and allowable range of values to enter

You can add numeric text in either the decimal (base 10), hex, or binary edit boxed. All the other boxes will be updated automatically. Of special note is the binary values if you want to create a bit pattern command for masking in the device

Edit

UInt16, Min: 0, Max: 65535

Name

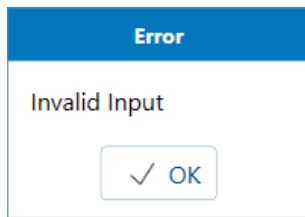
Command

Dec

Hex 0x

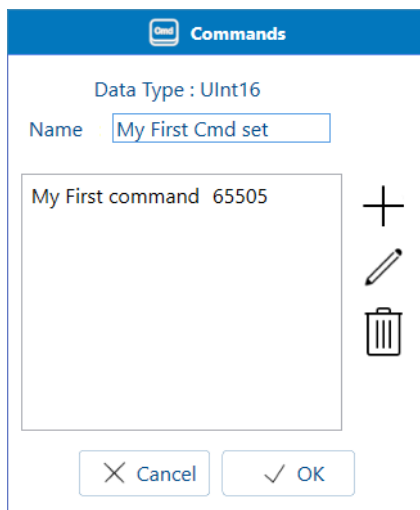
Bin

You can position the cursor at any spot and the number will be properly increased. The number is validated every time you add a value. If you exceed the limits, the value will not be inserted in the edit box and you will get an error dialog.

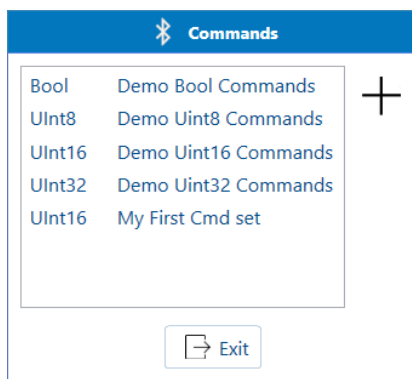


Click OK to close.

When your name and value are to your liking, click on OK on the Command edit window to save it to the set.



Add any other commands you want in the set. Nothing is saved to disk until you click on OK in the Commands dialog. When you do that it will show on the command sets list



When you exit the new set will be loaded in the commands pane only if you had a Characteristic of the same data type selected before you created the new set, like this example

Data Type	UInt16
Demo UInt16 Commands	
My First Cmd set	
My First command	
My Second Command	

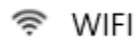
If you select the set, you will see the command names from the set below. Click on the command to see the value in the send box.

### Special Note

You will need a forked version of the ArduinoBLE library if you want it to work with Windows or Android. See link in sample

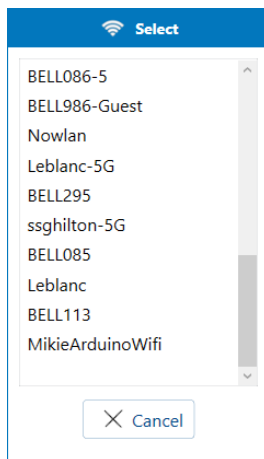
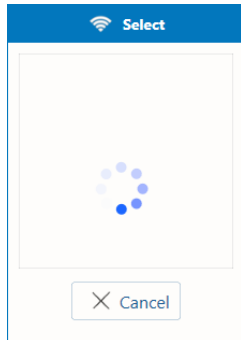
## WIFI Run Page

Click on WIFI on the menu



It will bring up the run page. General functionality of the run page is explained above.

Clicking on the Connect button will bring up the Select dialog for WIFI devices. This can take some time as it is scanning all the available WIFI connections.




Selecting a connection will close the dialog and set the selection on the run page.




## Info

Click on the info icon



 MikieArduinoWifi


SSID	: MikieArduinoWifi
Authentication Type	: RSNA PSK
Encryption Type	: CCMP
Up Time	: 3:22:05.938357
Signal Strength (Bars)	: 4
RSSI (Decible Milliwatts)	: -47
Physical Layer Kind	: HT High Throughput 802 11n
Mac Address (BSSID)	: 84:0d:8e:1e:d3:d5
Kind	: Infrastructure
WIFI Direct	: False
Channel Center Frequency (Khz)	: 2412000
Beacon Interval	: 0:00:00.1024

 Exit


## Settings

Click on the Settings icon



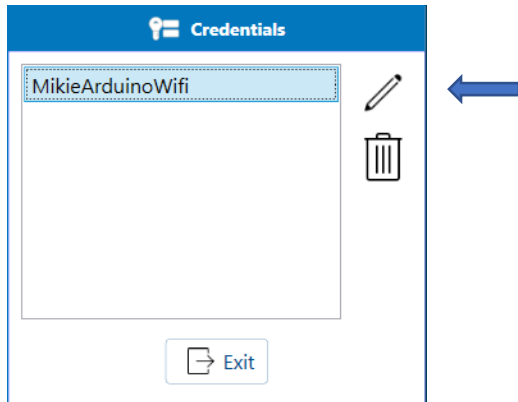
 Credentials

MikieArduinoWifi

 Exit

The Credentials dialog provides a way to edit or delete credentials and connection parameters for a previous WIFI connection. You can also add to the credentials list before connection. To edit or delete a credential highlight one of them





Click on the *Edit* icon to open the editor. The list has the SSID of the WIFI

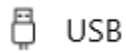
A screenshot of a software window titled "MikieArduinoWifi". It contains two main sections: "Network" and "Socket". Under "Network", there is a "Network Security Key" field filled with 10 dots. Under "Socket", there is a "Host Name/IP" field containing "192.168.4.1" and a "Network Service/Port" field containing "80". At the bottom of the window are two buttons: "Save" with a checkmark icon and "Cancel" with an 'X' icon.

The security key is the password for the WIFI access point (like a router). The Host name/IP or those that you have exposed on your embedded device. See the Arduino WIFI code sample in the appendix.

Click on Save to store your changes, or cancel to discard

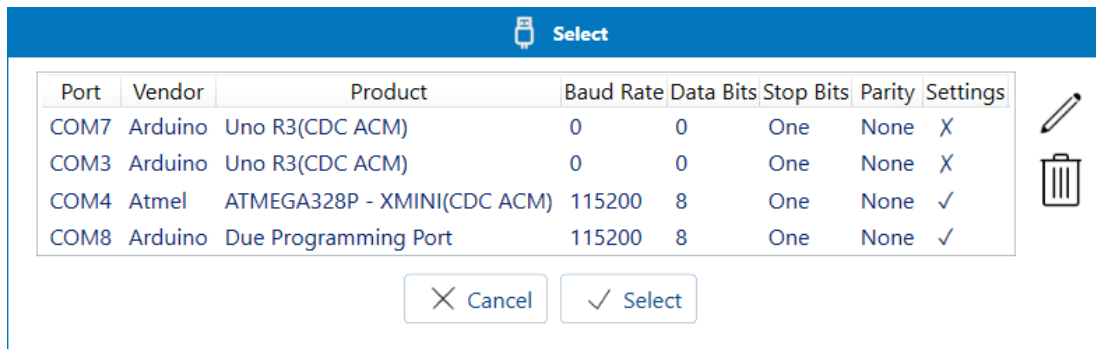
## USB Run Page

Click on the USB menu item



It will bring up the run page. General functionality of the run page is explained above.

Clicking on the Connect button will bring up the Select dialog for USB devices. If you previously created a configuration for the device with the same Name, Vendor and Product IDs then its values will be inserted. Currently I only have the names for Arduino and Atmel devices



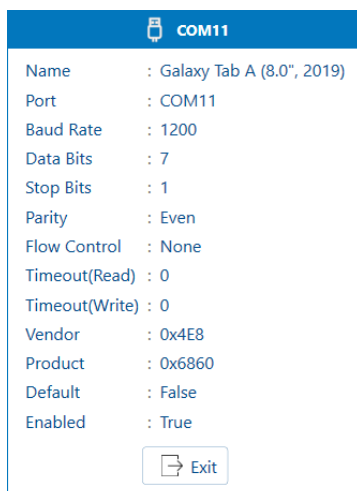
The Settings column is marked with a ✓ if you have an existing configuration. Otherwise, the values are those loaded from the OS which may not be what your device is expecting for connection

On successful connection, the device name will be posted on the title bar. NOTE: There is a bug at the OS level which sometimes gives the incorrect name for the device.



## Info


Click on the info icon





## Settings

Click on the Settings icon




 **Select**

Port	Vendor	Product	Baud Rate	Data Bits	Stop Bits	Parity	Settings
COM7	Arduino	Uno R3(CDC ACM)	0	0	One	None	X
COM3	Arduino	Uno R3(CDC ACM)	0	0	One	None	X
COM4	Atmel	ATMEGA328P - XMINI(CDC ACM)	115200	8	One	None	✓
COM8	Arduino	Due Programming Port	115200	8	One	None	✓

This is a list of the current devices. Highlight one and click on the edit icon.

 **COM9**

**Baud Rate**

**Data Bits**

**Stop Bits**

**Parity**

**Flow Control**

**Timeout(Read)**

**Timeout(Write)**

The configuration which comes up is either in the stored list of configurations or the default that the device presents.

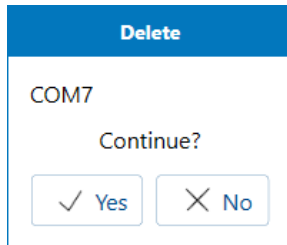
This will have the configuration information based on what was harvested from the serial connection and any changes you have stored in your configurations. Since there is no feedback if you have wrong settings, you may need to come back here to correct them. You must enter the settings that the embedded device is expecting. Look at the Arduino code sample for serial.

Change the settings and click OK to save. This is what will be used to connect. Note: The lookup is based on COM port and name, so if a different device of same name is connected to the same COM port you will have to change the settings to match those of the embedded device.

This will have the configuration information based on what was harvested from the serial connection and any changes you made on first connection. Since there is no feedback if you have wrong settings,

you may need to come back here to correct them. You must enter the settings that the embedded device is expecting. Look at the Arduino code sample for serial.

If you click on the delete button you are presented with a confirmation dialog.



Click on *Yes* to delete it, *No* to exit without changes.

Of special note:

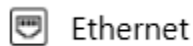
The Timeout values (milliseconds) should be set low so that the read and write will not stall before sending up accumulated bytes up the stack

Unless an error dialog comes up you should be ready to send and receive messages. You will not have to enter this information again

Note: The serial port does not return an error if your settings are not what is expected at the embedded device. In that case you will have to return to the Menu->USB Settings->Edit to change them.

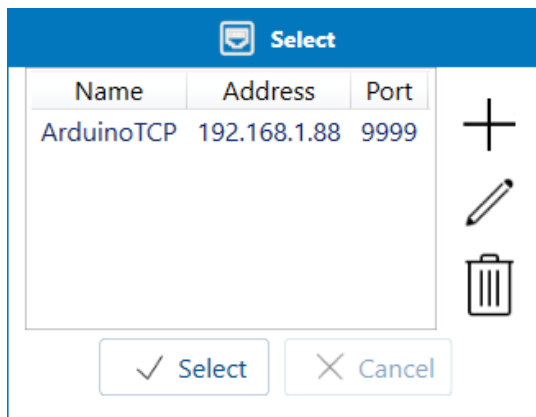
## Ethernet Run Page (TBD)

Click on the menu item



It will bring up the run page. General functionality of the run page is explained above.

Clicking on the Connect button will bring up the Select dialog for Ethernet connections. If there are none, you will be able to create a new one on first connection to a network. Ethernet does not provide the possibility of scanning for connections so you will need to know the IP address and port of your embedded device.

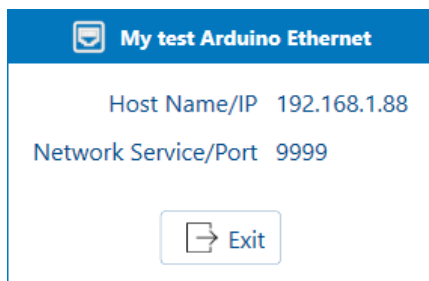


Selecting it will set the title bar to the connection



### Info

Click on the info icon



This will give you some basic information on the selected connection

## Settings

Click on the Settings icon



The 'Select' dialog box has a blue header with a shield icon and the word 'Select'. It contains a table with three columns: 'Name', 'Address', and 'Port'. The first row contains the text 'ArduinoTCP', '192.168.1.88', and '9999'. To the right of the table are three icons: a plus sign (+), a pencil (edit), and a trash can (delete). At the bottom of the dialog is an 'Exit' button with a right-pointing arrow icon.

Name	Address	Port
ArduinoTCP	192.168.1.88	9999

You can edit or delete any stored connection settings.

Click on the Edit icon to bring up the edit dialog

The 'Edit' dialog box has a blue header with a shield icon and the word 'Edit'. It contains three text input fields: 'Name' with the value 'My test Arduino Ethernet', 'Host Name/IP' with the value '192.168.1.88', and 'Network Service/Port' with the value '9999'. At the bottom are two buttons: 'Save' with a checkmark icon and 'Cancel' with an 'X' icon.

Modify entries and click on Save to store

Click on the Add icon to create a new Ethernet connection

The 'Create' dialog box has a blue header with a shield icon and the word 'Create'. It contains three text input fields: 'Name' with the value 'SampleName', 'Host Name/IP' with the value '192.168.1.100', and 'Network Service/Port' with the value '9999'. At the bottom are two buttons: 'Save' with a checkmark icon and 'Cancel' with an 'X' icon.

Enter the information and click on Save to store the information

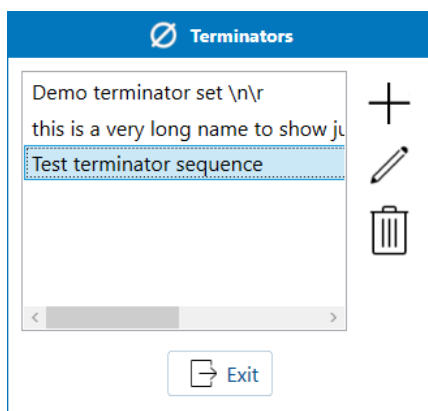
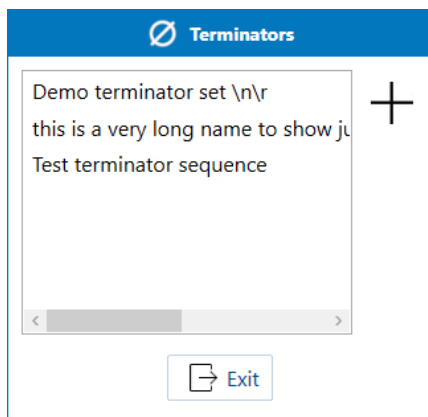
## Terminators

### Terminators

Clicking on this brings up a dialog box with currently stored terminator sequence.

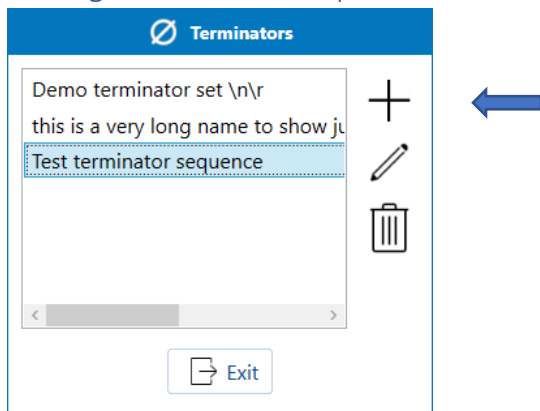
- A terminator sequence contains one or more nonprintable characters that are added to the end of the outgoing message.
- The embedded device must be expecting that sequence to determine when an incoming message is complete.
- The embedded device must also all those terminators to its outgoing messages since the App is expecting those to determine when the incoming message is complete

On Open, only the Add button is visible.

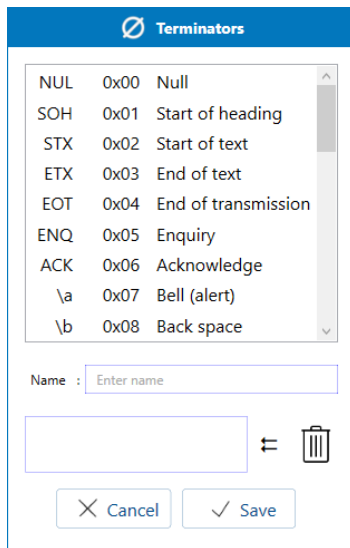


When you select a terminator set, the edit and delete button will appear.

## Adding a terminator sequence set



In the Terminator set dialog, click on the Add button and a Dialog appears to create a new set



Enter the name identify the set in the future. Click on any terminator in the list and it will be added to the sequence. You must do these in the sequence that is expected on the embedded device.



The result after you have clicked on form feed, then line feed.

Click on the delete icon to remove the last one in the sequence.



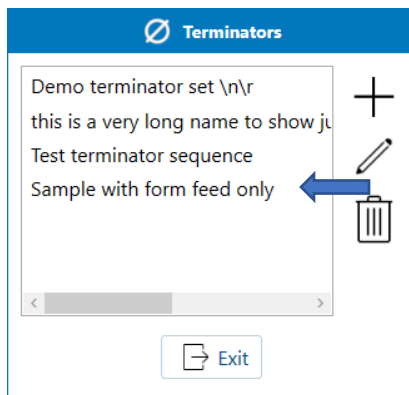
The result after clicking on the delete icon once

Make sure you enter a name then click on Save

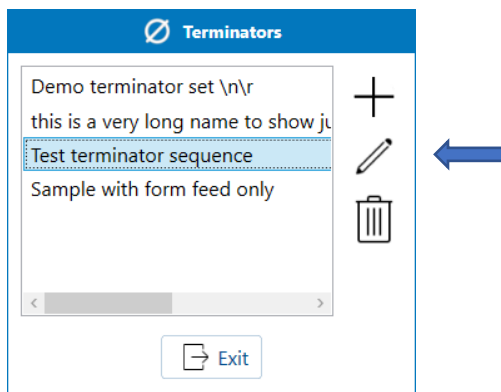


Name :

Your new terminator set is now stored



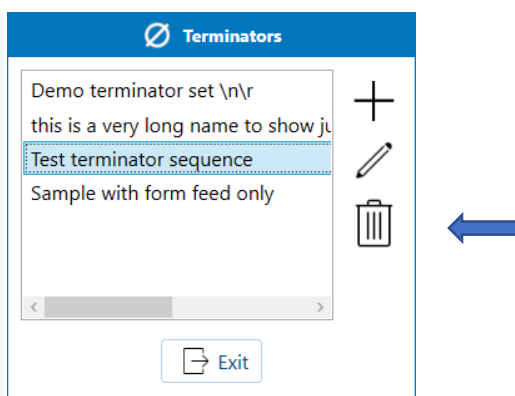
## Edit Terminator set



Select the set to edit and click on the *Edit* icon. Same process for adding a new one

## Deleting a Terminator set

At the Terminator set list dialog, select a set, and click on the *Delete* icon



You will be given the choice to delete or cancel

Delete

Test terminator sequence

Continue?

✓ Yes

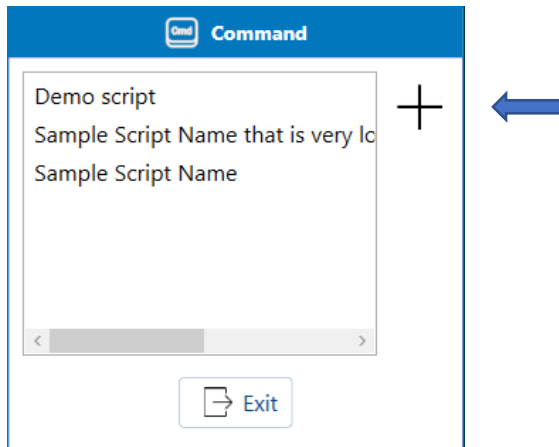
✗ No

## Commands



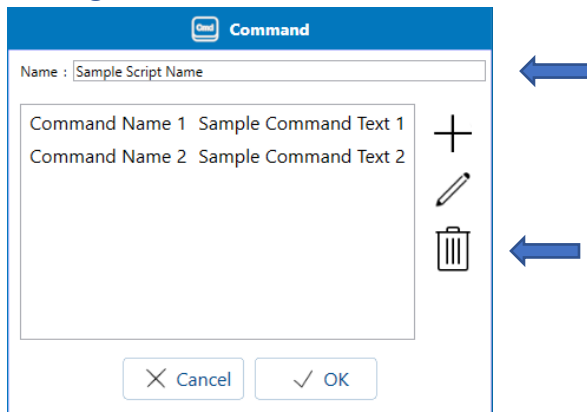
Click on the command menu item

The Commands list displays a set of existing command sets that are stored.

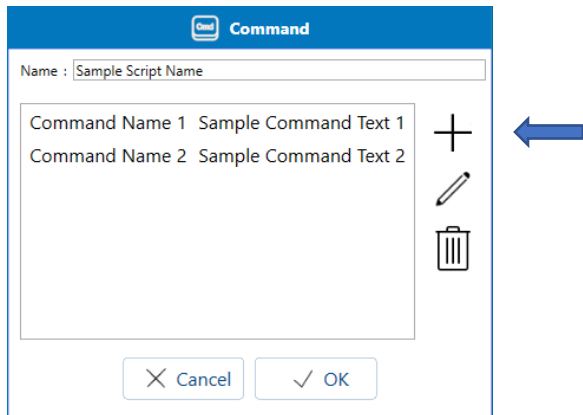


To add a new Command set, click on the Add button or select an existing to bring up more options.

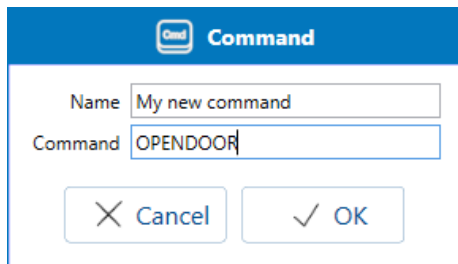
## Adding a command list



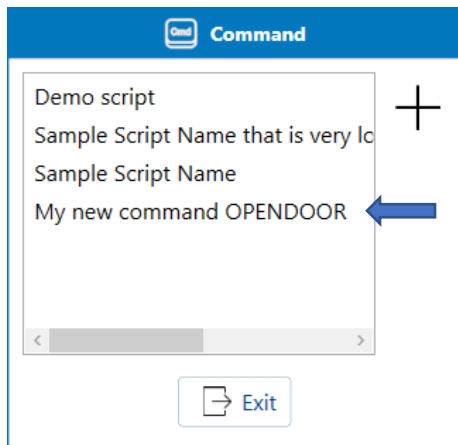
Edit the name. Edit the commands or select them and click on the Delete icon.



Click on the *Add* icon to create a new command



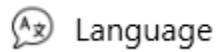
Enter the name that will be displayed for this command. It will show up on the list. The Command text (message) is what is sent to the embedded device. It must be exactly what the device is expecting. Terminators are added apart from this text. Select *Ok* to add the command



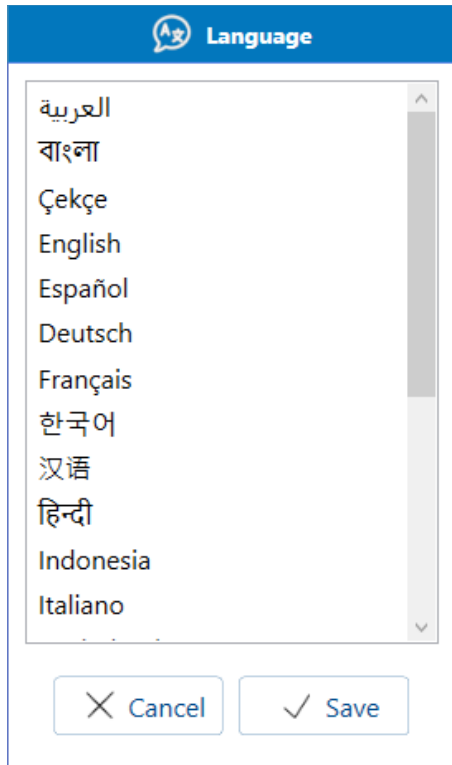
The new command is added to the command set

To delete a command, highlight it and click on the *Delete* icon

## Language



Clicking on this will bring up the Language selector. Currently there are 17 languages supported for UI elements



The language names are in the language they represent. They are sorted according to their phonetic sounding

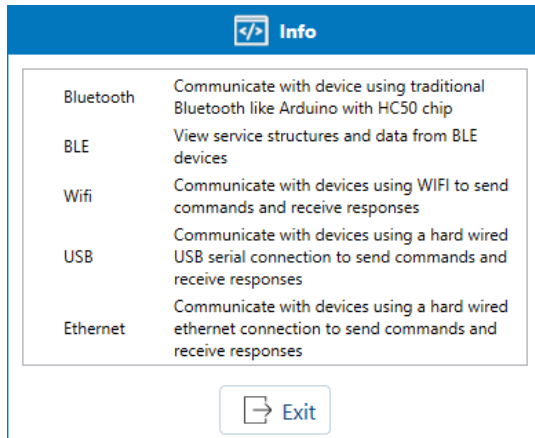
Highlight the requested language and you will see the language change on all open windows. Click Save to save that language as the current. It will be saved as your preference for future sessions. Click on Cancel to revert to previous language

## Code Samples

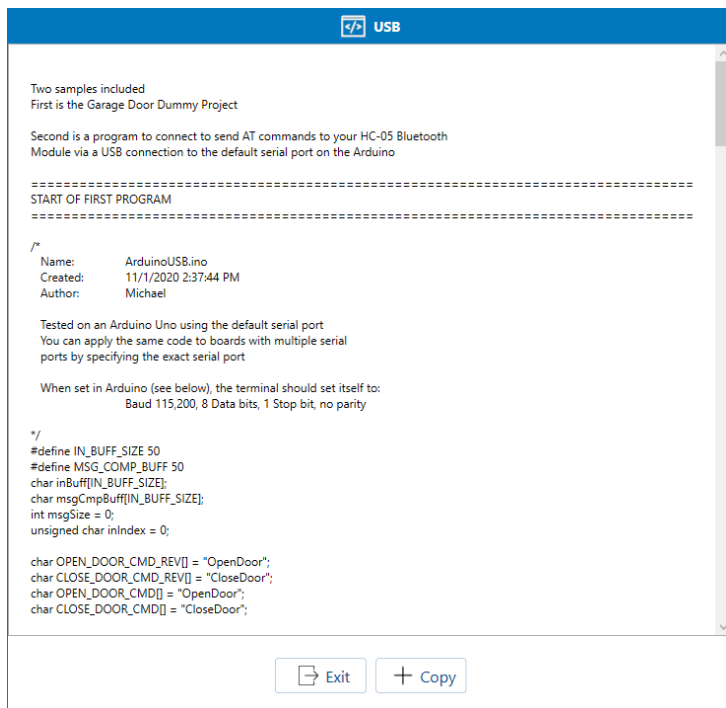
Select the menu item

### Code Samples

The code sample list will pop up



Select any of the samples in the list to open the code sample



Scroll to view the code. Click on the copy button to copy the contents to the clipboard

## Appendix

### Introduction to Embedded Messaging

Your embedded device such as Arduinos may run code that can run without any outside input. However in some cases you may want to send commands remotely to, say, turn on an IO that trips a solenoid that starts to open a garage door.

On simple systems, messages are sent as a string made up of ASCII printable characters (0x41 to 0x7E). That would include punctuations and upper and lower characters. Non printable characters are reserved to specify message termination indicators (terminators)

In this application I have a list from 0x00 (NULL) to 0x1F (unit separator (down arrow)) that you can select as your terminators

So, a command sent to the embedded device to open the garage door might look like this:

OpenDoor\n\r.

The “*OpenDoor*” is the message, the “\n\r” is the terminator sequence. The \n (0x0A) is a new line character and the \r (0x0D) is a carriage return. The backslash is how those are typically inserted in C or C# string.

The App allows you to create a terminator sequence which it automatically adds to the message. You select that sequence based on how you programmed your device. On the flip side, the App expects that same terminator sequence on all message sent back to it.

The messages and terminator sequences you create are stored separately so you can mix and match at run time.

### Arduino Bluetooth Sample

```
//-----  
// Name:          ArduinoBluetoothDataTests.ino  
// Created:   12/12/2019 3:14:05 PM  
// Author:    Michael  
//  
// Sample to send and receive date to and from Arduino Bluetooth shield  
//  
// Written and tested in Visual Studio using Visual Micro  
// Tested against Arduino Uno and itea Bluetooth shield with HC-05 module  
//  
// MUST HAVE BOARD TOGGLED TO DATA  
// MUST HAVE BOARD JUMPERS SET FOLLOWING for IO4 TX and IO5 RX  
//  °°°|°°  
//  °°|°°°  
//  °°°°°°  
//  
// MUST HAVE DEBUG SERIAL SET TO 9600 Baud  
// MUST HAVE BT SERIAL SET TO 38400 Baud  
//  
// Must pair with Bluetooth first. The Multi Comm Terminal provides that  
// functionality  
//-----
```

```

#include <SoftwareSerial.h>

#ifndef SECTION_DATA
int i = 0;
// The jumpers on BT board are set to 4TX and 5RX.
// They are reversed on serial since RX from BT gets TX to serial
SoftwareSerial btSerial(5, 4); //RX,TX
bool hasInput = false;
#define IN_BUFF_SIZE 100
char buff[IN_BUFF_SIZE];
unsigned char inIndex = 0;
#endif // !SECTION_DATA

// the setup function runs once when you press reset or power the board
void setup() {
    // There is some strange behaviour when using different baud rates
    SetupCommunications(9600, 38400);
}

// the loop function runs over and over again until power down or reset
void loop() {
    ListenToBTData();
}

// Private helpers
void SetupCommunications(long dbgBaud, long btBaud) {
    btSerial.begin(btBaud);
    Serial.begin(dbgBaud);

    while (!Serial) {
        ; // wait for serial port to connect. Needed for Native USB only
    }
    Serial.println("Debug serial active");
    // example had pin 9 set low, then high but does not seem necessary
}

void ListenToBTData() {
    if (btSerial.available() > 0) {
        if (inIndex >= IN_BUFF_SIZE) {
            inIndex = 0;
            Serial.println("Corrupt BT input. Purge buffer");
        }

        buff[inIndex] = (char)btSerial.read();
        if (buff[inIndex] == '\r') {
            Serial.write("CR");
        }
        else if (buff[inIndex] == '\n') {
            Serial.write("LN");
        }
        else {
            Serial.write(buff[inIndex]);
        }

        // Doing \n\r
        if (buff[inIndex] == '\r') {
            Serial.println("Printing msg in buffer");
            hasInput = true;
            Serial.write(buff, inIndex + 1);
            btSerial.write(buff, inIndex + 1);
        }
    }
}

```



```

        memset(buff, 0, IN_BUFF_SIZE);
        inIndex = 0;
    }
    else {
        inIndex++;
        // Wipe out buffer if too long
        if (inIndex >= IN_BUFF_SIZE) {
            inIndex = 0;
            Serial.println("Corrupt BT input. Purge buffer");
        }
        else {
        }
    }
}
else {
    if (hasInput) {
        hasInput = false;
    }

    if (i % 50 == 0) {
        Serial.print("No BT msg # ");
        Serial.print((i / 10));
        Serial.println("");
    }
    i++;
    delay(100);
}
}

```

## Arduino BLE Sample

```
/*
Name:      DemoArduinoBLE.ino
Created:   Jan 15, 2021

Sets up the BLE to have characteristics that change value in real time.
Battery Level // 0x180F Defined in BLE Spec
Temperature    // 0x2A6E
Humidity       // 0x2A6F
Pressure      // 0x2A6D
Custom        // My Uuid 0xF001 - has format descriptor for parsing - supports
Write
Bool on/Off switch // My Uuid 0xF002 - has format descriptor for parsing -
supports Write

Here is a partial list of BLE Format data types for the Format Descriptor. Some
are
not yet supported. Those will be displayed as a series of bytes
Boolean = 1
UInt_2bit = 2
UInt_4bit = 3
UInt_8bit = 4
UInt_12bit = 5
UInt_16bit = 6
UInt_24bit = 7
UInt_32bit = 8
UInt_48bit = 9
UInt_64bit = 10
UInt_128bit = 11 ** Not supported **
Int_8bit = 12
Int_12bit = 13
Int_16bit = 14
Int_24bit = 15
Int_32bit = 16
Int_48bit = 17
Int_64bit = 18
Int_128bit = 19 ** Not supported **
IEEE_754_32bit_floating_point = 20
IEEE_754_64bit_floating_point = 21
IEEE_11073_16bit_SFLOAT = 22 ** Not supported **
IEEE_11073_32bit_FLOAT = 23 ** Not supported **
IEEE_20601_format = 24 ** Not supported **
UTF8_String = 25
UTF16_String = 26
OpaqueStructure = 27 ** Not supported **
```

For this to work you need the forked version of ArduinoBLE  
which supports connection to Windows and Android  
<https://github.com/unknownconstant/ArduinoBLE>

Tested on Arduino Uno WIFI Rev 2

Arduino code partialy based on:

<https://programmaticponderings.com/2020/08/04/getting-started-with-bluetooth-low-energy-ble-and-generic-attribute-profile-gatt-specification-for-iot/>

With some modifications.

Values are generated internally for demo instead of reading I/O

```
*/
#include <ArduinoBLE.h>
#include <string.h>

//-----
// Variables
const int GENERIC_ACCESS_APPEARANCE_ID = 1792; // BLE Spec 0x1792

// Create battery service
BLEService batteryService("180F");
BLETypedCharacteristic<uint8_t> batteryLevelCharacteristic("2A19", BLERead | BLENotify);

// Environment sensing service
BLEService environmentService("181A");
BLETypedCharacteristic<uint32_t> pressureCharacteristic("2A6D", BLERead | BLENotify);
BLETypedCharacteristic<int16_t> tempCharacteristic("2A6E", BLERead | BLENotify);
BLETypedCharacteristic<uint16_t> humidCharacteristic("2A6F", BLERead | BLENotify);

// Custom 2 byte sensor service
BLEService ioService("FF01");
BLETypedCharacteristic<uint16_t> analogSensorCharacteristic("F001", BLERead | BLENotify |
BLEWrite);
BLEDescriptor sensorDescriptorUserDescription("2901", "An Sensor 1"); // Defined in Spec

uint8_t format[7] = {
    0x6,           // Data type. BLE spec 6 = uint16_t. Defines size of data in
characteristic read
    (byte)-2,      // Exponent -2 sbyte. stored as byte. parsed out as sbyte to
maintain sign
    0x0, 0x0,      // Measurement uint16_t. 0 is unit-less. Adds abbreviation on
display string
    0x0,           // Namespace byte. 0 is undefined, 1 is BLE Sig (only one defined in
spec)
    0x0, 0x0       // Description uint
};
// BLE spec 0x2904 is the Format Descriptor
BLEDescriptor sensorDescriptorFormat("2904", format, sizeof(format));

// Custom bool onOff sensor
BLEDescriptor onOffDescriptorUserDescription("2901", "On Off Switch"); // Defined in Spec
BLEBooleanCharacteristic onOffCharacteristic("F002", BLERead | BLENotify | BLEWrite);
uint8_t onOffData[] { 0x01, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 };
BLEDescriptor onOffDataDesc("2904", onOffData, sizeof(onOffData));

// Track dummy values
uint8_t batteryLevel = 93;
int16_t tempLevel = 3392;
uint16_t humidityLevel = 4400;
uint32_t pressureLevel = 28500;
uint16_t analogLevel = 16025;
byte isOn = 1; // On by default
```

```

unsigned long lasMsTimeUpdate = 0;
bool upIncrement = true;

//-----
// Main Arduino functions
// the setup function runs once when you press reset or power the board
void setup() {
    Serial.begin(9600);
    while (!Serial) {}
    Serial.println("started...");
    SetupBLE();
}

// the loop function runs over and over again forever
void loop() {
    BLEDevice central = BLE.central();
    if (central) {
        while (central.connected()) {
            WriteBatteryLevelOnMsInterval(2000);
            delayMicroseconds(10);
        }
    }
}

//-----
// Other functions
void SetupBLE() {
    if (!BLE.begin()) {
        Serial.println("FAIL");
        while (1);
    }
    // Device
    BLE.setLocalName("Mikie BLE"); // visible in search
    BLE.setDeviceName("MR BLE Device");
    BLE.setAppearance(GENERIC_ACCESS_APPEARANCE_ID);
    BLE.setEventHandler(BLEConnected, bleOnConnectHandler);
    BLE.setEventHandler(BLEDisconnected, bleOnDisconnectHandler);

    // Battery service
    batteryService.addCharacteristic(batteryLevelCharacteristic);
    BLE.addService(batteryService);
    BLE.setAdvertisedService(batteryService);

    // Environment sensing service with 3 characteristics
    environmentService.addCharacteristic(tempCharacteristic);
    environmentService.addCharacteristic(humidCharacteristic);
    environmentService.addCharacteristic(pressureCharacteristic);
    BLE.addService(environmentService);
    BLE.setAdvertisedService(environmentService);

    // My custom sensor service
    onOffCharacteristic.addDescriptor(onOffDataDesc);
    onOffCharacteristic.addDescriptor(onOffDescriptorUserDescription);
    onOffCharacteristic.setEventHandler(BLEWritten, onOffHandler);
    ioService.addCharacteristic(onOffCharacteristic);

    analogSensorCharacteristic.addDescriptor(sensorDescriptorUserDescription);

```

```

    analogSensorCharacteristic.addDescriptor(sensorDescriptorFormat);
    analogSensorCharacteristic.setEventHandler(BLEWritten, uint16CustomWriteHandler);
    ioService.addCharacteristic(analogSensorCharacteristic);

    BLE.addService(ioService);
    BLE.setAdvertisedService(ioService);
    BLE.advertise();

    WriteValues();
    Serial.println("BLE started");
}

```

```

void WriteBatteryLevelOnMsInterval(unsigned long msInterval) {
    // Simulate reading of actual hardware sensors for demo
    unsigned long currentMs = millis();
    if ((currentMs - lasMsTimeUpdate) > msInterval) {
        lasMsTimeUpdate = currentMs;
        if (isOn) {
            if (upIncrement) {
                tempLevel += 111;
                humidityLevl += 212;
                pressureLevel += 18;
                analogLevel += 67;
                batteryLevel++;
                if (batteryLevel > 99) {
                    batteryLevel = 99;
                    upIncrement = false;
                    batteryLevel--;
                }
            }
            else {
                tempLevel -= 111;
                humidityLevl -= 212;
                pressureLevel -= 18;
                analogLevel -= 67;
                batteryLevel--;
                if (batteryLevel < 85) {
                    upIncrement = true;
                    batteryLevel = 60;
                }
            }
        }
        WriteValues();
    }
}

```

```

void ResetValues() {
    batteryLevel = 93;
    tempLevel = 3392;
    humidityLevl = 4400;
    pressureLevel = 285;
    analogLevel = 16025;
}

```

```

void WriteValues() {

```

```

        batteryLevelCharacteristic.writeValue(batteryLevel);
        tempCharacteristic.writeValue(tempLevel);
        humidCharacteristic.writeValue(humidityLevel);
        pressureCharacteristic.writeValue(pressureLevel);
        analogSensorCharacteristic.writeValue(analogLevel);

        Serial.print("    Battery:"); Serial.println(batteryLevel);
        Serial.print("Temperature:"); Serial.println(tempLevel);
        Serial.print("    Pressure:"); Serial.println(pressureLevel);
        Serial.print("    Humidity:"); Serial.println(humidityLevel);
        Serial.print("    An sensor:"); Serial.println(analogLevel);
    }

//-----
// Event handlers
void bleOnConnectHandler(BLEDevice central) {
    Serial.println("CONNECTED");
}

void bleOnDisconnectHandler(BLEDevice central) {
    Serial.println("DISCONNECTED");
    ResetValues();
}

void onOffHandler(BLEDevice central, BLECharacteristic characteristic) {
    isOn = characteristic.value()[0];
    Serial.print("OnOff:"); Serial.println(isOn);
}

void uint16CustomWriteHandler(BLEDevice central, BLECharacteristic characteristic) {
    memcpy(&analogLevel, characteristic.value(), sizeof(analogLevel));
    Serial.print("analogLevel:"); Serial.println(analogLevel);
}

```

## Arduino WIFI Sample

```
// -----
// Name:          TestArduinoWifi.ino
// Created:   10/16/2020 1:22:40 PM
// Author:    Michael
//
// Tested on the Arduino UNO WIFI Rev2
//
// Sets up the board as a WIFI access point with a socket
//
// Initial example code found in:
// https://www.arduino.cc/en/Reference/WiFiNINABeginAP
//
// -----
#include <SPI.h>
#include <WiFiNINA.h>
#include "wifi_defines.h"
// -----
// The wifi_defines.h has the strings for the SSID and password
// Here are the contents
//      #pragma once
//
//      Must be 8 or more characters
//      #define MY_SSID "MikieArduinoWifi"
//
//      Must be 8 or more characters
//      #define MY_PASS "1234567890"
//
// -----

char ssid[] = MY_SSID;
char pwd[] = MY_PASS;
int keyIndex = 0;
int status = WL_IDLE_STATUS;
int led = LED_BUILTIN;

// Port 80 of socket usually used for HTTP - just using it as a entry
// The Multi Comm Terminal must enter whatever port number is set in
// the Arduino
WiFiServer server(80);

void setup() {
    // Serial is just to push data for debugging the Arduino code. Can be removed
    Serial.begin(57600);
    while(!Serial){}
    Serial.println("Serial started");

    pinMode(led, OUTPUT);

    // Check for the WiFi module:
    if (WiFi.status() == WL_NO_MODULE) {
        Serial.println("Communication with WiFi module failed!");
        // don't continue
        while (true);
    }

    String fv = WiFi.firmwareVersion();
    Serial.print("WIFI firmware version ");
    Serial.println(fv);
}
```

```

if (fv < WIFI_FIRMWARE_LATEST_VERSION) {
    Serial.print("Version below ");
    Serial.println(WIFI_FIRMWARE_LATEST_VERSION);
    Serial.println("Please upgrade the firmware");
}

// by default the local IP address of will be 192.168.4.1
// you can override it with the following:
// Whatever you choose will be the IP that you enter in
// the Multi Comm Connection parameters
// WiFi.config(IPAddress(10, 0, 0, 1));

// Print the network name (SSID);
Serial.print("Creating access point named: ");
Serial.println(ssid);

// Create access point
status = WiFi.beginAP(ssid, pwd);
if (status != WL_AP_LISTENING) {
    Serial.print("Status "); Serial.println(status);
    Serial.println("Access point creation failed");
    while (true) { }
}

delay(10000);

server.begin();
// you're connected now, so print out the status to the serial debug:
printWifiStatus();
}

// the loop function runs over and over again until power down or reset
void loop() {
    // Print a message to debug if a device has connected or disconnected
    if (status != WiFi.status()) {
        status = WiFi.status();
        if (status == WL_AP_CONNECTED) {
            Serial.println("Device connected to AP");
        }
        else {
            // Device has disconnected from AP. Back in listening mode
            Serial.println("Device disconnected from AP");
        }
    }
    ListenForClient();
}

// Determines if a client has connected a socket and sent a message
void ListenForClient() {
    //https://www.arduino.cc/en/Reference/WiFiNINABeginAP
    WiFiClient client = server.available();
    if (client) {
        Serial.println("Got a client connected new client");
        String currentLine = "";

        // Loop while the client's connected
        while (client.connected()) {
            if (client.available()) {
                // Read a byte
                char c = client.read();
                // Print character serial for debug

```



```

        Serial.write(c);

        // This will bounce each character through the socket
        // The MultiCommMonitor will pick up the terminators and
        // Display it as a return message
        //
        // In the real world, you would accumulate the bytes until
        // the expected terminator sequence is detected.
        // You would then
        // - Look at the message
        // - Determine operation requested
        // - Do the operation
        // - Optionally, send back a response with expected
terminators
        //
        // See samples for BT Classic and BLE
        client.print(c);
    }

    // close the connection:
    client.stop();
    // Send a debug message
    Serial.println("client disconnected");
}
}

```

```

void printWifiStatus() {
    // print the SSID of the network you're attached to:
    Serial.print("SSID: ");
    Serial.println(WiFi.SSID());

    // print your board's socket IP address:
    IPAddress ip = WiFi.localIP();
    Serial.print("IP Address: ");
    Serial.println(ip);

    // print the received signal strength:
    long rssi = WiFi.RSSI();
    Serial.print("signal strength (RSSI):");
    Serial.print(rssi);
    Serial.println(" dBm");
}

```

## Arduino USB Sample

```
/*
  Name:           ArduinoUSB.ino
  Created:    11/1/2020 2:37:44 PM
  Author:    Michael

  Tested on an Arduino Uno using the default serial port
  You can apply the same code to boards with multiple serial
  ports by specifying the exact serial port

  When set in Arduino (see below), the terminal should set itself to:
      Baud 115,200, 8 Data bits, 1 Stop bit, no parity

  */
#define IN_BUFF_SIZE 50
#define MSG_COMP_BUFF 50
char inBuff[IN_BUFF_SIZE];
char msgCmpBuff[IN_BUFF_SIZE];
int msgSize = 0;
unsigned char inIndex = 0;

char OPEN_DOOR_CMD[] = "OpenDoor";
char CLOSE_DOOR_CMD[] = "CloseDoor";
int OPEN_CMD_LEN = 8;
int CLOSE_CMD_LEN = 9;

// the setup function runs once when you press reset or power the board
void setup() {
    // 115,200 Baud, 8 Data bits, no parity, 1 stop bit
    // The terminal should find those same parameters. If you do not set
    // it here, the values are not detected in the terminal
    Serial.begin(115200, SERIAL_8N1);
    while (!Serial) {
    }
}

// the loop function runs over and over again until power down or reset
void loop() {
    ListenForData();
}

void ListenForData() {
    int available = Serial.available();
    if (available > 0) {
        msgSize = 0;

        // Error check to avoid overrun of buffer
        if ((inIndex + available) > IN_BUFF_SIZE) {
            Serial.write("ERROR-PURGING INPUT\r\n");
            inIndex = 0;
            return;
        }

        size_t count = Serial.readBytes(&inBuff[inIndex], available);
        inIndex += count;

        for (int i = 0; i < inIndex; i++) {
```

```

        // Make assumption that \n\r coming in so look for \r for end
        if (i > 1) {
            if (inBuff[i-1] == '\r' && inBuff[i] == '\n') {
                msgSize = i - 1;
                memset(msgCmpBuff, 0, MSG_COMP_BUFF);
                memcpy(msgCmpBuff, inBuff, msgSize);
                memmove(inBuff, &inBuff[i + 1], (inIndex + count) -
(msgSize + 2));

                inIndex -= msgSize + 2;
                memset(&inBuff[inIndex], 0, (IN_BUFF_SIZE -
inIndex));

                CompareForResponse(msgSize);
            }
        }
    }
}

/// <summary>Compare the incoming message to carry out IO actions</summary>
/// <param name="msgSize">Size of the incoming message</param>
void CompareForResponse(int msgSize) {

    // Compare from start of buffer. Garbage at end of Command
    // and before terminator is ignored (OpenDoorlsdlfkjdflj)
    if (strncmp(msgCmpBuff, OPEN_DOOR_CMD, OPEN_CMD_LEN) == 0) {
        Serial.write("OPENING\r\n");
        OpenGarageDoor();
    }
    else if (strncmp(msgCmpBuff, CLOSE_DOOR_CMD, CLOSE_CMD_LEN) == 0) {
        Serial.write("CLOSING\r\n");
        CloseGarageDoor();
    }
    else {
        Serial.write("NOT_HANDLED\r\n");
    }
}

void OpenGarageDoor() {
    // Do you IO stuff here to open the door
}

void CloseGarageDoor() {
    // Do you IO stuff here to close the door
}

```

## Arduino USB Sample 2 for AT Commands

```
/*
Name:          ArduinoSerial_BT_AT_Cmds.ino
Created:       11/11/2020 1:08:41 PM
Author:        Michael

To enable the Multi Comm Terminal to send AT commands to the
HC-05 module on an ITEA board

Module must be set to CMD via switch.
If the module is software switched I will need to add
that in a command

Written and tested in Visual Studio using Visual Micro
Tested against Arduino Uno and ITEA Bluetooth shield with HC-05 module

MUST HAVE BOARD TOGGLED TO CMD
MUST HAVE BOARD JUMPERS SET FOLLOWING for IO4 TX and IO5 RX
ooo|oo
oo|ooo
ooooo

MUST HAVE DEBUG SERIAL SET TO 9600 Baud
MUST HAVE BT SERIAL SET TO 38400 Baud

Command          Response          Details
AT               +NAME:itead
AT+VERSION       +VERSION:hc01.comV2.1
AT+ADDR         +ADDR:2016:4:76101
AT+UART         +UART:9600,0,0          Baud, data bits, stop bits
AT+NAME=<param>  OK              Sets the module name
AT+NAME?        +NAME:<Param>    Get the module name
AT+ORGL         OK              Restore to defaults
AT+RESET        OK              Reset
AT+ROLE         +ROLE:0        (1=master, 0=client)
AT+PSWD         +PSWD:1234      (pairing PIN)

*/
#include <SoftwareSerial.h>

#define IN_BUFF_SIZE 100
char buff[IN_BUFF_SIZE];

// The jumpers on BT board are set to 4TX and 5RX.
// They are reversed on serial since RX from BT gets TX to serial
SoftwareSerial btSerial(5, 4); //RX,TX

// the setup function runs once when you press reset or power the board
void setup() {
    //Need 38400 for BT and 9600 for serial to send AT commands
    //strange because BT AT cmd shows BT baud is 9600.
    SetupCommunications(9600, 38400);
}

// the loop function runs over and over again until power down or reset
```

```

void loop() {
    int count = Serial.available();
    if (count > 0) {
        Serial.readBytes(buff, count);
        btSerial.write(buff, count);
    }

    count = btSerial.available();
    if (count > 0) {
        btSerial.readBytes(buff, count);
        Serial.write(buff, count);
    }
}

void SetupCommunications(long dbgBaud, long btBaud) {
    Serial.begin(dbgBaud);
    btSerial.begin(btBaud);
    while (!Serial) {
        ; // wait for default serial port to connect
    }
}

```

## Arduino BLE Sample

```
/*
Name:    DemoArduinoBLE.ino
Created: Jan 15, 2021

Sets up the BLE to have characteristics that change value in real time.
    Battery Level // 0x180F Defined in BLE Spec
    Temperature    // 0x2A6E
    Humidity        // 0x2A6F
    Pressure        // 0x2A6D
    Custom          // My Uuid 0xF001 - has format descriptor for parsing

    a list of Format data types are found in user manual

For this to work you need the forked version of ArduinoBLE
which supports connection to Windows and Android
https://github.com/unknownconstant/ArduinoBLE/tree/pairing

Tested on Arduino Uno WIFI Rev 2

Arduino code partialy based on:
https://programmaticponderings.com/2020/08/04/getting-started-with-bluetooth-low-energy-ble-and-generic-attribute-profile-gatt-specification-for-iot/

With some modifications.
Values are generated internally for demo instead or reading I/O

*/
#include <ArduinoBLE.h>

//-----
// Variables
const int GENERIC_ACCESS_APPEARANCE_ID = 1792; // BLE Spec 0x1792

// Create battery service
BLEService batteryService("180F");
BLETypedCharacteristic<uint8_t> batteryLevelCharacteristic("2A19", BLERead | BLENotify);

// Environment sensing service
BLEService environmentService("181A");
BLETypedCharacteristic<uint32_t> pressureCharacteristic("2A6D", BLERead | BLENotify);
BLETypedCharacteristic<int16_t> tempCharacteristic("2A6E", BLERead | BLENotify);
BLETypedCharacteristic<uint16_t> humidCharacteristic("2A6F", BLERead | BLENotify);

// Custom 2 byte sensor service
BLEService ioService("FF01");
BLETypedCharacteristic<uint16_t> analogSensorCharacteristic("F001", BLERead | BLENotify);
BLEDescriptor sensorDescriptorUserDescription("2901", "aSensor1"); // Defined in Spec
uint8_t format[7] = {
    0x6,           // Data type. BLE spec 6 = uint16_t. Defines size of data in
characteristic read
    (byte)-2,      // Exponent -2 sbyte. stored as byte. parsed out as sbyte to
maintain sign
    0x0, 0x0,      // Measurement uint16_t. 0 is unit-less. Adds abbreviation on
display string
}
```

```

    0x0,          // Namespace byte. 0 is undefined, 1 is BLE Sig (only one defined in
spec)
    0x0, 0x0      // Description uint
};
// BLE spec 0x2904 is the Format Descriptor
BLEDescriptor senforDescriptorFormat("2904", format, sizeof(format));

// Track dummy values
uint8_t batteryLevel = 93;
int16_t tempLevel = 3392;
uint16_t humidityLevl = 4400;
uint32_t pressureLevel = 28500;
uint16_t analogLevel = 16025;

unsigned long lasMsTimeUpdate = 0;
bool upIncrement = true;
//-----

//-----
// Main Arduino functions

// Executed once on power on or reset
void setup() {
    Serial.begin(9600);
    while (!Serial) {}
    Serial.println("started...");
    SetupBLE();
}

// the loop function runs over and over again forever
void loop() {
    BLEDevice central = BLE.central();
    if (central) {
        while (central.connected()) {
            WriteBatteryLevelOnMsInterval(5000);
            delayMicroseconds(10);
        }
    }
}
//-----

//-----
// Helper functions
void SetupBLE() {
    if (!BLE.begin()) {
        Serial.println("FAIL");
        while (1);
    }
    // Device
    BLE.setLocalName("Mikie BLE"); // visible in search
    BLE.setDeviceName("MR BLE Device");
    BLE.setAppearance(GENERIC_ACCESS_APPEARANCE_ID);
    BLE.setEventHandler(BLEConnected, bleOnConnectHandler);
    BLE.setEventHandler(BLEDisconnected, bleOnDisconnectHandler);

    // Battery service

```

```

batteryService.addCharacteristic(batteryLevelCharacteristic);
BLE.addService(batteryService);
BLE.setAdvertisedService(batteryService);

// Environment sensing service with 3 characteristics
environmentService.addCharacteristic(tempCharacteristic);
environmentService.addCharacteristic(humidCharacteristic);
environmentService.addCharacteristic(pressureCharacteristic);
BLE.addService(environmentService);
BLE.setAdvertisedService(environmentService);

// My custom sensor service
analogSensorCharacteristic.addDescriptor(sensorDescriptorUserDescription);
analogSensorCharacteristic.addDescriptor(sensorDescriptorFormat);
ioService.addCharacteristic(analogSensorCharacteristic);
BLE.addService(ioService);
BLE.setAdvertisedService(ioService);

BLE.advertise();

WriteValues();
Serial.println("BLE started");
}

void WriteBatteryLevelOnMsInterval(unsigned long msInterval) {
    // Simulate reading of actual hardware sensors for demo
    unsigned long currentMs = millis();
    if ((currentMs - lastMsTimeUpdate) > msInterval) {
        lastMsTimeUpdate = currentMs;
        if (upIncrement) {
            tempLevel += 111;
            humidityLevel += 212;
            pressureLevel += 18;
            analogLevel += 67;
            batteryLevel++;
            if (batteryLevel > 99) {
                batteryLevel = 99;
                upIncrement = false;
                batteryLevel--;
            }
        }
        else {
            tempLevel -= 111;
            humidityLevel -= 212;
            pressureLevel -= 18;
            analogLevel -= 67;
            batteryLevel--;
            if (batteryLevel < 85) {
                upIncrement = true;
                batteryLevel = 60;
            }
        }
        WriteValues();
    }
}

```



```

void ResetValues() {
    batteryLevel = 93;
    tempLevel = 3392;
    humidityLevel = 4400;
    pressureLevel = 285;
    analogLevel = 16025;
}

void WriteValues() {
    batteryLevelCharacteristic.writeValue(batteryLevel);
    tempCharacteristic.writeValue(tempLevel);
    humidCharacteristic.writeValue(humidityLevel);
    pressureCharacteristic.writeValue(pressureLevel);
    analogSensorCharacteristic.writeValue(analogLevel);

    Serial.print("    Battery:"); Serial.println(batteryLevel);
    Serial.print("Temperature:"); Serial.println(tempLevel);
    Serial.print("    Pressure:"); Serial.println(pressureLevel);
    Serial.print("    Humidity:"); Serial.println(humidityLevel);
    Serial.print("    An sensor:"); Serial.println(analogLevel);
}

//-----

//-----
// Event handlers

// When a connection is established to device
void bleOnConnectHandler(BLEDevice central) {
    Serial.println("CONNECTED");
}

// When a connection to device is broken
void bleOnDisconnectHandler(BLEDevice central) {
    Serial.println("DISCONNECTED");
    ResetValues();
}

//-----

```