

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.....	5
WIFI Credentials.....	6
Languages	6
Settings.....	6
User Interface	6
Buttons.....	6
Wordless Buttons.....	7
Run Window.....	7
Elements	8
Main Window.....	9
Elements:	9
Menu.....	10
Bluetooth Run Page	10
Info	11
Settings.....	12
WIFI Run Page	13
Info	13
Settings.....	14
USB Run Page	16
Info	16
Settings.....	17
Ethernet Run Page (TBD)	19
Info	19
Settings.....	20
BLE Run Page (Bluetooth Low Energy) (TBD).....	21
Info	21
Special Note	23

Terminators.....	24
Adding a terminator sequence set.....	25
Edit Terminator set	26
Deleting a Terminator set	26
Commands	28
Adding a command list	28
Language	30
Code Samples.....	31
Appendix	32
Introduction to Embedded Messaging	32
Arduino Bluetooth Sample.....	32
Arduino BLE Sample	35
Arduino WIFI Sample	39
Arduino USB Sample	42
Arduino USB Sample 2 for AT Commands	44

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
- 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
- BLE (Bluetooth Low Power):
 - Connects to a device which implements BLE Characteristics for inputs and outputs.
 - As such it emulates a serial input output connection
 - Tested against an Arduino WIFI Rev2.
 - This was initially working with Arduino:
 - A Windows update changed connection requirement to require pairing.
 - Since Arduino does not support BLE pairing I removed connection functionality.
 - You can still get detailed info from devices that support the pairing requirement
 - When I can find an embedded device that supports the requirement for testing, I will enable connection.
 - An Arduino code sample is provided.

Data Storage

Data storage root depends if running as a Windows Application or a Win App Desktop Application (From the App Store)

- 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

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 11 languages in most areas of the UI:

- Chinese (Simplified)
- English
- French
- German
- Hindi
- Italian
- Japanese
- Korean
- Portuguese
- Russian
- Spanish

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 since it mostly avoids idioms.

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 will 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



Add: Add a new Item



Edit: Edit and existing item



Delete: Remove an item

Run Window

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

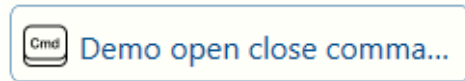
You can have one window of each communication medium open at the same time. The exception is Bluetooth and BLE. Since they both use the same radio, only one of the two can be open. If you try to open the second Bluetooth type, the first opened type will close.

The screenshot shows a software interface for Bluetooth communication. At the top, a blue header bar contains a Bluetooth icon and the text "Bluetooth". Below the header, there are two buttons: "Cmd Demo open close comma..." and "Response". The main area is divided into two large text boxes. The left box contains the text "Open door cmd" and "Close door cmd". The right box is empty. Below these boxes is a "Send" button with a paper plane icon and a text input field. At the bottom, there is a section with a label "this is a very long name to sh..." and a table of hex values: 0x04, 0x02, and two empty cells. Below the table are labels "EOT" and "STX", and two empty cells. To the right of this section are several buttons: "Discover" (with a magnifying glass icon), "Connect" (with a lightning bolt icon), "Disconnect" (with an 'X' icon), "Log" (with a list icon), and "Exit" (with a door icon). There are also icons for help (question mark) and settings (gear).

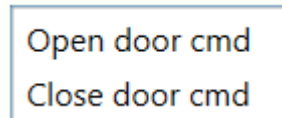
Elements



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



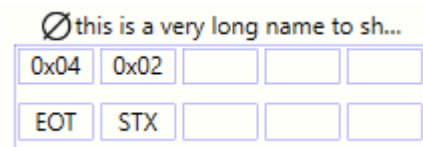
Commands from the selected commands list. When you tap one, its command is posted in the Send box



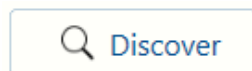
This will send the command in the Send box to the device if you have established a connection. You can also free edit the contents of the Send box and send that text



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



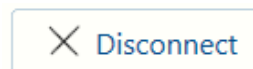
The shows the currently selected Terminators set. Tapping on this will bring up a select dialog to change the set



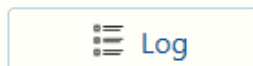
Launches a discovery of the available connections. The method of discovery varies per medium.



Establish a connection for the selected device. If you have not previously selected a device, you will be presented with a select dialog. The light to the left turns green on successful connection



Break any current connection



Opens a pane at the bottom to show scrolling debug information



Close the page

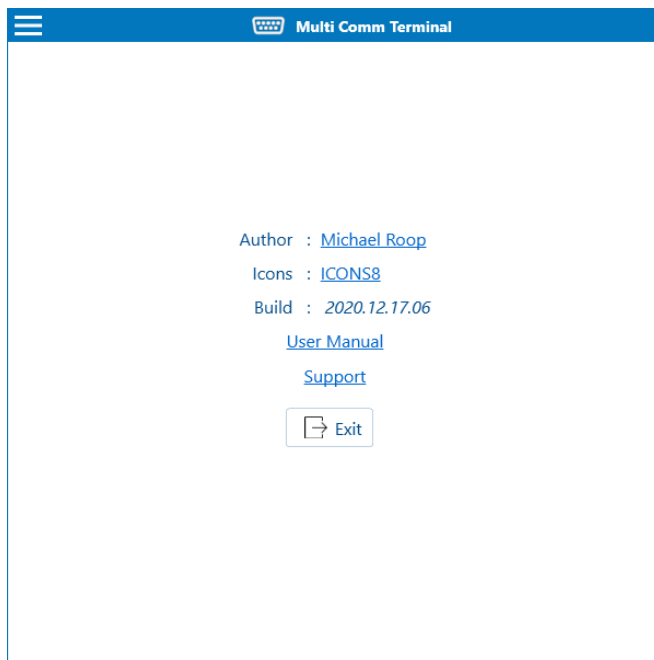


Provides information on selected device. If none is selected, you will get the select dialog first.



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

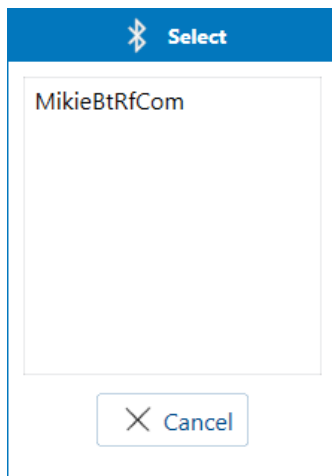
Bluetooth Run Page

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

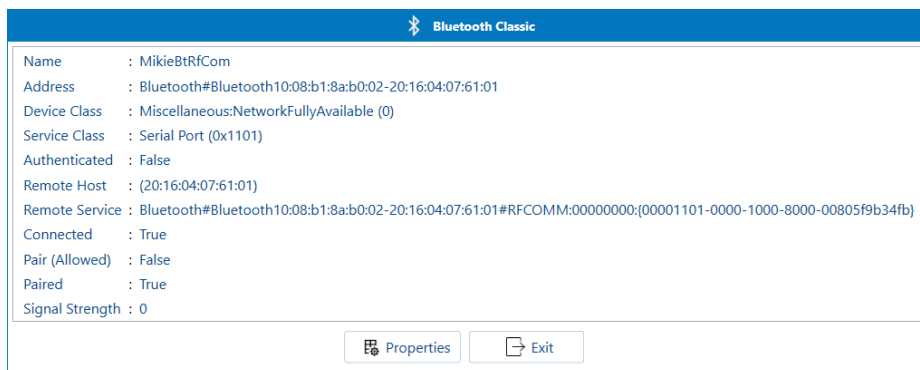


Clicking on a selection will close the dialog and set it as your selected device. You will notice that the device name is now listed on the title bar of the Run page

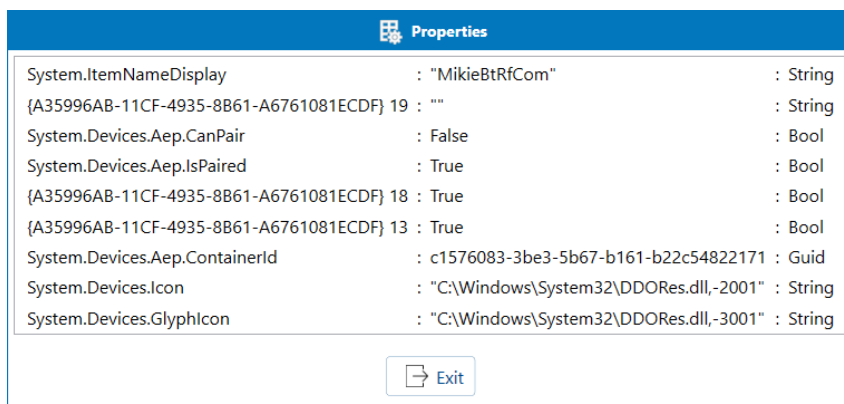


Info

Click on the info icon

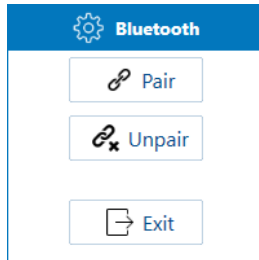


Click on exit to close, or the Properties button to show properties

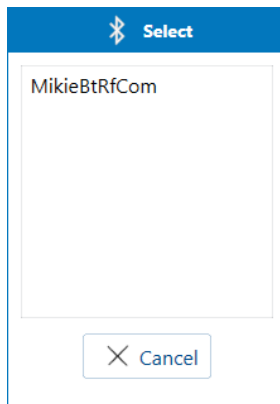


Settings

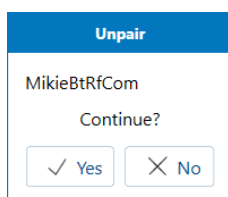
Click on the *Settings* icon



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.

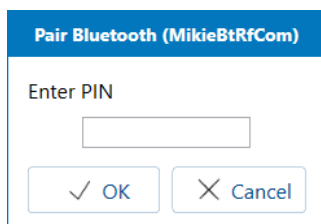


Clicking on a selection will bring up a decision dialog to determine whether to continue the operation. Note that the selection dialog for unpaired devices takes some time to load as it scans for any available device



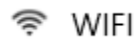
The title shows you if you are going to pair or unpair a device. The name of the device is posted next, the *Continue* message. Clicking on Yes will complete the operation.

When pairing you enter your PIN



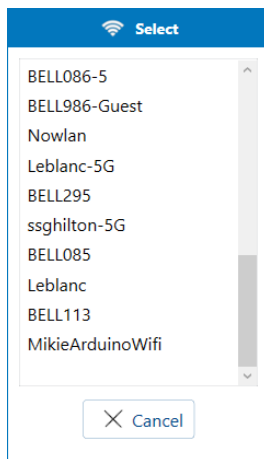
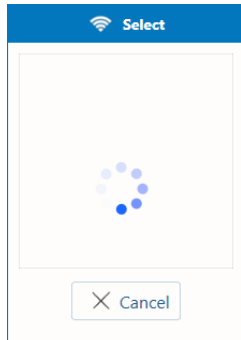
WIFI Run Page

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 Discover button (or info Icon if nothing yet selected) will bring up the Select dialog for WIFI devices. This can take some time as it is scanning all the available WIFI connections.



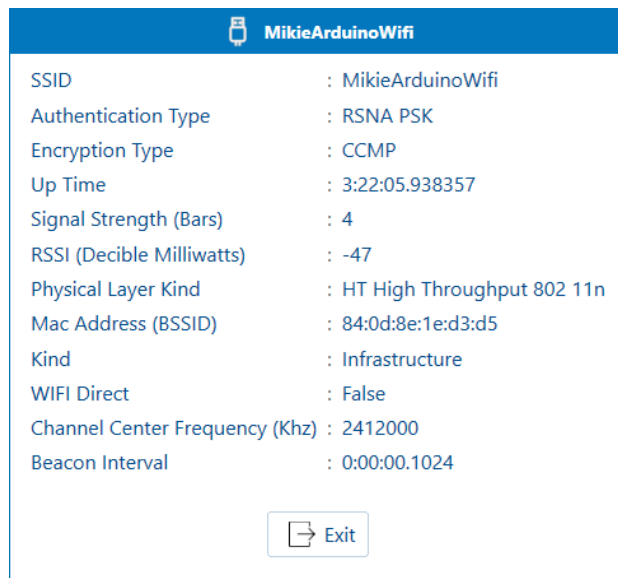
Selecting a connection will close the dialog and set the selection on the run page.



Info

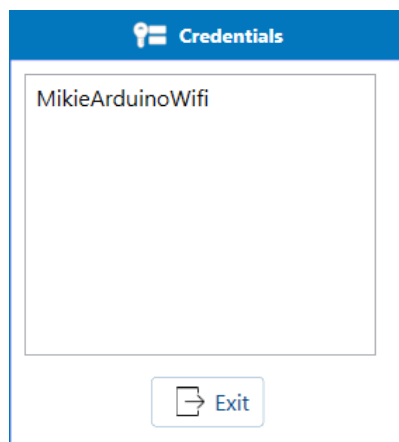
Click on the info icon



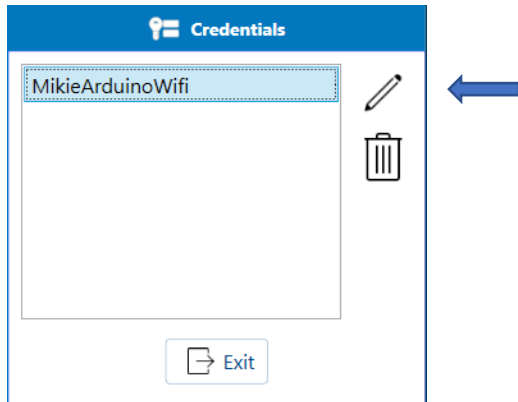


Settings

Click on the Settings icon



The Credentials dialog provides a way to edit or delete credentials and connection parameters for a previous WIFI connection. You can only add to the credentials list when you attempt a new 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

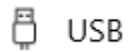
A screenshot of a software window titled "MikieArduinoWifi". It contains two main sections: "Network" and "Socket". Under "Network", there is a "Network Security Key" field filled with 10 dots. Under "Socket", there is a "Host Name/IP" field containing "192.168.4.1" and a "Network Service/Port" field 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

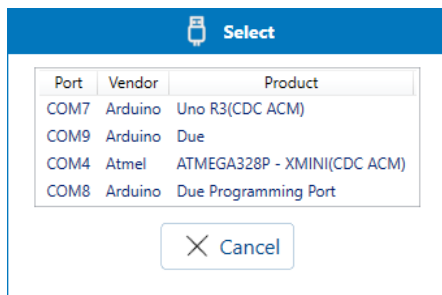
USB Run Page

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 Discover button (or info Icon if nothing yet selected) will bring up the Select dialog for USB devices.

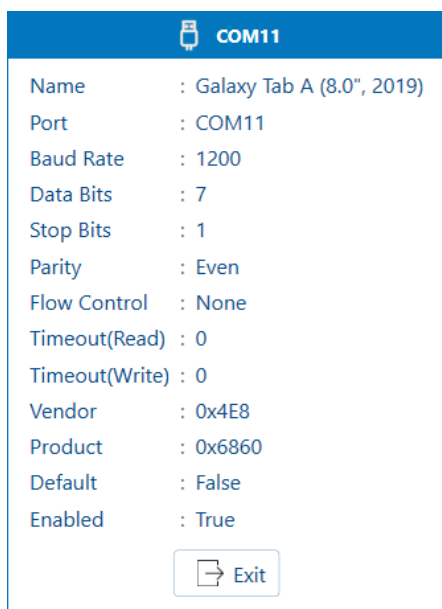


Selecting a device will set it as the device to connect to. Its name will be posted on the title bar. NOTE: There is a bug at the OS level which sometimes gives the incorrect name for the device.



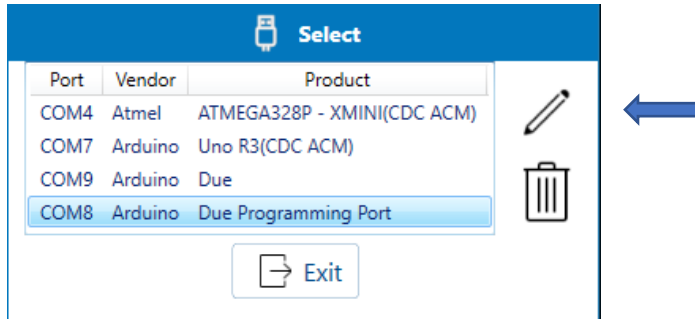
Info

Click on the info icon

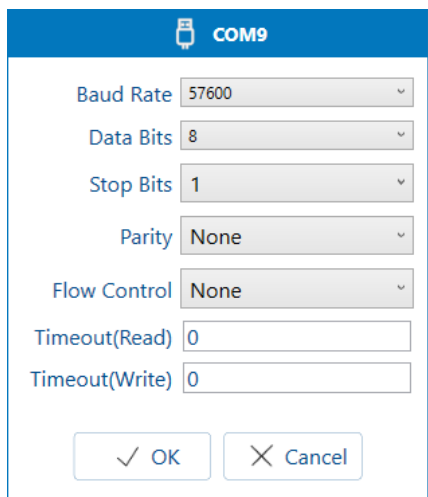


Settings

Click on the Settings icon



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



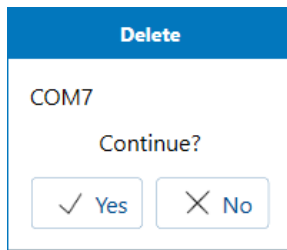
The configuration which comes up is either in the stored list of configurations or the default that the device presents.

This will have the configuration information based on what was harvested from the serial connection and any changes you 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.

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.

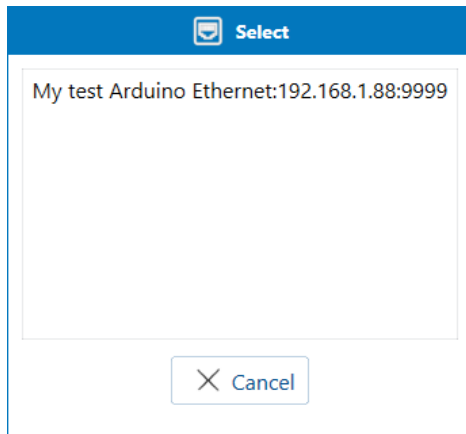
Ethernet Run Page (TBD)

Click on the menu item



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

Clicking on the Discover button (or info Icon if nothing yet selected) 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.

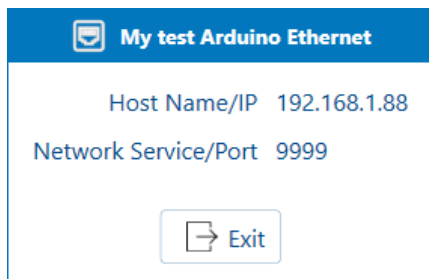


Selecting it will set the title bar to the connection



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 list box with one entry: 'My test Arduino Ethernet:192.168.1.88:9999'. To the right of the list box are three icons: a plus sign (+), a pencil (edit), and a trash can (delete). At the bottom are two buttons: 'Select' with a checkmark icon and 'Exit' with a door icon.

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

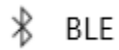
Click on the Add icon to create a new Ethernet connection

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.

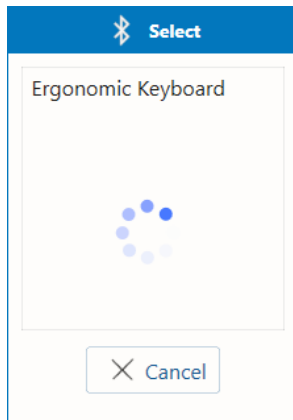
Enter the information and click on Save to store the information

BLE Run Page (Bluetooth Low Energy) (TBD)

Select BLE from the menu



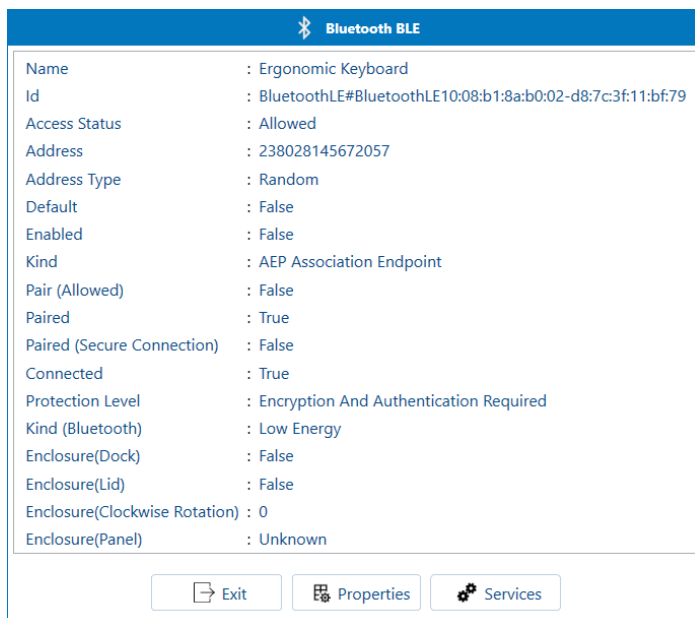
Click on Discover (or info if no selection previously made) to bring up the selection dialog. This can take some time as it is scanning for all available BLE devices



Click on a device to set it as the selected

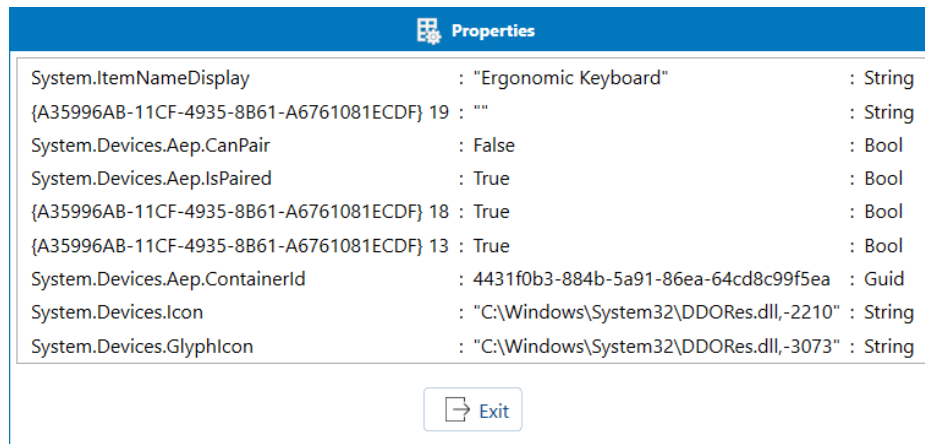
Info

Click on the info icon

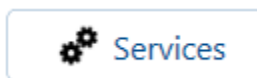


You will get the general information after a connection is established.

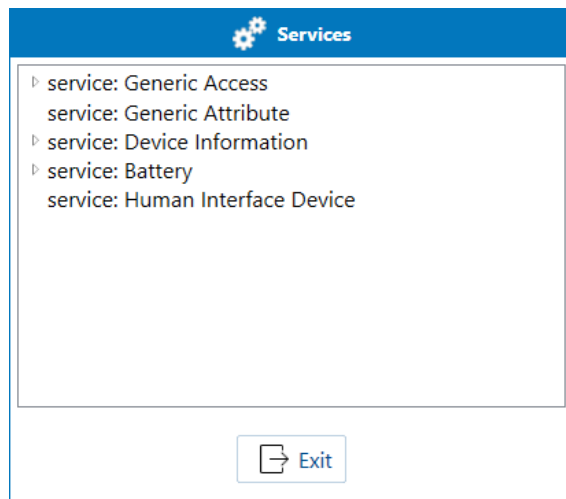
Click on the Properties button to see more information.



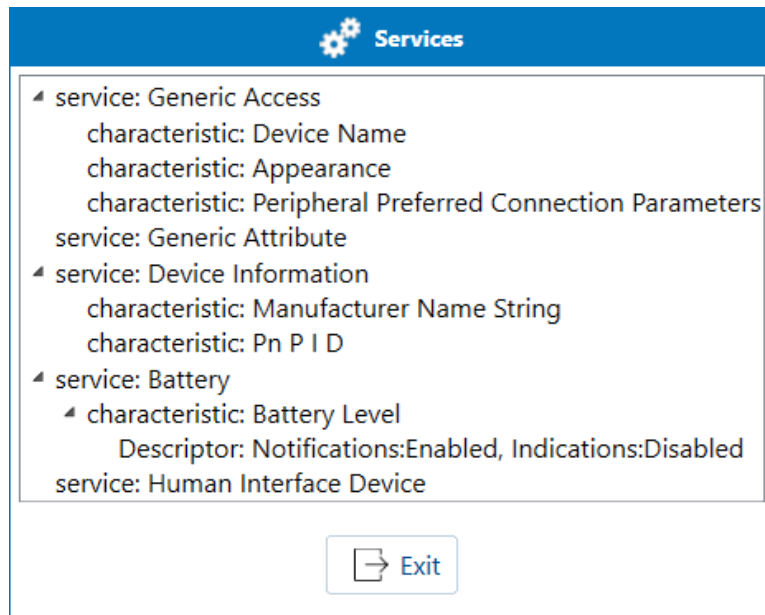
After exit check if there is a Services button on the main info page.



This will show if a connection could be established to gather information on exposed services, characteristics, and Descriptors.



You can then expand to see the Characteristics and Descriptors



Special Note

Connection had been working but a change after a Microsoft update changed BLE to require pairing on connection. Since the Arduino does not yet support BLE pairing, I have removed this functionality until I can find a device to test against

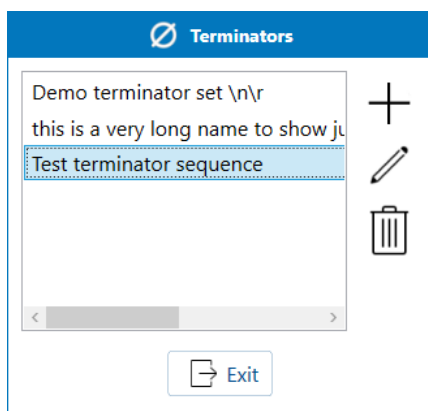
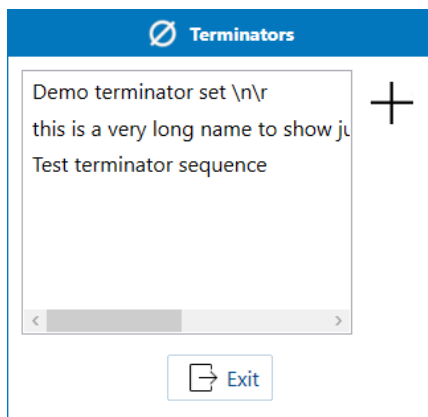
Terminators

Terminators

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

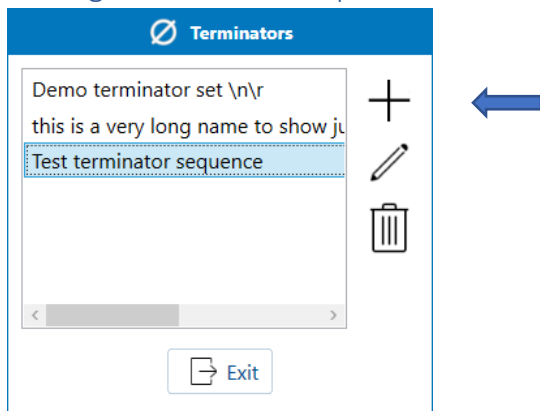
- A terminator sequence contains one or more nonprintable characters that are added to the end of the outgoing message.
- The embedded device must be expecting that sequence to determine when an incoming message is complete.
- The embedded device must also all those terminators to its outgoing messages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

On Open, only the Add button is visible.

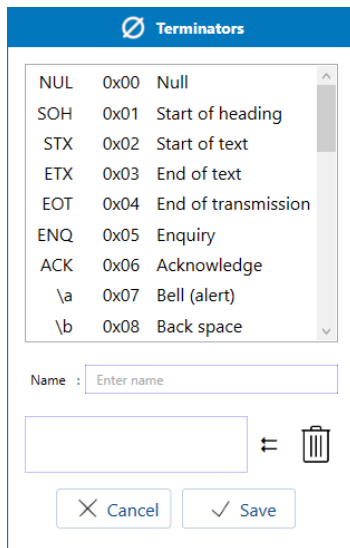


When you select a terminator set, the edit and delete button will appear.

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.

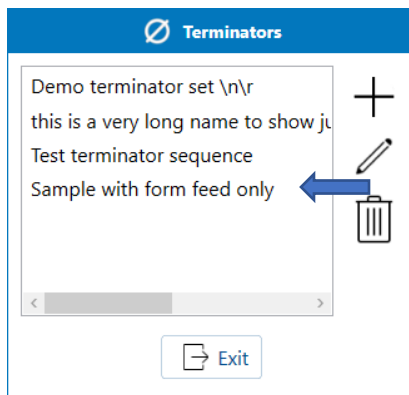


The result after clicking on the delete icon once

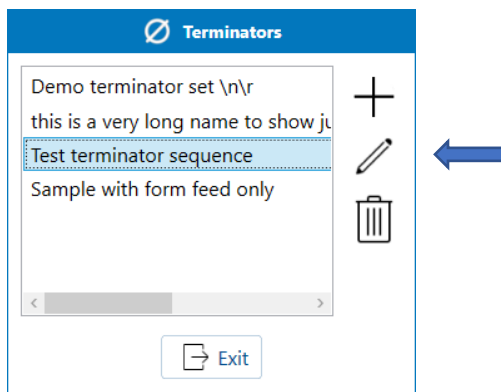
Make sure you enter a name then click on Save

Name :

Your new terminator set is now stored



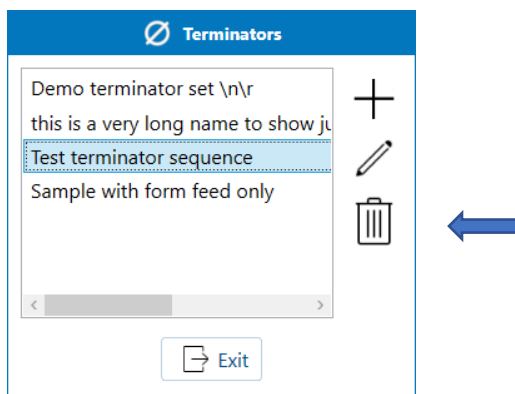
Edit Terminator set



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

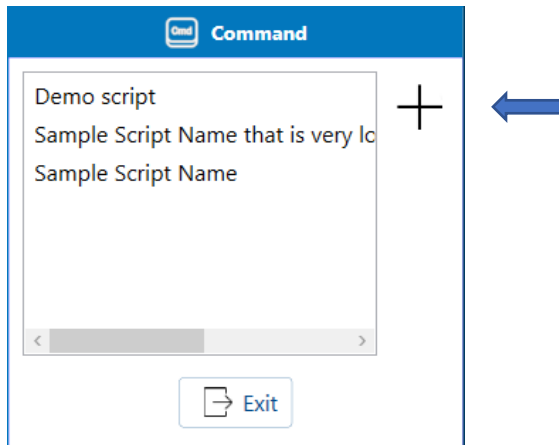
✗ No

Commands



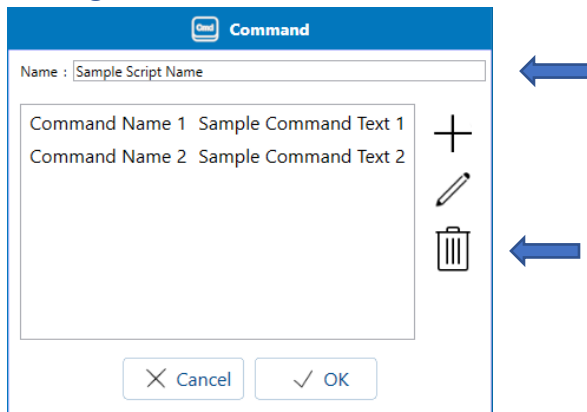
Click on the command menu item

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

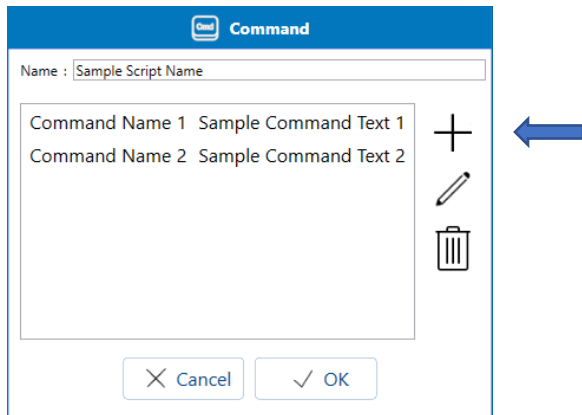


To add a new Command set, click on the Add button or select an existing to bring up more options.

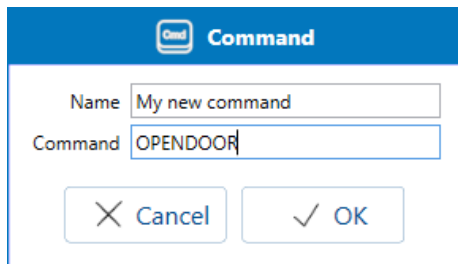
Adding a command list



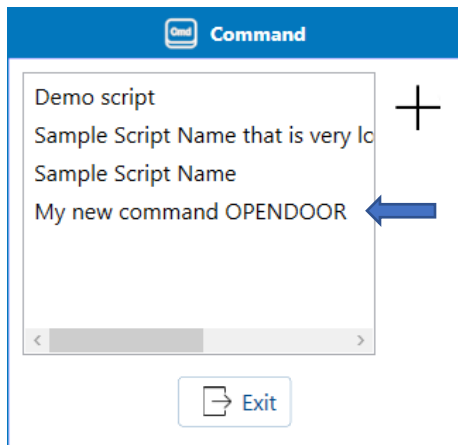
Edit the name. Edit the commands or select them and click on the Delete icon.



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



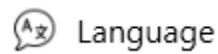
Enter the name that will be displayed for this command. It will show up on the list. The Command text (message) is what is sent to the embedded device. It must be exactly what the device is expecting. Terminators are added apart from this text. Select *Ok* to add the command



The new command is added to the command set

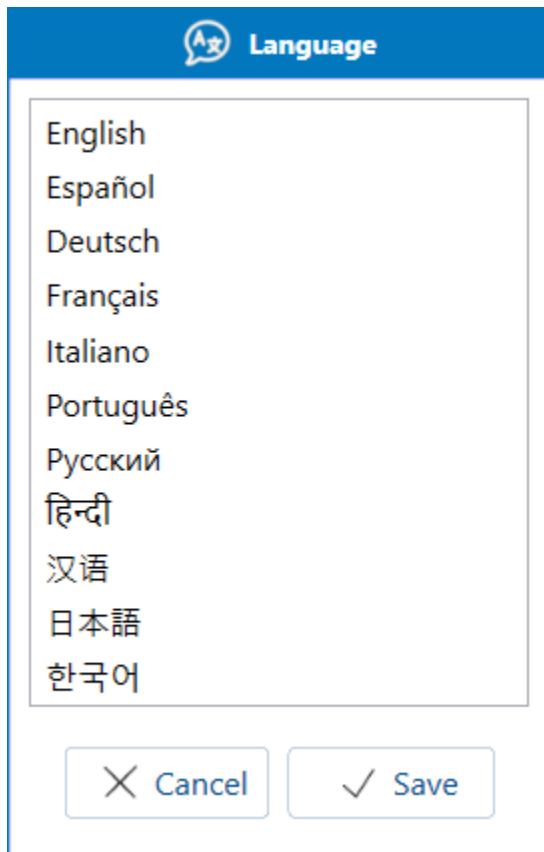
To delete a command, highlight it and click on the *Delete* icon

Language



Language

Clicking on this will bring up the Language selector. Currently there are 11 languages supported for UI elements



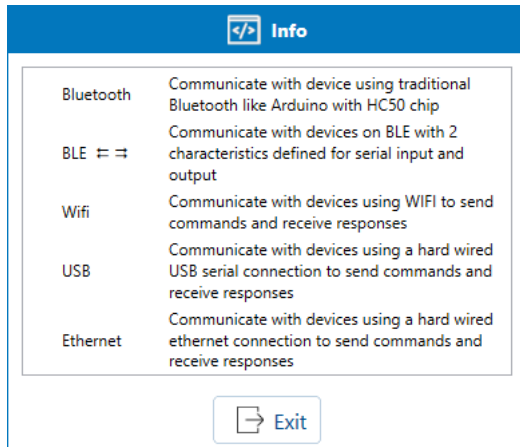
Highlight the requested language and you will see the language change on all open windows. Click Save to save that language as the current. It will be saved as your preference for future sessions. Click on Cancel to revert to previous language

Code Samples

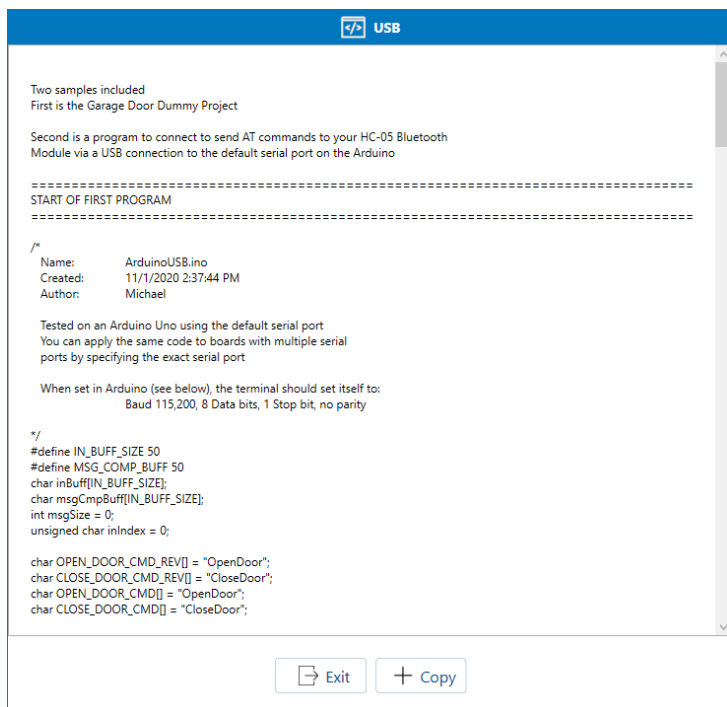
Select the menu item

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:      ArduinoBLESerial.ino
// Created:   4/28/2020 8:55:58 PM
// Author:    Michael
//
// Sets up the BLE to simulate a serial connection by having one characteristic
// configured as an incoming channel, and another one as the outgoing
//
// Tested on the Arduino UNO WIFI Rev2
//
// Was working but a subsequent update of Windows 10 put in BLE pairing on
// connect which is not yet supported in the Arduino library so it will fail
// on connection
//
// -----
#include <ArduinoBLE.h>
#include <string.h>

#ifndef SECTION_DATA

#define MAX_BLOCK_SIZE 20

// Create services that will act as a serial port. Max 20 bytes at a time
BLEService serialService("9999");

// Out channel 0x99,0x98 = 39,320 base 10. Caller reads or gets notifications from
this device
// Need a write in Setup with block of MAX_BLOCK_SIZE size OR it is not recognized by
caller
BLECharacteristic outputCharacteristic("9998", BLERead | BLENotify, MAX_BLOCK_SIZE);

// In channel 0x99,0x97 = 39,319 base 10. Caller writes to this device
BLECharacteristic inputCharacteristic("9997", BLEWrite, MAX_BLOCK_SIZE);

// 0x2901 is CharacteristicUserDescription data type
BLEDescriptor outputDescriptor("2901", "Serial Output");
BLEDescriptor inputDescriptor("2901", "Serial Input");

bool hasInput = false;
#define IN_BUFF_SIZE 500
char buff[IN_BUFF_SIZE];
unsigned char inIndex = 0;

#endif

// the setup function runs once when you press reset or power the board
void setup() {
    // TODO - This needs to be removed to work if not connected to USB serial port
    Serial.begin(9600);
    while (!Serial) {
        ; // wait for serial port to connect. Needed for Native USB only
    }
    Serial.println("Debug serial port active");

    pinMode(LED_BUILTIN, OUTPUT);
    if (!BLE.begin()) {
        Serial.println("Failed to start BLE");
        while (1);
    }
}
```

```

else {
    Serial.println("BLE begun");
}

// Name that users see in list for 'Add Device'
BLE.setLocalName("Test Arduino BLE serial device");
// The Device Name Characteristic
BLE.setDeviceName("Test BLE serial device");

// assign event handlers for connected, disconnected events
BLE.setEventHandler(BLEConnected, bleOnConnectHandler);
BLE.setEventHandler(BLEDisconnected, bleOnDisconnectHandler);

setupSerialDescriptor(serialService, outputCharacteristic, outputDescriptor);

setupSerialDescriptor(serialService, inputCharacteristic, inputDescriptor);
inputCharacteristic.setEventHandler(BLEWritten, inBytesWritten);

// Add service to BLE and start advertising
BLE.addService(serialService);
BLE.setAdvertisedService(serialService);
BLE.advertise();
Serial.println("Bluetooth device active, waiting for connections...");
}

// the loop function runs over and over again until power down or reset
void loop() {
    BLEDevice central = BLE.central();
    if (central) {
        // This will hold up the loop function as long as the connection persists
        while (central.connected()) {
            // The message processing is done via the event handler for incoming bytes
        }
        // Since the current connection has terminated wipe out accumulated in bytes
        ResetInBuffer();
    }
}

// Initialize the serial Characteristic, add Descriptor and connect to serial
void setupSerialDescriptor(BLEService& service, BLECharacteristic& characteristic,
BLEDescriptor& desc) {
    byte initBuffContents[MAX_BLOCK_SIZE];
    // If we initialize with 0x00 (null) creates weird output in future writes
    memset(initBuffContents, 0x20, MAX_BLOCK_SIZE);
    characteristic.writeValue(initBuffContents, MAX_BLOCK_SIZE);
    characteristic.addDescriptor(desc);
    service.addCharacteristic(characteristic);
}

void ProcessIncomingBuff() {
    // Doing \n\r terminator sequence
    if (buff[inIndex] == '\r') {
        Serial.println("Printing msg in buffer");
        hasInput = true;

        Serial.print("Sending back ");
        Serial.print(inIndex);
        Serial.println(" bytes to write");

        // This works by iterating through BLOCK_SIZE byte blocks.

```

```

    int count = inIndex / MAX_BLOCK_SIZE;
    int last = (inIndex % MAX_BLOCK_SIZE);
    int lastIndex = 0;
    for (int i = 0; i < count; i++) {
        lastIndex = i * MAX_BLOCK_SIZE;
        outputCharacteristic.writeValue(&buff[lastIndex], MAX_BLOCK_SIZE);
    }
    if (last > 0) {
        if (lastIndex > 0) {
            lastIndex += MAX_BLOCK_SIZE;
        }
        outputCharacteristic.writeValue(&buff[lastIndex], last);
    }

    ResetInBuffer();
}
else {
    //
    inIndex++;
    // Wipe out buffer if too long
    if (inIndex >= IN_BUFF_SIZE) {
        ResetInBuffer();
        Serial.println("Corrupt BT input. Buffer purged");
    }
}
}

```

```

void ResetInBuffer() {
    // Reset with spaces. Acts weird if you use 0x00 null)
    memset(buff, 0x20, IN_BUFF_SIZE);
    inIndex = 0;
}

```

```

byte inBuff[MAX_BLOCK_SIZE];

```

```

// Event handler for when data is written to the inByte characteristic
void inBytesWritten(BLEDevice device, BLECharacteristic byteCharacteristic) {
    digitalWrite(LED_BUILTIN, HIGH);

    int count = inputCharacteristic.readValue(inBuff, MAX_BLOCK_SIZE);
    ProcessIncomingBuff2(inBuff, count);
    digitalWrite(LED_BUILTIN, LOW);
}

```

```

void ProcessIncomingBuff2(uint8_t* inData, int length) {
    // index is pos of next write
    memcpy(&buff[inIndex], inData, length);

    int newIndex = inIndex + length;
    // This is where I would process the data. Remove complete command, etc
    inIndex = newIndex;

    Serial.write(inData, length); Serial.println("");
    Serial.write(buff, newIndex); Serial.println("");

    // Just bounce back whatever came in
    outputCharacteristic.writeValue(inData, length);
    // Check if there was a \n and simply obliterate main buffer
    for (int i = 0; i < length; i++) {
        if (inData[i] == '\n') {

```

```
        ResetInBuffer();
        break;
    }
}

// Event handler for on BLE connected
void bleOnConnectHandler(BLEDevice central) {
    Serial.print("CONNECTED, central: ");
    Serial.println(central.address());
}

// Event handler for on BLE disconnected
void bleOnDisconnectHandler(BLEDevice central) {
    Serial.print("DISCONNECTED, central: ");
    Serial.println(central.address());
}
```

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 (OpenDoorlsdflkdjdfldj)
    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
    }
}

```