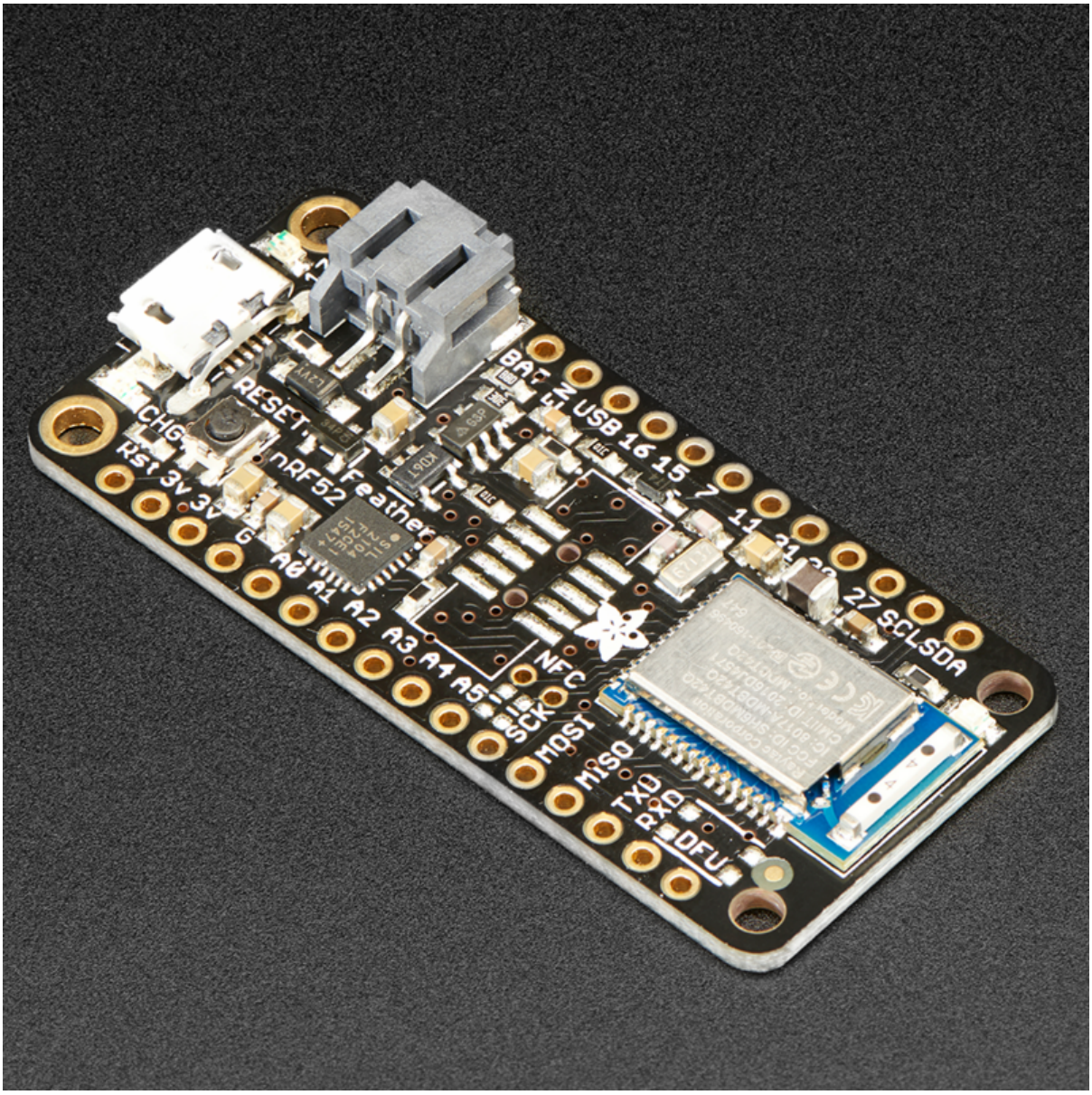
Bluefruit nRF52 Feather Learning Guide

Created by Adam Bachman



https://learn.adafruit.com/bluefruit-nrf52-feather-learning-guide

Last updated on 2024-03-21 11:03:05 AM EDT

Table of Contents

Introduction 7

- nRF52832 Technical Details
- nRF51 or nRF52 Bluefruit Devices?

Device Pinout 9

- Special Notes
- Power Pins
- Analog Inputs
- PWM Outputs
- I2C Pins

Assembly 12

- Header Options!
- Soldering in Plain Headers
- Prepare the header strip:
- Add the breakout board:
- And Solder!
- Soldering on Female Header
- Tape In Place
- Flip & Tack Solder
- And Solder!

Power Management 22

- Battery + USB Power
- Power Supplies
- Measuring Battery
- ENable pin
- Alternative Power Options

Arduino Support Setup 26

- 1. BSP Installation
- 2. LINUX ONLY: adafruit-nrfutil Tool Installation
- 3. Update the bootloader (nRF52832 ONLY)
- Advanced Option: Manually Install the BSP via 'git'

Arduino Board Testing 30

- 1. Select the Board Target
- 2. Select the USB CDC Serial Port
- 2.1 Download & Install CP2104 Driver (nRF52832)
- 2.2 Download & Install Adafruit Driver (nRF52840 Windows)
- 3. Update the bootloader (nRF52832 Feather Only)
- 4. Run a Test Sketch

Using the Bootloader 35

- Forcing Serial Boot Mode
- Factory Reset
- Advanced: OTA DFU Bootloader

Updating the Bootloader 37

- Upgrading an Existing Bootloader

Flashing the Bootloader 39
• 1. Updating the Bootloader with a Segger J-Link and Arduino IDE
• 2. Manually Burning the Bootloader via nrfjprog
• 3. Manually Burning the Bootloader via AdaLink
Arduino BLE Examples 41
• Example Source Code
• Documented Examples
Advertising: Beacon 43
• Complete Code
• Output
BLE UART: Controller 49
• Setup
• Complete Code
Custom: HRM 56
• HRM Service Definition
• Implementing the HRM Service and Characteristics
• Service + Characteristic Setup Code Analysis
• Full Sample Code
BLE Pin I/O 63
• Setup
• Complete Code
Central BLEUART 78
• Client Services
• Scanner
• Central Role
• Full Sample Code
Dual Roles BLEUART 84
• Server & Client Service Setup
• Peripheral Role
• Central Role
• Advertising and Scanner
• Full Sample Code
Custom: Central HRM 90
• HRM Service Definition
• Implementing the HRM Service and Characteristics
• Client Service + Characteristic Code Analysis
• Full Sample Code
Arduino Bluefruit nRF52 API 98
AdafruitBluefruit 99
• API
• Examples
BLEGap 101

BLEAdvertising 102
• API
• Related Information
• Example
BLEScanner 104
• API
• setRxCallback(rx_callback_t fp)
• void useActiveScan(bool enable);
• void filterRssi(int8_t min_rssi);void filterMSD(uint16_t manuf_id);void filterUuid(BLEUuid ble_uuid);void filterUuid(BLEUuid ble_uuid1, BLEUuid ble_uuid2);void filterUuid(BLEUuid ble_uuid1, BLEUuid ble_uuid2, BLEUuid ble_uuid3);void filterUuid(BLEUuid ble_uuid1, BLEUuid ble_uuid2, BLEUuid ble_uuid3, BLEUuid ble_uuid4);void filterUuid(BLEUuid ble_uuid[], uint8_t count);
• void clearFilters(void);
• bool start(uint16_t timeout = 0); bool stop(void);
• void restartOnDisconnect(bool enable);
• Examples
BLEService 109
• Basic Usage
• Order of Operations (Important!)
• API
• Example
BLECharacteristic 112
• Basic Usage
• Order of Operations (Important!)
• API
• Example
BLEClientService 117
• Basic Usage
• API
• Example
BLEClientCharacteristic 122
• Basic Usage
• API
• Example
BLEDiscovery 127
• API
BLEDis 128
• API
• Example
• Output
BLEUart 131
• API
• Example
BLEClientUart 133
• API
• Examples

BLEBeacon 137
• API
• Example
• Testing
BLEMidi 140
• API
• Installing the Arduino MIDI Library
• Example
• Usage
BLEHidAdafruit 144
• API
• Example Sketches
• Bonding HID Devices
• Setting up your Bluefruit device for bonding
• Bonding on iOS
• Testing the HID Keyboard and Bonding
BLEAncs 148
• API
• ANCS OLED Example
• Sketch Requirements
• Loading the Sketch
• Pairing to your Mobile Device
• Wait for Alerts
BLEClientCts 153
• API
• Client CTS OLED Example
• Sketch Requirements
• Loading the Sketch
• Pairing to your Mobile Device
• Wait for Time Data
BLECentral 156
nRF52 ADC 157
• Analog Reference Voltage
• Analog Resolution
• Default ADC Example (10-bit, 3.6V Reference)
• Advanced Example (12-bit, 3.0V Reference)
Memory Map 160
• BSP release & Bootloader version
• Flash Memory
• SRAM Layout
• Functions affecting SoftDevice SRAM usage
Software Resources 164
• Bluefruit LE Client Apps and Libraries
• Bluefruit LE Connect (Android/Java)
• Bluefruit LE Connect (iOS/Swift)
• Bluefruit LE Connect for OS X (Swift)
• Bluefruit LE Command Line Updater for OS X (Swift)

• Deprecated: Bluefruit Buddy (OS X)
• ABLE (Cross Platform/Node+Electron)
• Bluefruit LE Python Wrapper
• Debug Tools
• AdaLink (Python)
• Adafruit nRF51822 Flasher (Python)

Downloads 170

• MyNewt Bootloader
• Module Details
• Schematic
• Schematic (pre March 16 2018)

FAQs 172

Introduction

The **Adafruit Feather nRF52 Bluefruit** is our latest easy-to-use all-in-one Bluetooth Low Energy board, with a native-Bluetooth chip, the nRF52832! It's our take on an 'all-in-one' Arduino-compatible + Bluetooth Low Energy with built in USB and battery charging.

This chip has twice the flash, SRAM and performance of the earlier nRF51-based Bluefruit modules. Best of all it has Arduino IDE support, so there is no 'helper' chip like the ATmega32u4 or ATSAMD21. Instead, this chip is programmed directly! It has tons of awesome peripherals: plenty of GPIO, analog inputs, PWM, timers, etc. Leaving out the extra microcontroller means the price, complexity and power-usage are all lower/better. It allows you to run code directly on the nRF52832, straight from the Arduino IDE as you would with any other MCU or Arduino compatible device. A single MCU means better performance, lower overall power consumption, and lower production costs if you ever want to design your own hardware based on your Bluefruit nRF52 Feather project!

The chips are pre-programed with an auto-resetting bootloader so you can upload quickly in the Arduino IDE with no button-pressing. Want to program the chip directly? You can use our command line tools with your favorite editor and toolchain.

And to get you up and running quickly, we've done all the heavy lifting of getting the low level BLE stack into shape so that you can focus on your project from day one!

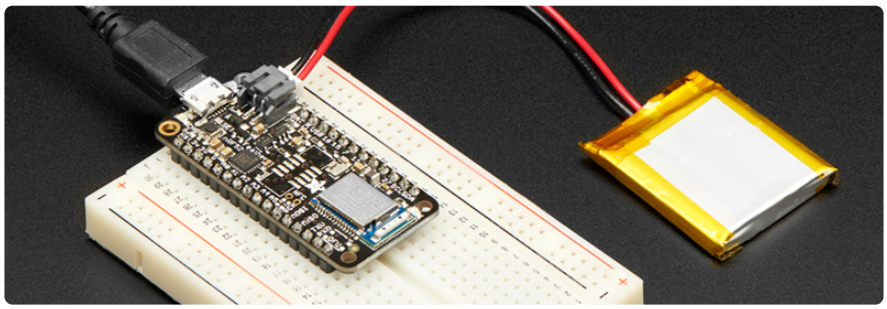
nRF52832 Technical Details

- ARM Cortex M4F (with HW floating point acceleration) running at 64MHz
- 512KB flash and 64KB SRAM
- **Built in USB Serial converter for fast and efficient programming and debugging**
- Bluetooth Low Energy compatible 2.4GHz radio (Details available in the nRF52832 (https://adafru.it/vaJ) product specification)
- **FCC / IC / TELEC certified module**
- Up to +4dBm output power
- 1.7v to 3.3v operation with internal linear and DC/DC voltage regulators
- 19 GPIO, 8 x 12-bit ADC pins, up to 12 PWM outputs (3 PWM modules with 4 outputs each)
- Pin #17 red LED for general purpose blinking
- Power/enable pin
- Measures 2.0" x 0.9" x 0.28" (51mm x 23mm x 8mm) without headers soldered in

- Light as a (large?) feather - 5.7 grams
- 4 mounting holes
- Reset button
- Optional SWD connector for debugging
- Works out of the box with just about all of our Adafruit FeatherWings! (https://adafru.it/vby) (Wings that require the UART like the GPS FeatherWing won't work)

Further technical details available in the nRF52832 (https://adafru.it/vaJ) product specification.

Like all of our Feather boards, the Bluefruit nRF52 Feather includes on board USB-based LIPO charging, and has a standard LIPO battery connector to make your wireless projects genuinely 'wireless' at no additional cost (aside from the LIPO cell itself).



nRF51 or nRF52 Bluefruit Devices?

The Bluefruit nRF52 Feather (based on the nRF52832 (https://adafru.it/vaJ) SoC) is quite different from the earlier nRF51822 based Bluefruit products (Bluefruit M0 Feather (http://adafru.it/2995), etc.), both of which will continue to exist.

From a hardware perspective, the nRF52 Feather is based on a much more powerful ARM Cortex M4F processor, with 512KB flash, 64KB SRAM and hardware floating point acceleration ... whereas the earlier nRF51822 is based on the smaller ARM Cortex M0 core (fewer internal instructions), with 256KB flash and either 16KB or 32KB SRAM.

More importantly, the design approach that we took with the nRF52 is completely different:

- nRF51 based Bluefruit boards run as modules that you connect to via an external MCU (typically an Atmel 32u4 or a SAMD21), sending AT style commands over SPI or UART.

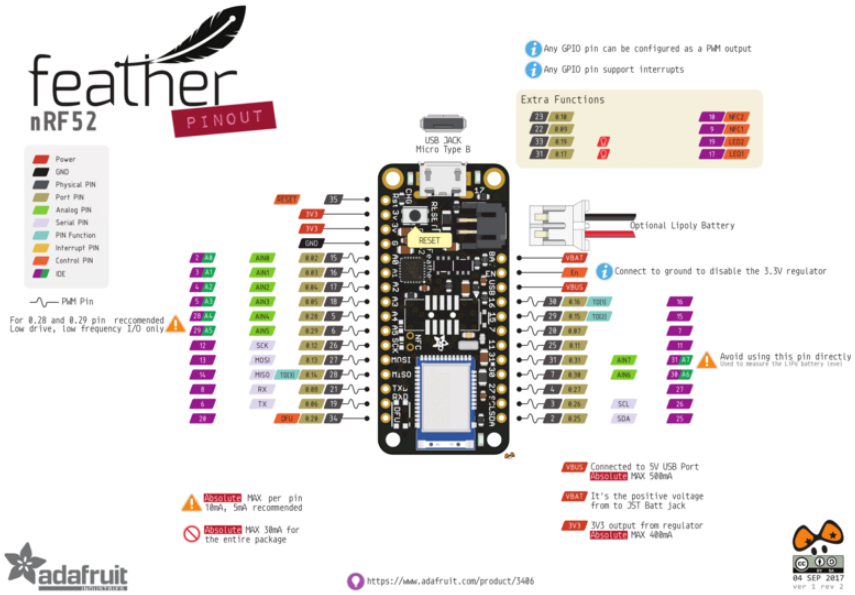
- **With the nRF52, you run all of your code directly on the nRF52832 and no external MCU is used or required!**

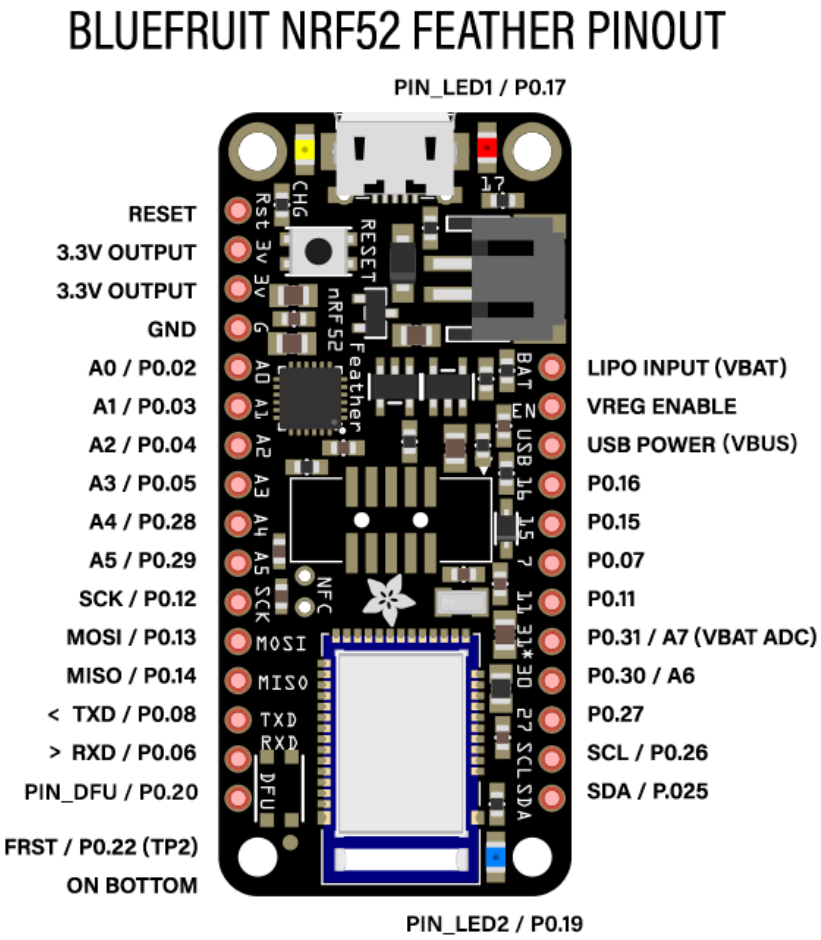
This change of design helps keep the overall costs lower, allows for far better performance since you aren't limited by the SPI or UART transport channel, and can help improve overall power consumption.

As a tradeoff, it also means a completely different API and development process, though!

nRF51 Bluefruit sketches won't run on nRF52 Bluefruit hardware without modification! The two device families have different APIs and programming models, and aim to solve your wireless problems in two different ways.

Device Pinout





Please note that the nRF52832 uses a USB serial adapter. RXD/TXD are with respect to the nRF52.

Special Notes

The following pins have some restrictions that need to be taken into account when using them:

- **PIN_DFU / P0.20**: If this pin is detected to be at GND level at startup, the board will enter a special serial bootloader mode and will not execute any user code, going straight into bootloader mode. If you wish to use this pin as a standard GPIO, make sure that it is pulled high with a pullup resistor so that your code will execute normally when the MCU starts up.
- **P0.31 / A7**: This pin is hard wired to a voltage-divider on the LIPO battery input, allow you to safely measure the LIPO battery level on your device. If possible, you should avoid using this pin as an input because you will lose the ability to read the battery voltage. You can use it as an output just make sure to switch

the pin to analog input when you want to do the battery read, then back to output when toggling pins

- **FRST/P0.22**: Setting this pin to GND at startup will cause the device to perform a factory reset at startup, erasing and config data as well as the user sketch. At the next reset, you should enter serial bootloader mode by default, since no user sketch will be present. You can use this to recover 'bricked' boards, but if you don't wish to do this be careful not to have FRST low at startup. By default, a weak internal pull-up resistor is enabled on this pin during the bootloader phase. **FRST does not work with Mynewt bootloader !!**

Power Pins

- **3.3V Output**: This two pins are connected to the output of the on board 3.3V regulator. They can be used to supply 3.3V power to external sensors, breakouts or Feather Wings.
- **LIPO Input (VBAT)**: This is the voltage supply off the optional LIPO cell that can be connected via the JST PH connector. It is nominally ~3.5-4.2V.
- **VREG Enable**: This pin can be set to GND to disable the 3.3V output from the on board voltage regulator. By default it is set high via a pullup resistor.
- **USB Power (VBUS)**: This is the voltage supply off USB connector, nominally 4.5-5.2V.

Analog Inputs

The 8 available analog inputs can be configured to generate 8, 10 or 12-bit data (or 14-bits with over-sampling), at speeds up to 200kHz (depending on the bit-width of the values generated), based on either an internal 0.6V reference or the external supply.

The following default values are used:

- **Default voltage range**: 0-3.6V (uses the internal 0.6V reference with 1/6 gain)
- **Default resolution**: 10-bit (0..1023)

Unlike digital functions, which can be remapped to any GPIO/digital pin, the ADC functionality is tied to specified pins, labelled as A* in the image above (A0, A1, etc.).

PWM Outputs

Any GPIO pin can be configured as a PWM output, using the dedicated PWM block.

Three PWM modules can provide up to 12 PWM channels with individual frequency control in groups of up to four channels.

Please note that DMA based PWM output is still a work in progress in the initial release of the nR52 BSP, and further improvements are planned here.

I2C Pins

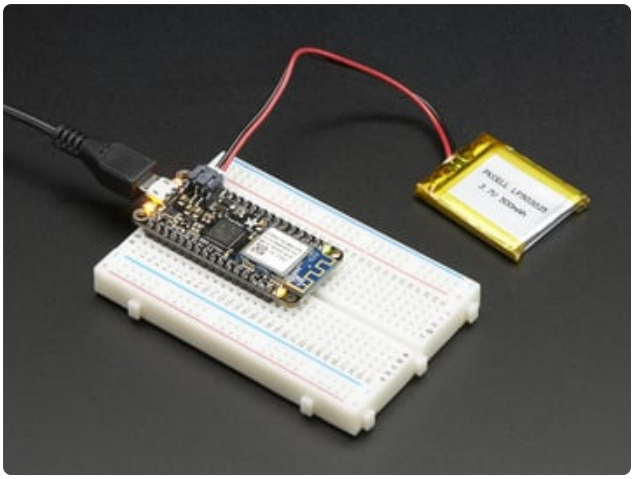
I2C pins on the nRF52832 require external pullup resistors to function, which are not present on the Adafruit nRF52 Feather by default. You will need to supply external pullups to use these. All Adafruit I2C breakouts have appropriate pullups on them already, so this normally won't be an issue for you.

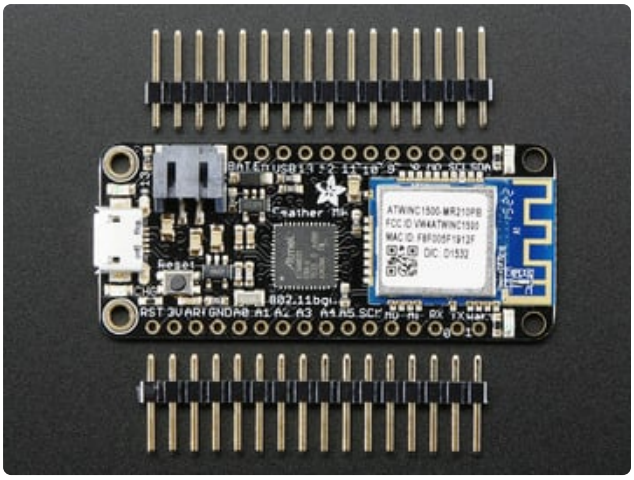
Assembly

We ship Feathers fully tested but without headers attached - this gives you the most flexibility on choosing how to use and configure your Feather

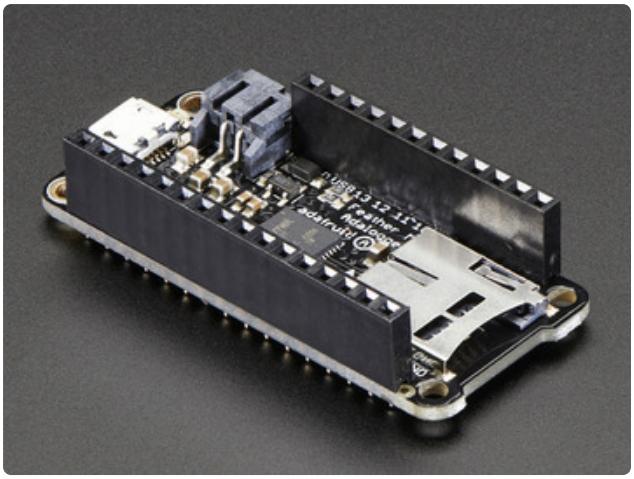
Header Options!

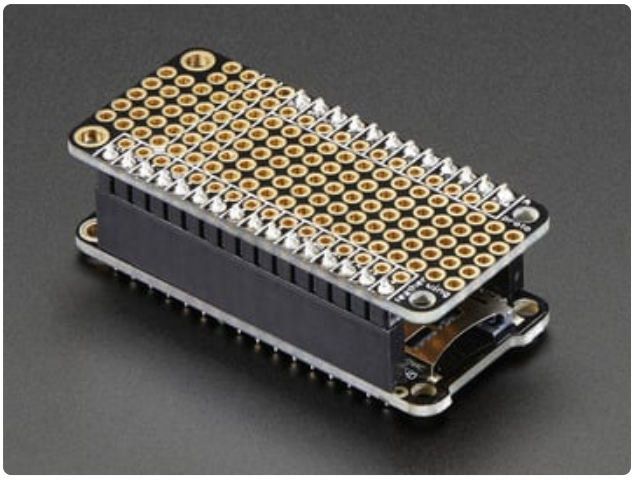
Before you go gung-ho on soldering, there's a few options to consider!

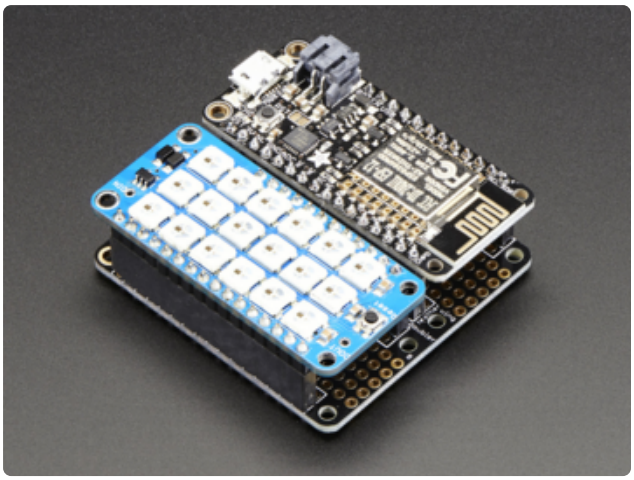




The first option is soldering in plain male headers, this lets you plug in the Feather into a solderless breadboard

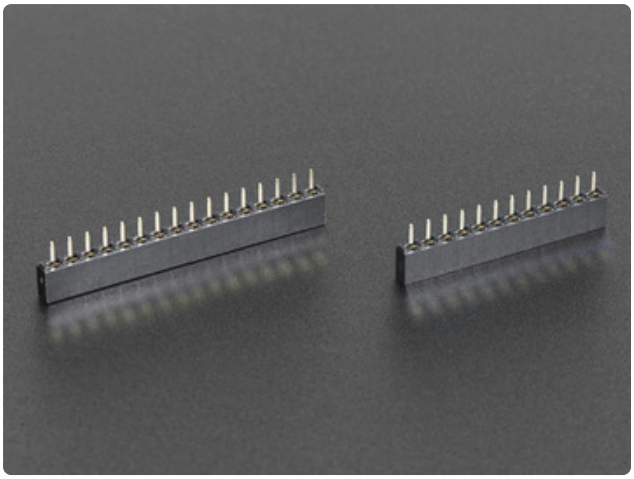


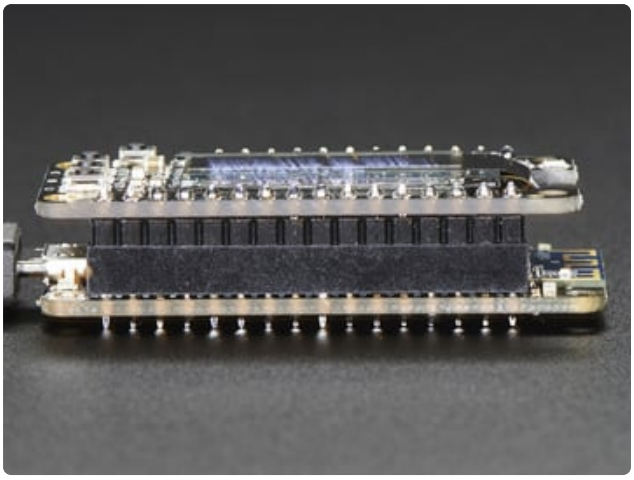




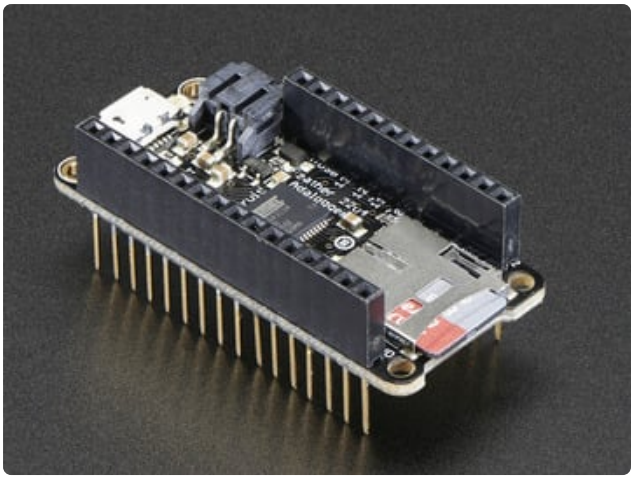
Another option is to go with socket female headers. This won't let you plug the Feather into a breadboard but it will let you attach featherwings very easily

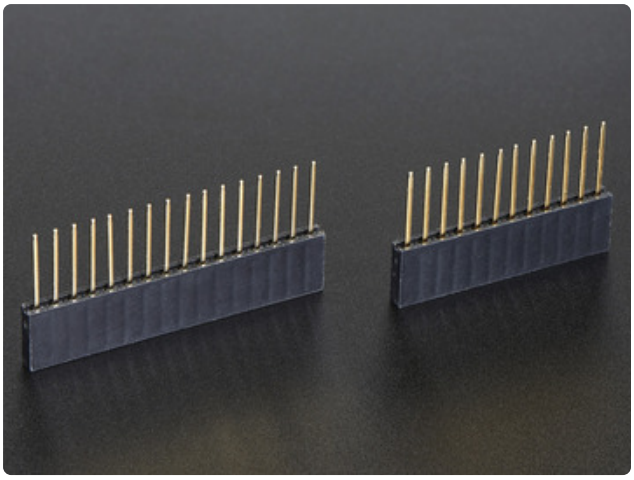
A few Feather boards require access to top-side components like buttons or connectors, making stacking impractical. Sometimes you can stack in the opposite order—FeatherWing underneath—or, if both Feather and Wing require top-side access, place the boards side-by-side with a FeatherWing Doubler (http://adafru.it/2890) or Tripler (http://adafru.it/3417).





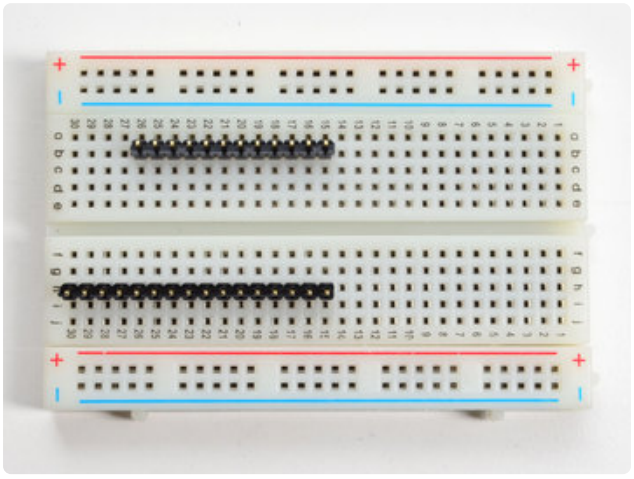
We also have 'slim' versions of the female headers, that are a little shorter and give a more compact shape





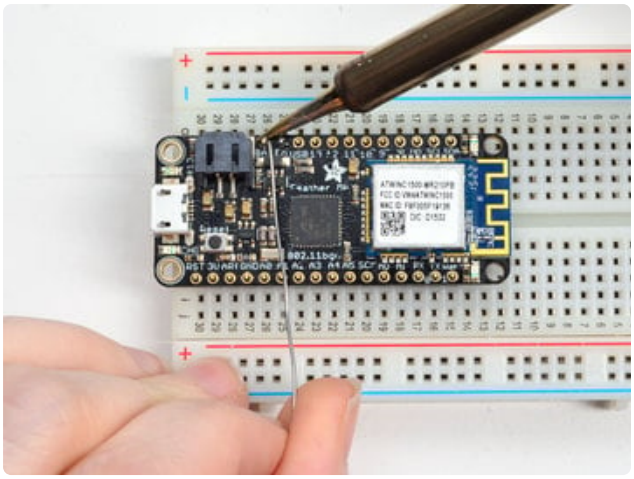
Finally, there's the "Stacking Header" option. This one is sort of the best-of-both-worlds. You get the ability to plug into a solderless breadboard and plug a featherwing on top. But its a little bulky

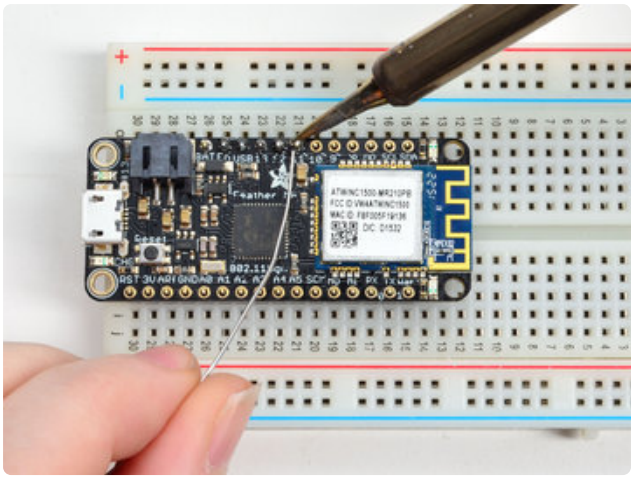
Soldering in Plain Headers

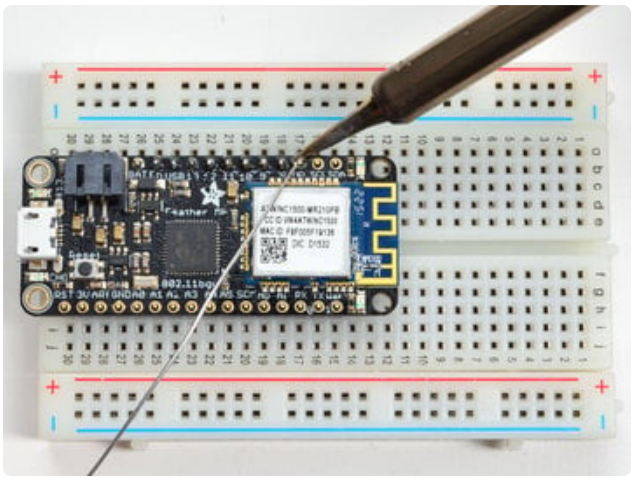


Prepare the header strip:

Cut the strip to length if necessary. It will be easier to solder if you insert it into a breadboard - **long pins down**







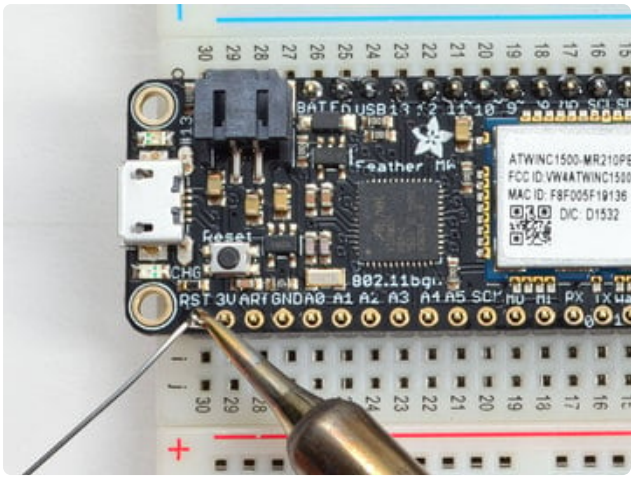
Add the breakout board:

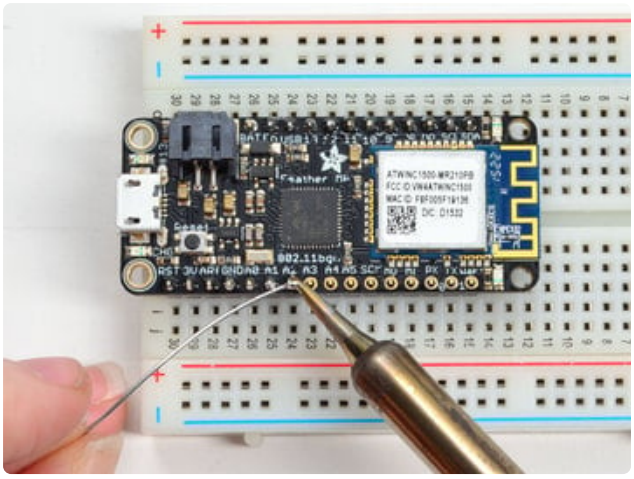
Place the breakout board over the pins so that the short pins poke through the breakout pads

And Solder!

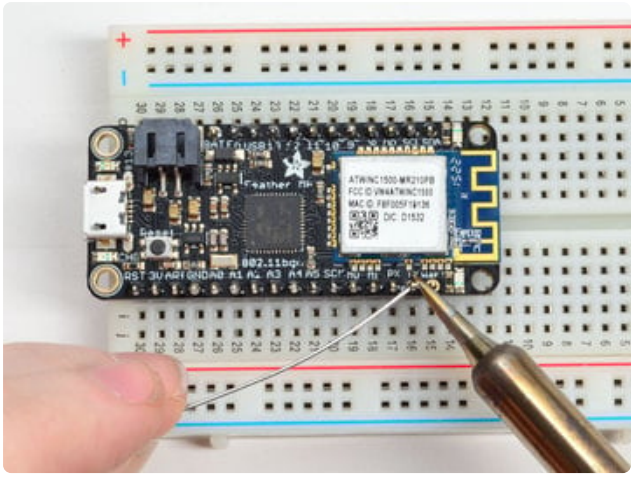
Be sure to solder all pins for reliable electrical contact.

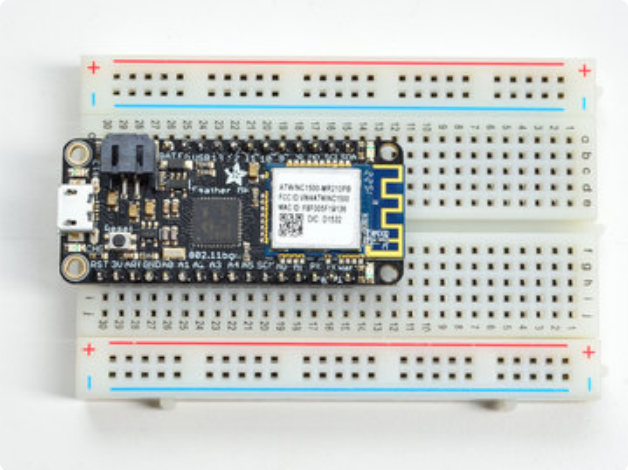
(For tips on soldering, be sure to check out our Guide to Excellent Soldering (https://adafru.it/aTk)).





Solder the other strip as well.



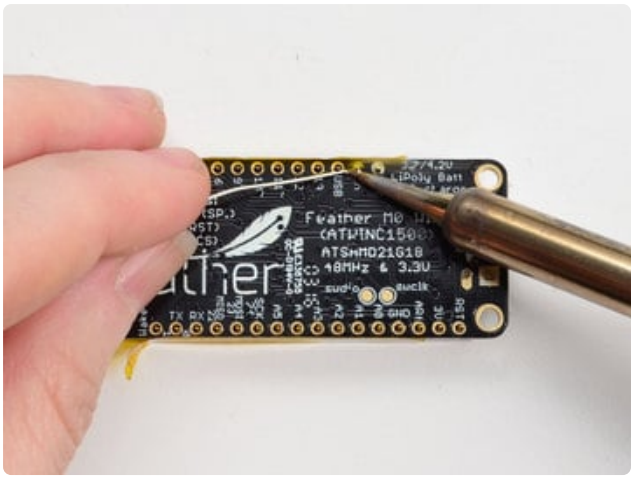


You're done! Check your solder joints visually and continue onto the next steps

Soldering on Female Header

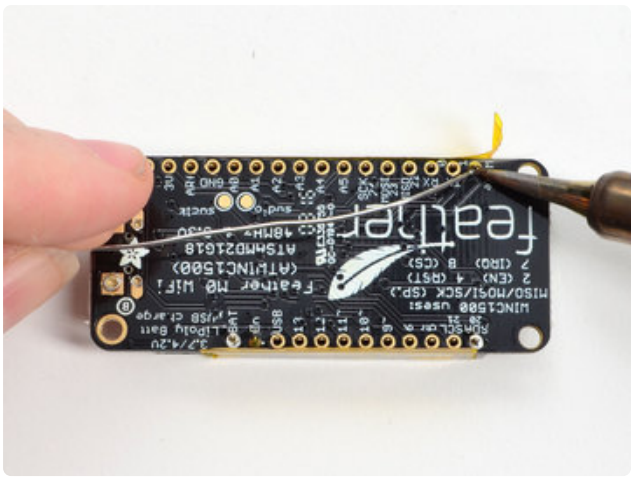
Tape In Place

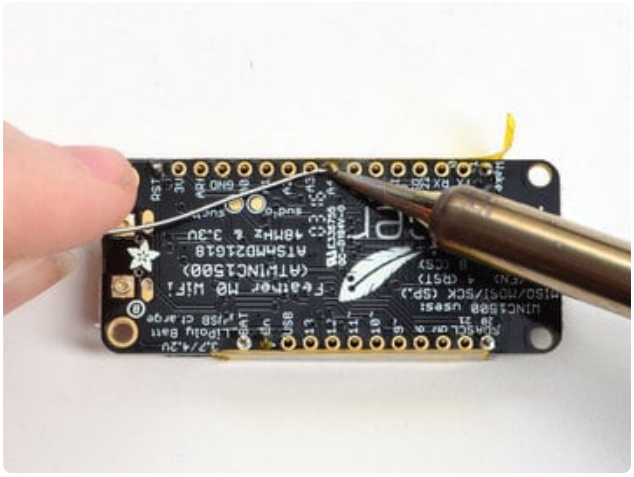
For sockets you'll want to tape them in place so when you flip over the board they don't fall out

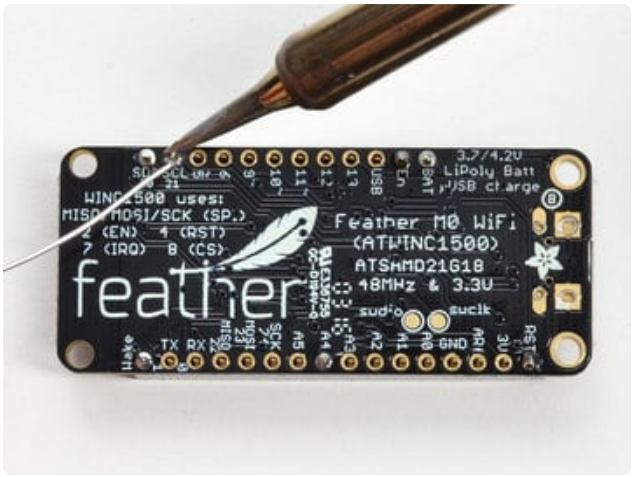


Flip & Tack Solder

After flipping over, solder one or two points on each strip, to 'tack' the header in place







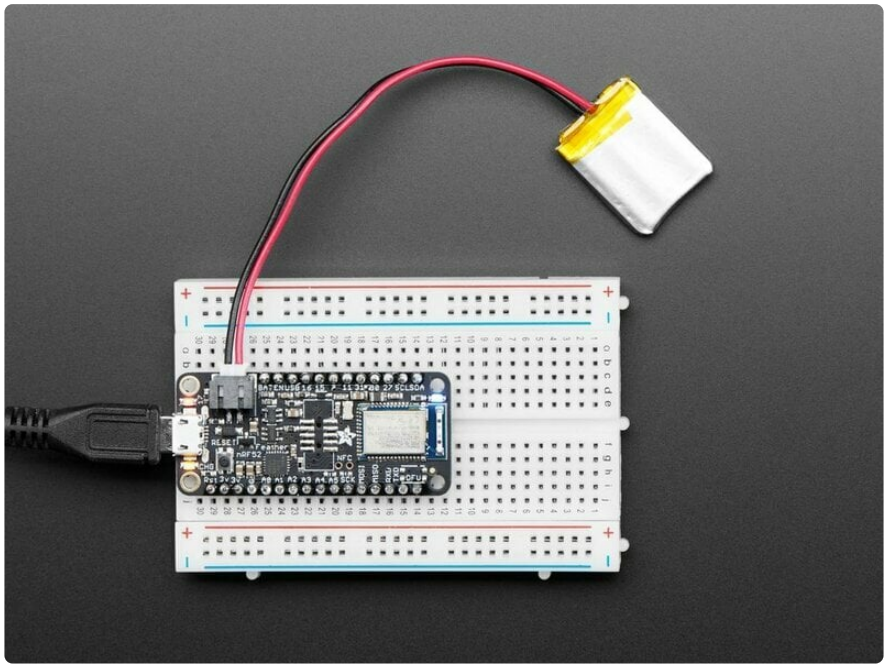
And Solder!

Be sure to solder all pins for reliable electrical contact.

(For tips on soldering, be sure to check out our Guide to Excellent Soldering (https://adafru.it/aTk)).

You're done! Check your solder joints visually and continue onto the next steps

Power Management



Battery + USB Power

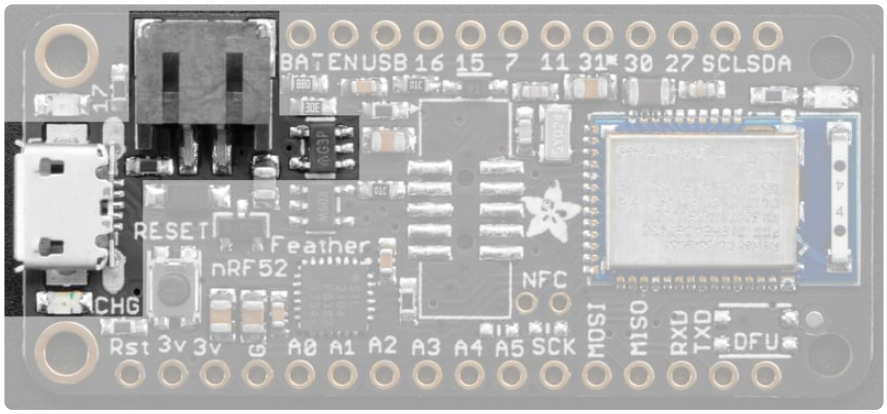
We wanted to make our Feather boards easy to power both when connected to a computer as well as via battery.

There's **two ways to power** a Feather:

1. You can connect with a USB cable (just plug into the jack) and the Feather will regulate the 5V USB down to 3.3V.
2. You can also connect a 4.2/3.7V Lithium Polymer (LiPo/LiPoly) or Lithium Ion (LiIon) battery to the JST jack. This will let the Feather run on a rechargeable battery.

When the USB power is powered, it will automatically switch over to USB for power, as well as start charging the battery (if attached). This happens 'hot-swap' style so you can always keep the LiPoly connected as a 'backup' power that will only get used when USB power is lost.

The JST connector polarity is matched to Adafruit LiPoly batteries. Using wrong polarity batteries can destroy your Feather. Many customers try to save money by purchasing Lipoly batteries from Amazon only to find that they plug them in and the Feather is destroyed!



The above shows the Micro USB jack (left), LiPoly JST jack (top left), as well as the changeover diode (just to the right of the JST jack) and the LiPoly charging circuitry (to the right of the JST jack).

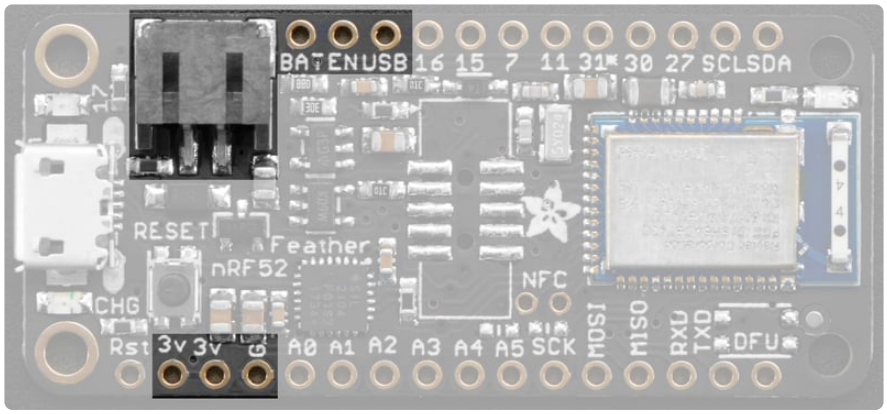
There's also a **CHG** LED next to the USB jack, which will light up while the battery is charging. This LED might also flicker if the battery is not connected, it's normal.

The charge LED is automatically driven by the LiPoly charger circuit. It will try to detect a battery and is expecting one to be attached. If there isn't one it may flicker once in a while when you use power because it's trying to charge a (non-existent) battery. It's not harmful, and it's totally normal!

Power Supplies

You have a lot of power supply options here! We bring out the **BAT** pin, which is tied to the LiPoly JST connector, as well as **USB** which is the +5V from USB if connected. We also have the **3V** pin which has the output from the 3.3V regulator. We use a 500mA peak regulator. While you can get 500mA from it, you can't do it continuously from 5V as it will overheat the regulator.

It's fine for, say, powering an ESP8266 WiFi chip or XBee radio though, since the current draw is 'spikey' & sporadic.



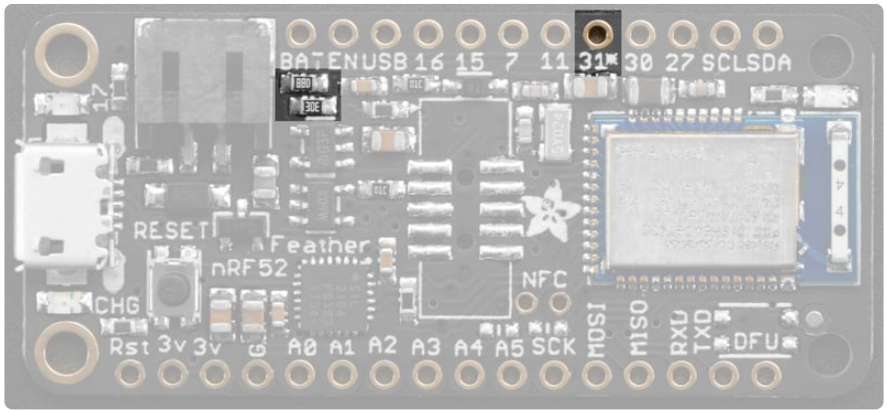
Measuring Battery

If you're running off of a battery, chances are you wanna know what the voltage is at! That way you can tell when the battery needs recharging. LiPoly batteries are 'maxed out' at 4.2V and stick around 3.7V for much of the battery life, then slowly sink down to 3.2V or so before the protection circuitry cuts it off. By measuring the voltage you can quickly tell when you're heading below 3.7V.

To make this easy we stuck a double-100K resistor divider on the **BAT** pin, and connected it to pin **31** (a.k.a analog #6 **A6**). You can read this pin's voltage, then double it, to get the battery voltage.

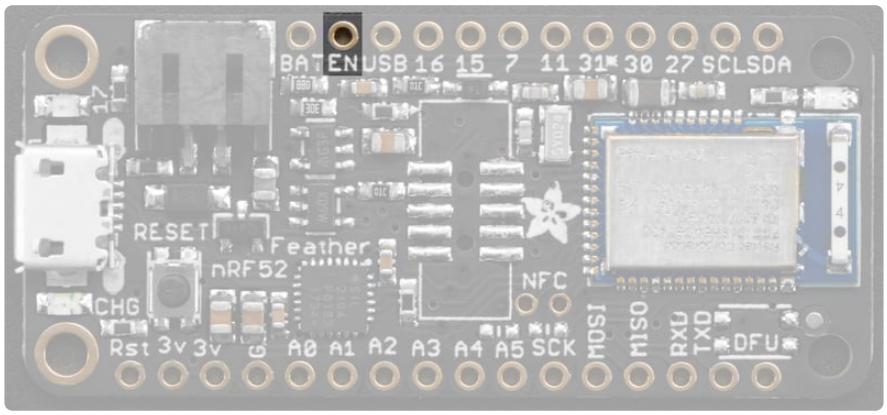
```
#define VBATPIN A6

float measuredvbat = analogRead(VBATPIN);
measuredvbat *= 2;    // we divided by 2, so multiply back
```

```
measuredvbat *= 3.3;  // Multiply by 3.3V, our reference voltage
measuredvbat /= 1024; // convert to voltage
Serial.print("VBat: " ); Serial.println(measuredvbat);
```



ENable pin

If you'd like to turn off the 3.3V regulator, you can do that with the **EN**(able) pin. Simply tie this pin to **Ground** and it will disable the 3V regulator. The **BAT** and **USB** pins will still be powered.



Alternative Power Options

The two primary ways for powering a feather are a 3.7/4.2V LiPo battery plugged into the JST port or a USB power cable.

If you need other ways to power the Feather, here's what we recommend:

- For permanent installations, a 5V 1A USB wall adapter (http://adafru.it/501) will let you plug in a USB cable for reliable power
- For mobile use, where you don't want a LiPoly, use a USB battery pack! (http://adafru.it/1959)

- If you have a higher voltage power supply, use a 5V buck converter (https://adafru.it/DHs) and wire it to a USB cable's 5V and GND input (http://adafru.it/3972)

Here's what you cannot do:

- **Do not use alkaline or NiMH batteries** and connect to the battery port - this will destroy the LiPoly charger and there's no way to disable the charger
- **Do not use 7.4V RC batteries on the battery port** - this will destroy the board

The Feather is not designed for external power supplies - this is a design decision to make the board compact and low cost. It is not recommended, but technically possible:

- **Connect an external 3.3V power supply to the 3V and GND pins.** Not recommended, this may cause unexpected behavior and the **EN** pin will no longer work. Also this doesn't provide power on **BAT** or **USB** and some Feathers/Wings use those pins for high current usages. You may end up damaging your Feather.
- **Connect an external 5V power supply to the USB and GND pins.** Not recommended, this may cause unexpected behavior when plugging in the USB port because you will be back-powering the USB port, which could confuse or damage your computer.

Arduino Support Setup

You can install the Adafruit Bluefruit nRF52 BSP (Board Support Package) in two steps:

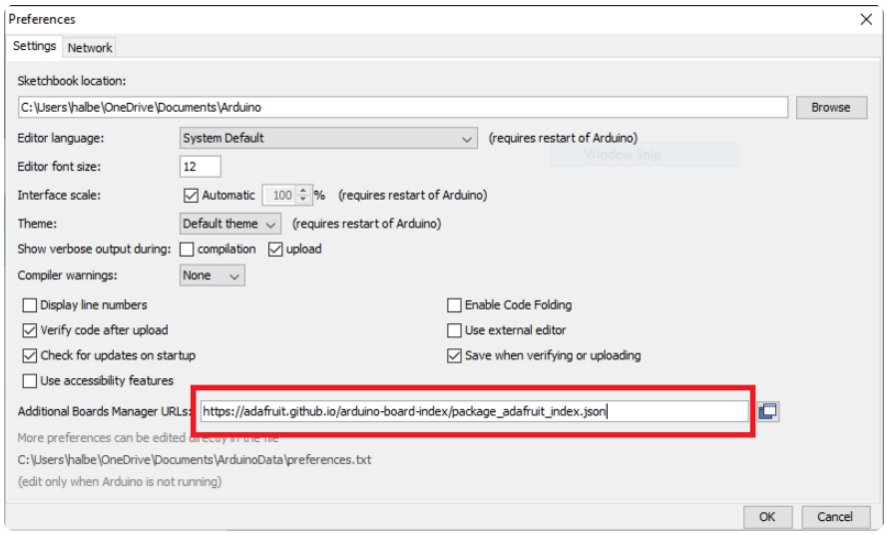
nRF52 support requires at least Arduino IDE version 1.8.15! Please make sure you have an up to date version before proceeding with this guide!

Please consult the FAQ section at the bottom of this page if you run into any problems installing or using this BSP!

1. BSP Installation

Recommended: Installing the BSP via the Board Manager

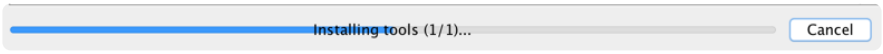
- Download and install the Arduino IDE (https://adafru.it/fvm) (At least **v1.8**)
- Start the Arduino IDE
- Go into Preferences
- Add `https://adafruit.github.io/arduino-board-index/package_adafruit_index.json` as an '**Additional Board Manager URL**' (see image below)



- Restart the Arduino IDE
- Open the **Boards Manager** option from the **Tools -> Board** menu and install '**Adafruit nRF52 by Adafruit**' (see image below)

It will take up to a few minutes to finish installing the cross-compiling toolchain and tools associated with this BSP.

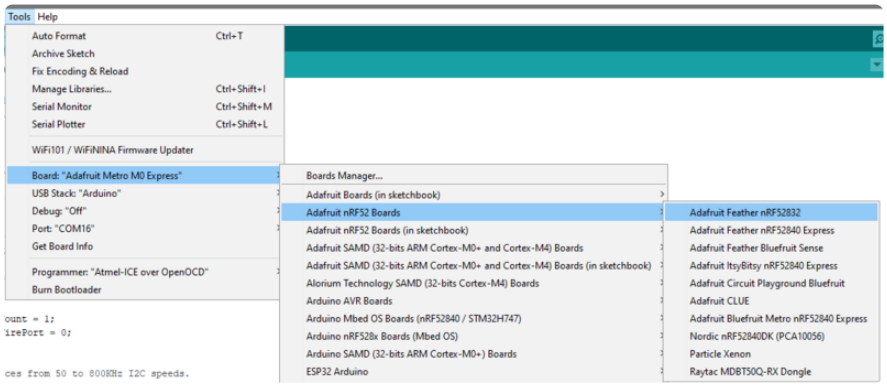
The delay during the installation stage shown in the image below is normal, please be patient and let the installation terminate normally:



Once the BSP is installed, select

- **Adafruit Bluefruit nRF52832 Feather** (for the nRF52 Feather)
- **Adafruit Bluefruit nRF52840 Feather Express** (for the nRF52840 Feather)
- **Adafruit ItsyBitsy nRF52840** (for the Itsy '850)
- **Adafruit Circuit Playground Bluefruit** (for the CPB)
- etc...

from the **Tools -> Board** menu, which will update your system config to use the right compiler and settings for the nRF52:



2. LINUX ONLY: adafruit-nrfutil Tool Installation

adafruit-nrfutil (https://adafru.it/Cau) is a **modified version** of Nordic's nrfutil (https://adafru.it/vaG), which is used to flash boards using the built in serial bootloader. It is originally written for python2, but have been migrated to python3 and renamed to **adafruit-nrfutil** since BSP version 0.8.5.

This step is only required on Linux, pre-built binaries of adafruit-nrfutil for Windows and MacOS are already included in the BSP. That should work out of the box for most setups.

Install python3 if it is not installed in your system already

```
$ sudo apt-get install python3
```

Then run the following command to install the tool from PyPi

```
$ pip3 install --user adafruit-nrfutil
```

Add pip3 installation dir to your **PATH** if it is not added already. Make sure adafruit-nrfutil can be executed in terminal by running

```
$ adafruit-nrfutil version
adafruit-nrfutil version 0.5.3.post12
```

3. Update the bootloader (nRF52832 ONLY)

To keep up with Nordic's SoftDevice advances, you will likely need to update your bootloader if you are using the original nRF52832 based **Bluefruit nRF52 Feather** boards.

Follow this link for instructions on how to do that

This step ISN'T required for the newer nRF52840 Feather Express, which has a different bootloader entirely!

Update the nRF52832 Bootloader

https://adafru.it/Dsx

Advanced Option: Manually Install the BSP via 'git'

If you wish to do any development against the core codebase (generate pull requests, etc.), you can also optionally install the Adafruit nRF52 BSP manually using 'git', as decribed below:

Adafruit nRF52 BSP via git (for core development and PRs only)

1. Install BSP via Board Manager as above to install compiler & tools.
2. Delete the core folder **nrf52** installed by Board Manager in Adruino15, depending on your OS. It could be
   macOS: `~/Library/Arduino15/packages/adafruit/hardware/nrf52`
   Linux: `~/.arduino15/packages/adafruit/hardware/nrf52`
   Windows: `%APPDATA%\Local\Arduino15\packages\adafruit\hardware\nrf52`
3. Go to the sketchbook folder on your command line, which should be one of the following:
   macOS: `~/Documents/Arduino`

Linux: `~/Arduino`
Windows: `~/Documents/Arduino`

4. Create a folder named `hardware/Adafruit`, if it does not exist, and change directories into it.
5. Clone the Adafruit_nRF52_Arduino (https://adafru.it/vaF) repo in the folder described in step 2:
   `git clone --recurse-submodules git@github.com:adafruit/Adafruit_nRF52_Arduino.git`
6. This should result in a final folder name like `~/Documents/Arduino/hardware/Adafruit/Adafruit_nRF52_Arduino` (macOS).
7. Restart the Arduino IDE

Arduino Board Testing

On a Mac, MacOS 10.13 (High Sierra) or newer may be required

Once you have the Bluefruit nRF52 BSP setup on your system, you need to select the appropriate board, which will determine the compiler and expose some new menus options:

1. Select the Board Target

- Go to the **Tools** menu
- Select **Tools** > **Board** > **Adafruit Bluefruit nRF52 Feather** for **nRF52832-based boards**
- Select **Tools** > **Board** > **Adafruit Bluefruit nRF52840 Feather Express** for **nRF52840-based boards**
- Select **Tools** > **Board** > **Adafruit CLUE** for the **Adafruit CLUE**

2. Select the USB CDC Serial Port

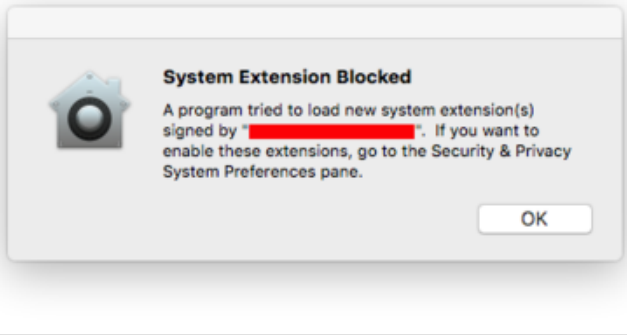
Finally, you need to set the serial port used by Serial Monitor and the serial bootloader:

- Go to **Tools** > **Port** and select the appropriate device

2.1 Download & Install CP2104 Driver (nRF52832)

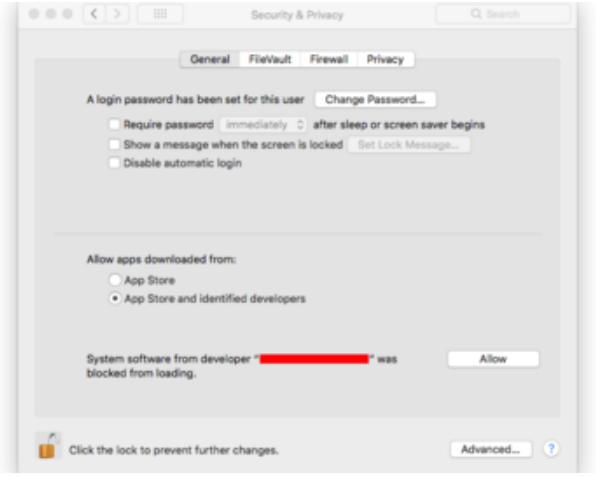
For Feather nRF52832 If you don't see the serial ports device listed, you may need to install the SiLabs CP2104 driver (https://adafru.it/11em) on your system.

On MacOS If you see this dialog message while installing driver



On MacOS If you see this dialog message while installing driver, **System Extension Blocked**

And cannot find the serial port of CP2104, it is highly possible that driver is blocked.



To enable it go to **System Preferences -> Security & Privacy** and click allow if you see Silab in the developer name.

After installing cp210x driver, If feather nRF52832 appear as 2 serial ports on your macos. e.g "/dev/cu.SLAB_USBtoUART" and "/dev/cu.usbserial-1234", the correct port to use is "/dev/cu.usbserial-1234"

2.2 Download & Install Adafruit Driver (nRF52840 Windows)

For Feather nRF52840, If you are using Windows, you will need to follows Windows Driver Installation (https://adafru.it/D0H) to download and install driver.

3. Update the bootloader (nRF52832 Feather Only)

To keep up with Nordic's SoftDevice advances, you will likely need to update your bootloader

Follow this link for instructions on how to do that

This step is only necessary on the nRF52832-based devices, NOT on the newer nRF52840 Feather Express.

Update the Bootloader

https://adafru.it/Dsx

4. Run a Test Sketch

At this point, you should be able to run a test sketch from the **Examples** folder, or just flash the following blinky code from the Arduino IDE:

#include <Adafruit_TinyUSB.h> is required when using with nRF52840 based board for Serial port implementation.

```
#if defined(USE_TINYUSB)
#include <Adafruit_TinyUSB.h> // for Serial
#endif

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

This will blink the red LED beside the USB port on the Feather, or the red LED labeled "LED" by the corner of the USB connector on the CLUE.

Arduino sketch failed to compile with error: ld returned 1 exit status

If the sketch fails to compile and reports an error message like this:

`collect2.exe: error: ld returned 1 exit status`

`exit status 1`

`Compilation error: exit status 1`

Look in the rest of the compile message output (turn on verbose output if needed) and check if the actual error(s) look something like:

`undefined reference to `Adafruit_USBD_CDC::begin(unsigned long)`

`undefined reference to `Adafruit_USBD_CDC::write(unsigned char const*, unsigned int)`

If so, the required `#include` is missing. See the example code above and add the required lines to your sketch.

If Arduino failed to upload sketch to the Feather

If you get this error:

```
Timed out waiting for acknowledgement from device.

Failed to upgrade target. Error is: No data received on serial
port. Not able to proceed.
Traceback (most recent call last):
  File "nordicsemi\__main__.py", line 294, in serial
  File "nordicsemi\dfu\dfu.py", line 235, in dfu_send_images
  File "nordicsemi\dfu\dfu.py", line 203, in _dfu_send_image
  File "nordicsemi\dfu\dfu_transport_serial.py", line 155, in
send_init_packet
  File "nordicsemi\dfu\dfu_transport_serial.py", line 243, in
send_packet
  File "nordicsemi\dfu\dfu_transport_serial.py", line 282, in
get_ack_nr
nordicsemi.exceptions.NordicSemiException: No data received on
serial port. Not able to proceed.
```

This is probably caused by the **bootloader** version mismatched on your Feather and installed BSP. Due to the difference in flash layout (more details (https://adafru.it/Dsy)) and Softdevice API (which is bundled with bootloader), sketch built with selected bootloader can only upload to board having the same version. In short, you need to **upgrade/burn bootloader to match** on your Feather, follow above Update The Bootloader (https://adafru.it/Dsx) guide

It only has to be done once to update your Feather

On Linux I'm getting 'arm-none-eabi-g++: no such file or directory', even though 'arm-none-eabi-g++' exists in the path specified. What should I do?

This is probably caused by a conflict between 32-bit and 64-bit versions of the compiler, libc and the IDE. The compiler uses 32-bit binaries, so you also need to have a 32-bit version of libc installed on your system (details (https://adafru.it/vnE)). Try running the following commands from the command line to resolve this:

```
sudo dpkg --add-architecture i386

sudo apt-get update

sudo apt-get install libc6:i386
```

Using the Bootloader

This page is for information purposes only. Normally the bootloader will work transparently and automatically from the Arduino IDE, requiring no manual intervention on your behalf.

The Bluefruit nRF52 Feather includes a customized version of the Nordic bootloader that enables serial support, over the air (OTA) DFU support, and various fail safe features like factory reset when the FRST pin is grounded at startup.

The bootloader that all Bluefruit nRF52 Feathers ships with allows you to flash user sketches to the nRF52832 using only the CP2104 USB to serial adapter populated on the board.

The bootloader source code can be found here Adafruit_nRF52_Bootloader (https://adafru.it/Cnt)

Forcing Serial Boot Mode

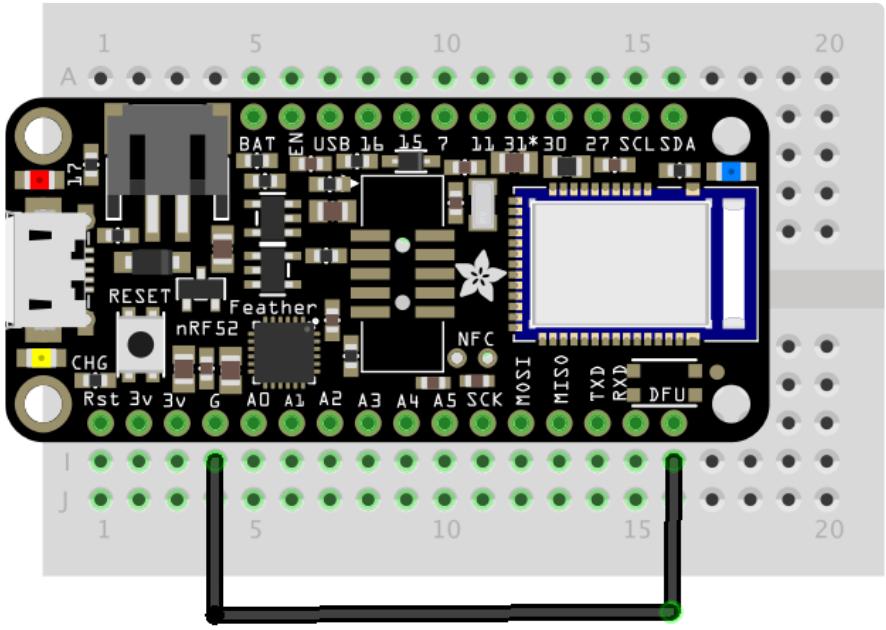
The Bluefruit nRF52 Feather is designed to briefly enter serial bootloader mode for a short delay every time the device comes out of reset, and the DTR line on the CP2104 USB to Serial adapter will trigger a reset every time the Serial Monitor is opened. This means that you can normally flash a user sketch to the nRF52 with no manual intervention on your part at a HW level.

If you need to force the serial bootloader mode, however, you can connect the **DFU** pin to **GND** at startup, which will force you to enter serial bootloader mode and stay in that mode until you reset or power cycle the board.

This can be used to recover bricked boards where a bad user sketch has been uploaded, since you will enter serial bootloader mode without executing the user sketch, and you can flash a new sketch directly from the Arduino IDE.

Forcing the serial bootloader can often be used to recover bricked devices.

nRF52840-based board can also use double reset trick to force into serial bootloader



Factory Reset

The Bluefruit nRF52 Feather has an optional FRST pad on the bottom of the PCB.

If you brick your device, you can solder a wire to the **FRST** pad, connecting it to **GND**. When a GND state is detected at power up the following actions will be performed:

- The user application flash section will be erased
- The user 'App Data' section that stores non volatile config data will be erased

This will cause the device to enter serial bootloader mode at startup, and the user sketch or config data that caused the device to stop responding should be removed.

Be sure to disconnect the pin from GND after a successful factory reset!

Advanced: OTA DFU Bootloader

While this is only recommended for advanced users, you can also force OTA (Over The Air) DFU bootloader mode to enable OTA updates using BLE and Nordic's proprietary update protocol (which is support by both Nordic mobile apps, and out own Bluefruit LE Connect).

To force OTA DFU mode, set both FRST and DFU to GND at startup. Power cycling the board will cause the device to boot up into OTA DFU mode.

This option is not actively support nor recommended by Adafruit, and we are still working on making this as safe as possible for users via our Bluefruit LE Connect application. Use OTA DFU at your own risk knowing you can brick your device and may need a Segger J-Link or similar device to regain control of it!

Updating the Bootloader

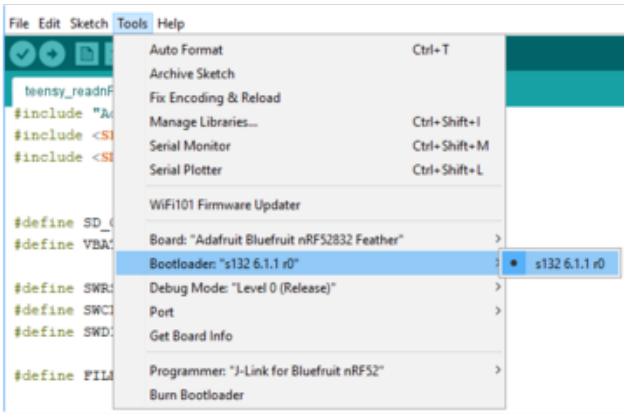
Upgrading the Bootloader is only possible from BSP release 0.8.0 and higher.

Upgrading an Existing Bootloader

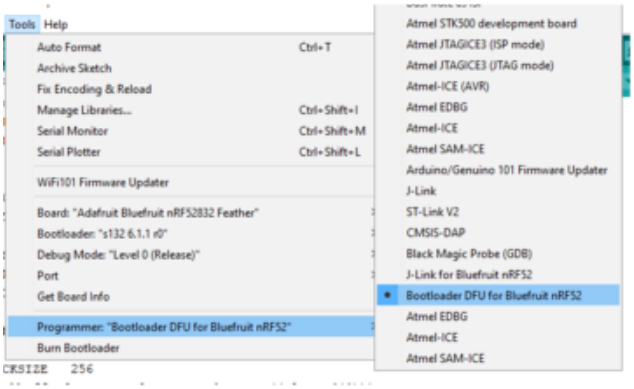
The Adafruit Feather nRF52 Bootloader binary contains not only the DFU code, but also the Bluetooth stack (a.k.a SoftDevice) to make sure they work together reliably. To get the latest and greatest features from the stack such as Bluetooth 5.0 with higher throughput, increased broadcast capacities or larger MTU it is necessary to upgrade Bootloader to get the latest stack.

Therefore latest BSP release could support newer SoftDevice version than one currently exists on your board. Due to the flash memory & API difference between SoftDevice major versions, upgrade your board's bootloader to match one supported by BSP is required to upload compiled sketch. Luckily the Bluefruit nRF52 Bootloader can be upgraded/downgraded without any additional hardware, and we can even do that right in Arduino IDE without at risk of typos or common user errors.

Close the Serial Monitor before you click "Burn Bootloader". Afterwards, you shouldn't close the Arduino IDE, unplug the Feather, launch Serial Monitor etc ... to abort the process. There is a high chance it will brick your device! Do this with care and caution.



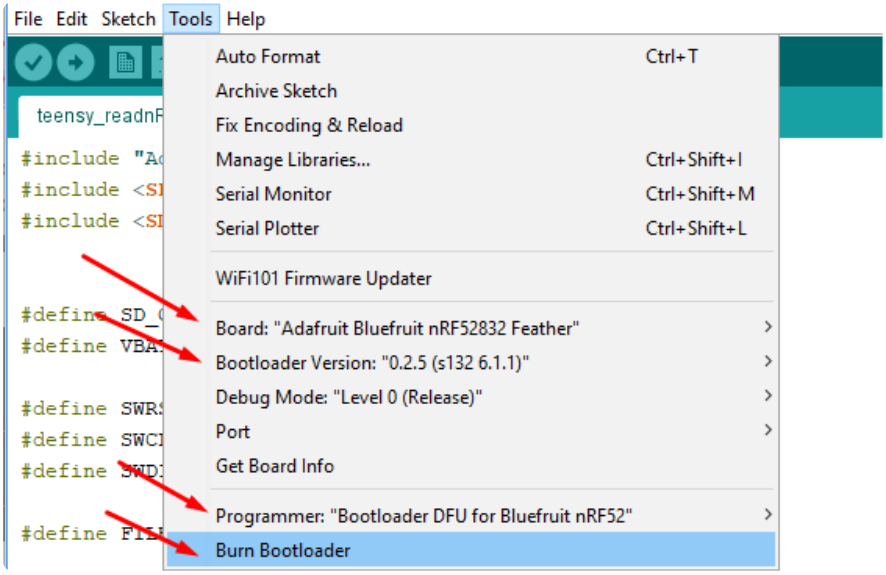
First select the Bootloader version that you want to upgrade under **Tools->Bootloader**



Then select "**Bootloader DFU for Bluefruit nRF52**" for **Tools->Programmer**

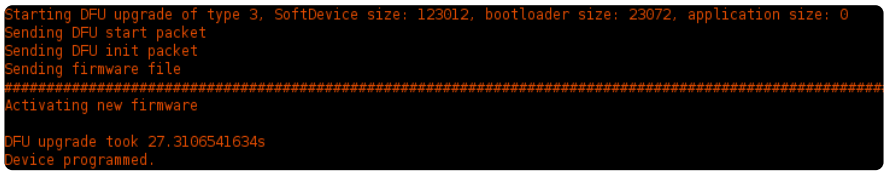
Double check all of the following: Board, Bootloader Version, Programmer...

Select **Tools->Burn Bootloader** to start the upgrade.



After receiving the new Bootloader over the serial connection, the old Bootloader will erase itself! The new bootloader will then be flashed. The process typically takes

30-60 seconds to complete. Make sure you see the "**Device programmed**" in the output log before launching Serial monitor or uploadinga new sketch.



In case you wonder, the command that IDE uses to upgrade bootloader is as follows (bootloader zip file is different for boards and version).

```
$ adafruit-nrfutil --verbose dfu serial --package
feather_nrf52832_bootloader-0.2.8_s132_6.1.1.zip
 -p /dev/ttyACM0 -b 115200 --singlebank --touch 1200
```

Flashing the Bootloader

All Adafruit nRF52 boards chip with the bootloader pre-flashed. This page is provided for information purposes only!

All Bluefruit nRF52 Feather boards and Bluefruit nRF52 modules ship with the serial bootloader pre-flashed, so this page is normally not required when setting your device and system up.

The information provided here is only intended for for rare cases where you may want or need to reflash the bootloader yourself, and have access to the HW required to do so.

1. Updating the Bootloader with a Segger J-Link and Arduino IDE

To burn the bootloader from within the Arduino IDE using a Segger J-Link, you will need the following tools installed on your system and available in the system path:

JLink Drivers and Tools

Download and install the JLink Software and Documentation Pack (https://adafru.it/val) from Segger, which will also install a set of command line tools.

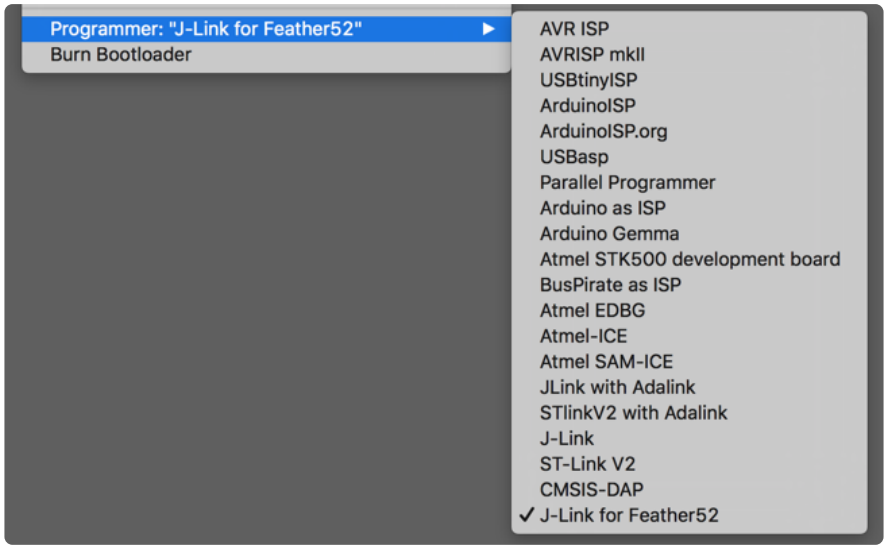
Burning the Bootloader from the Arduino IDE

Once the tools above have been installed and added to your system path, from the Arduino IDE:

- Select `**Tools > Board > Adafruit Bluefruit Feather52**`
- Select `**Tools > Programmer > J-Link for Feather52**`
- Select `**Tools > Burn Bootloader**` with the board and J-Link connected

The appropriate **Programmer** target and **Burn Bootloader** button can be seen below:

You will need a Segger J-Link to flash the bootloader to the nRF52832/ nRF52840 SoC!



2. Manually Burning the Bootloader via nrfjprog

You can also manually burn the bootloader from the command line, using `nrfjprog` from Nordic.

You can either download nRF5x-Command-Line-Tools (https://adafru.it/vaJ) for OSX/ Linux/Win32, or use the version that ships with the BSP in the **tools/nrf5x-command-line-tools** folder.

Run the folllwing commands, updating the path to the .hex file as appropriate:

```
$ nrfjprog --program bootloader_binary.hex --chiperase -f nrf52 --reset
```

You should see something similar to the following output, followed by a fast blinky on the status LED to indicate that you are in DFU/bootloader mode since no user sketch was found after the device reset:

All commands below were run from 'tools/nrf5x-command-line-tools/osx/nrfjprog'

```
$ ./nrfjprog -e -f nrf52
Erasing code and UICR flash areas.
Applying system reset.

$ ./nrfjprog --program  ../../../../bin/bootloader/bootloader_v050_s132_v201.hex -f
nrf52
Parsing hex file.
Reading flash area to program to guarantee it is erased.
Checking that the area to write is not protected.
Programing device.

$ ./nrfjprog --reset -f nrf52
Applying system reset.
Run.
```

OS X Note: You may need to create a symlink in `/usr/local/bin` to the `nrfjprog` tool wherever you have added it. You can run the following command, for example:

$ ln -s $HOME/prog/nordic/nrfjprog/nrfjprog /usr/local/bin/nrfjprog

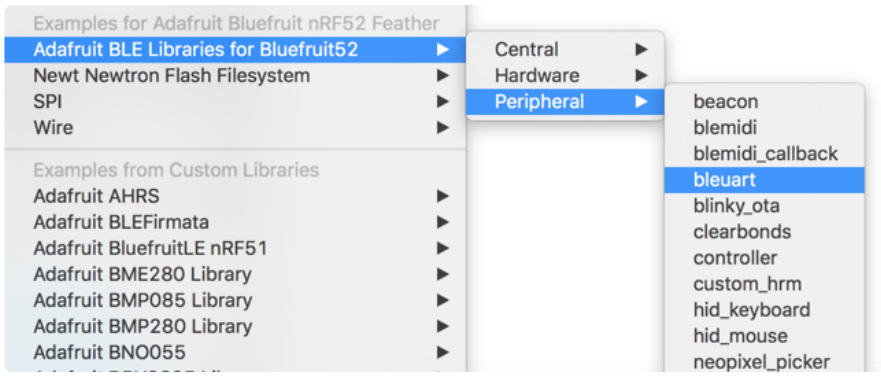
3. Manually Burning the Bootloader via AdaLink

Alternatively, you can use AdaLink (https://adafru.it/fPq) to flash the bootloader with a Segger J-Link:

```
# First erase the device's flash contents
$ adalink nrf52832 -p jlink -w

# Then flash the bootloader and SoftDevice .hex file
$ adalink nrf52832 -p jlink -h feather52_bootloader_v050_s132_v201.hex
```

Arduino BLE Examples

There are numerous examples available for the Bluefruit nRF52/nRF52840 Feathers in the **Examples** menu of the nRF52 BSP, and these are always up to date. You're first stop looking for example code should be there:



Example Source Code

The latest example source code is always available and visible on Github, and the public git repository should be considered the definitive source of example code for this board.

Click here to browse the example source code on Github

https://adafru.it/vaK

Documented Examples

To help explain some common use cases for the nRF52 BLE API, feel free to consult the example documentation in this section of the learning guide:

- **Advertising: Beacon** - Shows how to use the BLEBeacon helper class to configure your Bleufruit nRF52 Feather as a beacon
- **BLE UART: Controller** - Shows how to use the **Controller** utility in our Bluefruit LE Connect apps to send basic data between your peripheral and your phone or tablet.
- **Custom: HRM** - Shows how to defined and work with a custom GATT Service and Characteristic, using the officially adopted Heart Rate Monitor (HRM) service as an example.
- **BLE Pin I/O** (StandardFirmataBLE) Shows how to control Pin I/O of nRF52 with Firmata protocol

For more information on the Arduino nRF52 API, check out this page (https://adafru.it/lla).

Advertising: Beacon

This example shows how you can use the BLEBeacon helper class and advertising API to configure your Bluefruit nRF52 board as a 'Beacon'.

Complete Code

```
/*********************************************************************
 This is an example for our nRF52 based Bluefruit LE modules

 Pick one up today in the adafruit shop!

 Adafruit invests time and resources providing this open source code,
 please support Adafruit and open-source hardware by purchasing
 products from Adafruit!

 MIT license, check LICENSE for more information
 All text above, and the splash screen below must be included in
 any redistribution
*********************************************************************/
#include <bluefruit.h>

// Beacon uses the Manufacturer Specific Data field in the advertising packet,
// which means you must provide a valid Manufacturer ID. Update
// the field below to an appropriate value. For a list of valid IDs see:
// https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers
// - 0x004C is Apple
// - 0x0822 is Adafruit
// - 0x0059 is Nordic
// For testing with this sketch, you can use nRF Beacon app
// - on Android you may need change the MANUFACTURER_ID to Nordic
// - on iOS you may need to change the MANUFACTURER_ID to Apple.
//   You will also need to "Add Other Beacon, then enter Major, Minor that you set
in the sketch
#define MANUFACTURER_ID   0x0059

// "nRF Connect" app can be used to detect beacon
uint8_t beaconUuid[16] = {
  0x01, 0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78,
  0x89, 0x9a, 0xab, 0xbc, 0xcd, 0xde, 0xef, 0xf0
};

// A valid Beacon packet consists of the following information:
// UUID, Major, Minor, RSSI @ 1M
BLEBeacon beacon(beaconUuid, 1, 2, -54);

void setup() {
  Serial.begin(115200);

  // Uncomment to blocking wait for Serial connection
//  while ( !Serial ) delay(10);

  Serial.println("Bluefruit52 Beacon Example");
  Serial.println("--------------------------\n");

  Bluefruit.begin();

  // off Blue LED for lowest power consumption
  Bluefruit.autoConnLed(false);
  Bluefruit.setTxPower(0);    // Check bluefruit.h for supported values
```

```
  // Manufacturer ID is required for Manufacturer Specific Data
  beacon.setManufacturer(MANUFACTURER_ID);

  // Setup the advertising packet
  startAdv();

  Serial.printf("Broadcasting beacon with MANUFACTURER_ID = 0x%04X\n",
MANUFACTURER_ID);
  Serial.println("open your beacon app to test such as: nRF Beacon");
  Serial.println("- on Android you may need to change the MANUFACTURER_ID to
0x0059");
  Serial.println("- on iOS you may need to change the MANUFACTURER_ID to 0x004C");

  // Suspend Loop() to save power, since we didn't have any code there
  suspendLoop();
}

void startAdv(void)
{
  // Advertising packet
  // Set the beacon payload using the BLEBeacon class populated
  // earlier in this example
  Bluefruit.Advertising.setBeacon(beacon);

  // Secondary Scan Response packet (optional)
  // Since there is no room for 'Name' in Advertising packet
  Bluefruit.ScanResponse.addName();

  /* Start Advertising
   * - Enable auto advertising if disconnected
   * - Timeout for fast mode is 30 seconds
   * - Start(timeout) with timeout = 0 will advertise forever (until connected)
   *
   * Apple Beacon specs
   * - Type: Non-connectable, scannable, undirected
   * - Fixed interval: 100 ms -> fast = slow = 100 ms
   */

Bluefruit.Advertising.setType(BLE_GAP_ADV_TYPE_NONCONNECTABLE_SCANNABLE_UNDIRECTED);
  Bluefruit.Advertising.restartOnDisconnect(true);
  Bluefruit.Advertising.setInterval(160, 160);    // in unit of 0.625 ms
  Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
  Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising
after n seconds
}

void loop() {
  // loop is already suspended, CPU will not run loop() at all
}
```

```
/*
 * The MIT License (MIT)
 *
 * Copyright (c) 2019, hathach (tinyusb.org) for Adafruit
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
```

```
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
#ifndef BLUEFRUIT_H_
#define BLUEFRUIT_H_

#include <Arduino.h>
#include "bluefruit_common.h"

#define CFG_ADV_BLINKY_INTERVAL   500

/* Note changing these parameters will affect APP_RAM_BASE
 * --> need to update RAM region in linker file
 * - BLE_GATT_ATT_MTU_MAX from 23 (default) to 247
 */
#define BLE_GATT_ATT_MTU_MAX      247
#define BLE_MAX_CONNECTION        20 // SD support up to 20 connections

// Allocate more memory for GATT table for 840
#ifdef NRF52840_XXAA
  #define CFG_SD_ATTR_TABLE_SIZE    0x1000
#else
  #define CFG_SD_ATTR_TABLE_SIZE    0xC00
#endif

#include "BLEUuid.h"
#include "BLEAdvertising.h"
#include "BLECharacteristic.h"
#include "BLEService.h"
#include "BLEScanner.h"
#include "BLEPeriph.h"
#include "BLECentral.h"
#include "BLEClientCharacteristic.h"
#include "BLEClientService.h"
#include "BLEDiscovery.h"
#include "BLEConnection.h"
#include "BLEGatt.h"
#include "BLESecurity.h"

// Services
#include "services/BLEDis.h"
#include "services/BLEDfu.h"
#include "services/BLEUart.h"
#include "services/BLEBas.h"
#include "services/BLEIas.h"
#include "services/BLEBeacon.h"
#include "services/BLEHidGeneric.h"
#include "services/BLEHidAdafruit.h"
#include "services/BLEHidGamepad.h"
#include "services/BLEMidi.h"
#include "services/EddyStone.h"

#include "clients/BLEAncs.h"
#include "clients/BLEClientUart.h"
#include "clients/BLEClientDis.h"
#include "clients/BLEClientCts.h"
#include "clients/BLEClientHidAdafruit.h"
#include "clients/BLEClientBas.h"
#include "clients/BLEClientIas.h"

#include "utility/AdaCallback.h"
#include "utility/bonding.h"

enum
{
  BANDWIDTH_AUTO = 0,
```

```
  BANDWIDTH_LOW,
  BANDWIDTH_NORMAL,
  BANDWIDTH_HIGH,
  BANDWIDTH_MAX,
};

enum
{
  CONN_CFG_PERIPHERAL = 1,
  CONN_CFG_CENTRAL = 2,
};

extern "C"
{
  void SD_EVT_IRQHandler(void);
}

class AdafruitBluefruit
{
  public:
    typedef void (*event_cb_t) (ble_evt_t* evt);
    typedef void (*rssi_cb_t) (uint16_t conn_hdl, int8_t rssi);

    AdafruitBluefruit(void);

    /*------------------------------------------------------------------*/
    /* Lower Level Classes (Bluefruit.Advertising.*, etc.)
     *------------------------------------------------------------------*/
    BLEPeriph          Periph;
    BLECentral         Central;
    BLESecurity        Security;
    BLEGatt            Gatt;

    BLEAdvertising     Advertising;
    BLEAdvertisingData ScanResponse;
    BLEScanner         Scanner;
    BLEDiscovery       Discovery;

    /*------------------------------------------------------------------*/
    /* SoftDevice Configure Functions, must call before begin().
     * These function affect the SRAM consumed by SoftDevice.
     *------------------------------------------------------------------*/
    void configServiceChanged (bool     changed);
    void configUuid128Count   (uint8_t  uuid128_max);
    void configAttrTableSize  (uint32_t attr_table_size);

    // Configure Bandwidth for connections
    void configPrphConn        (uint16_t mtu_max, uint16_t event_len, uint8_t
hvn_qsize, uint8_t wrcmd_qsize);
    void configCentralConn     (uint16_t mtu_max, uint16_t event_len, uint8_t
hvn_qsize, uint8_t wrcmd_qsize);
    void configPrphBandwidth   (uint8_t bw);
    void configCentralBandwidth(uint8_t bw);

    bool begin(uint8_t prph_count = 1, uint8_t central_count = 0);

    /*------------------------------------------------------------------*/
    /* General Functions
     *------------------------------------------------------------------*/
    ble_gap_addr_t  getAddr(void);
    uint8_t         getAddr(uint8_t mac[6]);
    bool            setAddr(ble_gap_addr_t* gap_addr);

    void     setName           (const char* str);
    uint8_t  getName           (char* name, uint16_t bufsize);

    // Supported tx_power values depending on mcu:
    // - nRF52832: -40dBm, -20dBm, -16dBm, -12dBm, -8dBm, -4dBm, 0dBm, +3dBm and
+4dBm.
```

```
    // - nRF52840: -40dBm, -20dBm, -16dBm, -12dBm, -8dBm, -4dBm, 0dBm, +2dBm,
+3dBm, +4dBm, +5dBm, +6dBm, +7dBm and +8dBm.
    bool     setTxPower         (int8_t power);
    int8_t   getTxPower         (void);

    bool     setAppearance      (uint16_t appear);
    uint16_t getAppearance      (void);

    void     autoConnLed        (bool enabled);
    void     setConnLedInterval (uint32_t ms);

    /*------------------------------------------------------------------*/
    /* GAP, Connections and Bonding
     *------------------------------------------------------------------*/
    uint8_t  connected         (void); // Number of connected
    bool     connected         (uint16_t conn_hdl);

    uint8_t  getConnectedHandles(uint16_t* hdl_list, uint8_t max_count);
    uint16_t connHandle        (void);

    // Alias to BLEConnection API()
    bool     disconnect        (uint16_t conn_hdl);

    uint16_t getMaxMtu(uint8_t role);

    BLEConnection* Connection(uint16_t conn_hdl);

#ifdef ANT_LICENSE_KEY
    /*------------------------------------------------------------------*
     * Optional semaphore for additional event handlers for SD event.
     * It can be used for handling non-BLE  SD events
     *------------------------------------------------------------------*/
    void setMultiprotocolSemaphore(SemaphoreHandle_t mprot_event_semaphore)
    {
      _mprot_event_sem= mprot_event_semaphore;
    }
#endif

    /*------------------------------------------------------------------*/
    /* Callbacks
     *------------------------------------------------------------------*/
    void setRssiCallback(rssi_cb_t fp);
    void setEventCallback(event_cb_t fp);

    /*------------------------------------------------------------------*/
    /* INTERNAL USAGE ONLY
     * Although declare as public, it is meant to be invoked by internal
     * code. User should not call these directly
     *------------------------------------------------------------------*/
    void _startConnLed (void);
    void _stopConnLed  (void);
    void _setConnLed   (bool on_off);

    void printInfo(void);

  private:
    /*------------- SoftDevice Configuration -------------*/
    struct {
      uint32_t attr_table_size;
      uint8_t  service_changed;
      uint8_t  uuid128_max;

      // Bandwidth configuration
      struct {
        uint16_t  mtu_max;
        uint16_t  event_len;
        uint8_t   hvn_qsize;
        uint8_t   wrcmd_qsize;
      }prph, central;
```

```
    }_sd_cfg;

    uint8_t _prph_count;
    uint8_t _central_count;

    int8_t _tx_power;

    ble_gap_sec_params_t _sec_param;

    SemaphoreHandle_t _ble_event_sem;
    SemaphoreHandle_t _soc_event_sem;
#ifdef ANT_LICENSE_KEY
    /* Optional semaphore for additional event handlers for SD event.
     * It can be used for handling non-BLE  SD events
     */
    SemaphoreHandle_t _mprot_event_sem;
#endif

    // Auto LED Blinky
    TimerHandle_t _led_blink_th;
    bool _led_conn;

    uint16_t _conn_hdl;

    BLEConnection* _connection[BLE_MAX_CONNECTION];

    //------------- Callbacks -------------//
    rssi_cb_t _rssi_cb;
    event_cb_t _event_cb;

    /*------------------------------------------------------------------*/
    /* INTERNAL USAGE ONLY
     *------------------------------------------------------------------*/
    void _ble_handler(ble_evt_t* evt);

    friend void SD_EVT_IRQHandler(void);
    friend void adafruit_ble_task(void* arg);
    friend void adafruit_soc_task(void* arg);
    friend class BLECentral;
};

extern AdafruitBluefruit Bluefruit;

#endif
```

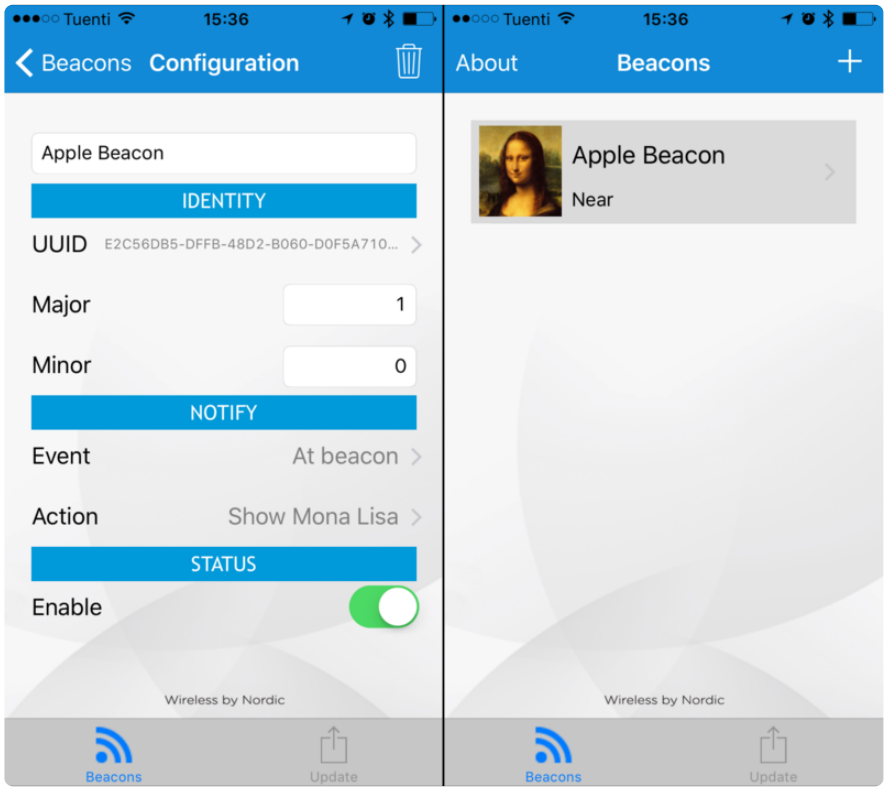
Output

You can use the nRF Beacons application from Nordic Semiconductors to test this sketch:

- nRF Beacons for iOS (https://adafru.it/vaC)
- nRF Beacons for Android (https://adafru.it/vaD)

Make sure that you set the UUID, Major and Minor values to match the sketch above, and then run the sketch at the same time as the nRF Beacons application.

With the default setup you should see a Mona Lisa icon when the beacon is detected. If you don't see this, double check the UUID, Major and Minor values to be sure they match exactly.



BLE UART: Controller

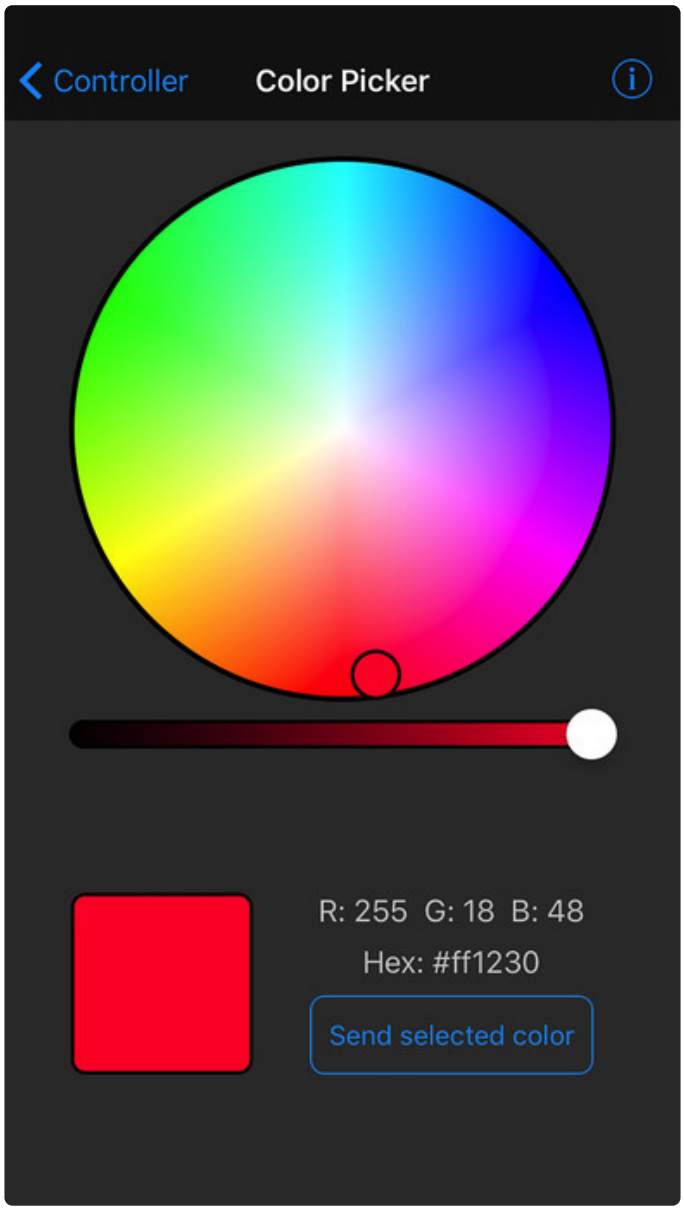
This examples shows you you can use the BLEUart helper class and the Bluefruit LE Connect applications to send based keypad and sensor data to your nRF52.

Setup

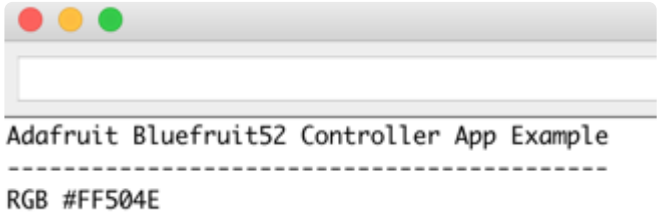
In order to use this sketch, you will need to open Bluefruit LE Connect on your mobile device using our free iOS (https://adafru.it/f4H), Android (https://adafru.it/f4G) or OS X (https://adafru.it/o9F) applications.

- Load the **Controller** example sketch (https://adafru.it/vaN) in the Arduino IDE
- Compile the sketch and flash it to your nRF52 based Feather
- Once you are done uploading, open the **Serial Monitor** (Tools > Serial Monitor)
- Open the **Bluefruit LE Connect** application on your mobile device
- Connect to the appropriate target (probably '**Bluefruit52**')
- Once connected switch to the **Controller** application inside the app

- Enable an appropriate control surface. The **Color Picker** control surface is shown below, for example (screen shot taken from the iOS application):



As you change the color (or as other data becomes available) you should receive the data on the nRF52, and see it in the Serial Monitor output:



Complete Code

```
/*********************************************************************
 This is an example for our nRF52 based Bluefruit LE modules

 Pick one up today in the adafruit shop!

 Adafruit invests time and resources providing this open source code,
 please support Adafruit and open-source hardware by purchasing
 products from Adafruit!

 MIT license, check LICENSE for more information
 All text above, and the splash screen below must be included in
 any redistribution
*********************************************************************/

#include <bluefruit.h>

// OTA DFU service
BLEDfu bledfu;

// Uart over BLE service
BLEUart bleuart;

// Function prototypes for packetparser.cpp
uint8_t readPacket (BLEUart *ble_uart, uint16_t timeout);
float   parsefloat (uint8_t *buffer);
void    printHex   (const uint8_t * data, const uint32_t numBytes);

// Packet buffer
extern uint8_t packetbuffer[];

void setup(void)
{
  Serial.begin(115200);
  while ( !Serial ) delay(10);   // for nrf52840 with native usb

  Serial.println(F("Adafruit Bluefruit52 Controller App Example"));
  Serial.println(F("-------------------------------------------"));

  Bluefruit.begin();
  Bluefruit.setTxPower(4);    // Check bluefruit.h for supported values

  // To be consistent OTA DFU should be added first if it exists
  bledfu.begin();

  // Configure and start the BLE Uart service
  bleuart.begin();

  // Set up and start advertising
  startAdv();

  Serial.println(F("Please use Adafruit Bluefruit LE app to connect in Controller
mode"));
  Serial.println(F("Then activate/use the sensors, color picker, game controller,
etc!"));
  Serial.println();
}

void startAdv(void)
{
  // Advertising packet
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
```

```
  Bluefruit.Advertising.addTxPower();

  // Include the BLE UART (AKA 'NUS') 128-bit UUID
  Bluefruit.Advertising.addService(bleuart);

  // Secondary Scan Response packet (optional)
  // Since there is no room for 'Name' in Advertising packet
  Bluefruit.ScanResponse.addName();

  /* Start Advertising
   * - Enable auto advertising if disconnected
   * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
   * - Timeout for fast mode is 30 seconds
   * - Start(timeout) with timeout = 0 will advertise forever (until connected)
   *
   * For recommended advertising interval
   * https://developer.apple.com/library/content/qa/qa1931/_index.html
   */
  Bluefruit.Advertising.restartOnDisconnect(true);
  Bluefruit.Advertising.setInterval(32, 244);    // in unit of 0.625 ms
  Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
  Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising
after n seconds
}

/**************************************************************************/
/*!
    @brief  Constantly poll for new command or response data
*/
/**************************************************************************/
void loop(void)
{
  // Wait for new data to arrive
  uint8_t len = readPacket(&bleuart, 500);
  if (len == 0) return;

  // Got a packet!
  // printHex(packetbuffer, len);

  // Color
  if (packetbuffer[1] == 'C') {
    uint8_t red = packetbuffer[2];
    uint8_t green = packetbuffer[3];
    uint8_t blue = packetbuffer[4];
    Serial.print ("RGB #");
    if (red < 0x10) Serial.print("0");
    Serial.print(red, HEX);
    if (green < 0x10) Serial.print("0");
    Serial.print(green, HEX);
    if (blue < 0x10) Serial.print("0");
    Serial.println(blue, HEX);
  }

  // Buttons
  if (packetbuffer[1] == 'B') {
    uint8_t buttnum = packetbuffer[2] - '0';
    boolean pressed = packetbuffer[3] - '0';
    Serial.print ("Button "); Serial.print(buttnum);
    if (pressed) {
      Serial.println(" pressed");
    } else {
      Serial.println(" released");
    }
  }

  // GPS Location
  if (packetbuffer[1] == 'L') {
    float lat, lon, alt;
    lat = parsefloat(packetbuffer+2);
```

```
    lon = parsefloat(packetbuffer+6);
    alt = parsefloat(packetbuffer+10);
    Serial.print("GPS Location\t");
    Serial.print("Lat: "); Serial.print(lat, 4); // 4 digits of precision!
    Serial.print('\t');
    Serial.print("Lon: "); Serial.print(lon, 4); // 4 digits of precision!
    Serial.print('\t');
    Serial.print(alt, 4); Serial.println(" meters");
  }

  // Accelerometer
  if (packetbuffer[1] == 'A') {
    float x, y, z;
    x = parsefloat(packetbuffer+2);
    y = parsefloat(packetbuffer+6);
    z = parsefloat(packetbuffer+10);
    Serial.print("Accel\t");
    Serial.print(x); Serial.print('\t');
    Serial.print(y); Serial.print('\t');
    Serial.print(z); Serial.println();
  }

  // Magnetometer
  if (packetbuffer[1] == 'M') {
    float x, y, z;
    x = parsefloat(packetbuffer+2);
    y = parsefloat(packetbuffer+6);
    z = parsefloat(packetbuffer+10);
    Serial.print("Mag\t");
    Serial.print(x); Serial.print('\t');
    Serial.print(y); Serial.print('\t');
    Serial.print(z); Serial.println();
  }

  // Gyroscope
  if (packetbuffer[1] == 'G') {
    float x, y, z;
    x = parsefloat(packetbuffer+2);
    y = parsefloat(packetbuffer+6);
    z = parsefloat(packetbuffer+10);
    Serial.print("Gyro\t");
    Serial.print(x); Serial.print('\t');
    Serial.print(y); Serial.print('\t');
    Serial.print(z); Serial.println();
  }

  // Quaternions
  if (packetbuffer[1] == 'Q') {
    float x, y, z, w;
    x = parsefloat(packetbuffer+2);
    y = parsefloat(packetbuffer+6);
    z = parsefloat(packetbuffer+10);
    w = parsefloat(packetbuffer+14);
    Serial.print("Quat\t");
    Serial.print(x); Serial.print('\t');
    Serial.print(y); Serial.print('\t');
    Serial.print(z); Serial.print('\t');
    Serial.print(w); Serial.println();
  }
}
```

You will also need the following helper class in a file called **packetParser.cpp**:

```
#include <string.h>
#include <Arduino.h>
#include <bluefruit.h>
```

```
#define PACKET_ACC_LEN                  (15)
#define PACKET_GYRO_LEN                 (15)
#define PACKET_MAG_LEN                  (15)
#define PACKET_QUAT_LEN                 (19)
#define PACKET_BUTTON_LEN               (5)
#define PACKET_COLOR_LEN                (6)
#define PACKET_LOCATION_LEN             (15)

//    READ_BUFSIZE            Size of the read buffer for incoming packets
#define READ_BUFSIZE                    (20)


/* Buffer to hold incoming characters */
uint8_t packetbuffer[READ_BUFSIZE+1];

/**************************************************************************/
/*!
    @brief  Casts the four bytes at the specified address to a float
*/
/**************************************************************************/
float parsefloat(uint8_t *buffer)
{
  float f;
  memcpy(&f, buffer, 4);
  return f;
}

/**************************************************************************/
/*!
    @brief  Prints a hexadecimal value in plain characters
    @param  data      Pointer to the byte data
    @param  numBytes  Data length in bytes
*/
/**************************************************************************/
void printHex(const uint8_t * data, const uint32_t numBytes)
{
  uint32_t szPos;
  for (szPos=0; szPos < numBytes; szPos++)
  {
    Serial.print(F("0x"));
    // Append leading 0 for small values
    if (data[szPos] <= 0xF)
    {
      Serial.print(F("0"));
      Serial.print(data[szPos] & 0xf, HEX);
    }
    else
    {
      Serial.print(data[szPos] & 0xff, HEX);
    }
    // Add a trailing space if appropriate
    if ((numBytes > 1) && (szPos != numBytes - 1))
    {
      Serial.print(F(" "));
    }
  }
  Serial.println();
}

/**************************************************************************/
/*!
    @brief  Waits for incoming data and parses it
*/
/**************************************************************************/
uint8_t readPacket(BLEUart *ble_uart, uint16_t timeout)
{
  uint16_t origtimeout = timeout, replyidx = 0;
```

```
  memset(packetbuffer, 0, READ_BUFSIZE);

  while (timeout--) {
    if (replyidx >= 20) break;
    if ((packetbuffer[1] == 'A') && (replyidx == PACKET_ACC_LEN))
      break;
    if ((packetbuffer[1] == 'G') && (replyidx == PACKET_GYRO_LEN))
      break;
    if ((packetbuffer[1] == 'M') && (replyidx == PACKET_MAG_LEN))
      break;
    if ((packetbuffer[1] == 'Q') && (replyidx == PACKET_QUAT_LEN))
      break;
    if ((packetbuffer[1] == 'B') && (replyidx == PACKET_BUTTON_LEN))
      break;
    if ((packetbuffer[1] == 'C') && (replyidx == PACKET_COLOR_LEN))
      break;
    if ((packetbuffer[1] == 'L') && (replyidx == PACKET_LOCATION_LEN))
      break;

    while (ble_uart->available()) {
      char c =  ble_uart->read();
      if (c == '!') {
        replyidx = 0;
      }
      packetbuffer[replyidx] = c;
      replyidx++;
      timeout = origtimeout;
    }

    if (timeout == 0) break;
    delay(1);
  }

  packetbuffer[replyidx] = 0;  // null term

  if (!replyidx)  // no data or timeout
    return 0;
  if (packetbuffer[0] != '!')  // doesn't start with '!' packet beginning
    return 0;

  // check checksum!
  uint8_t xsum = 0;
  uint8_t checksum = packetbuffer[replyidx-1];

  for (uint8_t i=0; i<replyidx-1; i++) {
    xsum += packetbuffer[i];
  }
  xsum = ~xsum;

  // Throw an error message if the checksum's don't match
  if (xsum != checksum)
  {
    Serial.print("Checksum mismatch in packet : ");
    printHex(packetbuffer, replyidx+1);
    return 0;
  }

  // checksum passed!
  return replyidx;
}
```

Custom: HRM

The BLEService and BLECharacteristic classes can be used to implement any custom or officially adopted BLE service of characteristic using a set of basic properties and callback handlers.

The example below shows how to use these classes to implement the Heart Rate Monitor (https://adafru.it/vaO) service, as defined by the Bluetooth SIG.

HRM Service Definition

UUID: 0x180D (https://adafru.it/vaO)

Characteristic Name	UUID	Requirement	Properties
Heart Rate Measurement	0x2A37	Mandatory	Notify
Body Sensor Location	0x2A38	Optional	Read
Heart Rate Control Point	0x2A39	Conditional	Write

Only the first characteristic is mandatory, but we will also implement the optional **Body Sensor Location** characteristic. Heart Rate Control Point won't be used in this example to keep things simple.

Implementing the HRM Service and Characteristics

The core service and the first two characteristics can be implemented with the following code:

First, define the BLEService and BLECharacteristic variables that will be used in your project:

```
/* HRM Service Definitions
 * Heart Rate Monitor Service:  0x180D
 * Heart Rate Measurement Char: 0x2A37
 * Body Sensor Location Char:   0x2A38
 */
BLEService        hrms = BLEService(UUID16_SVC_HEART_RATE);
BLECharacteristic hrmc = BLECharacteristic(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLECharacteristic bslc = BLECharacteristic(UUID16_CHR_BODY_SENSOR_LOCATION);
```

Then you need to 'populate' those variables with appropriate values. For simplicity sake, you can define a custom function for your service where all of the code is placed, and then just call this function once in the 'setup' function:

```
void setupHRM(void)
{
  // Configure the Heart Rate Monitor service
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.service.heart_rate.xml
  // Supported Characteristics:
  // Name                         UUID    Requirement Properties
  // ---------------------------- ------  ----------- ----------
  // Heart Rate Measurement       0x2A37  Mandatory   Notify
  // Body Sensor Location         0x2A38  Optional    Read
  // Heart Rate Control Point     0x2A39  Conditional Write       &lt;-- Not used
here
  hrms.begin();

  // Note: You must call .begin() on the BLEService before calling .begin() on
  // any characteristic(s) within that service definition.. Calling .begin() on
  // a BLECharacteristic will cause it to be added to the last BLEService that
  // was 'begin()'ed!

  // Configure the Heart Rate Measurement characteristic
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
  // Permission = Notify
  // Min Len    = 1
  // Max Len    = 8
  //    B0      = UINT8  - Flag (MANDATORY)
  //      b5:7  = Reserved
  //      b4    = RR-Internal (0 = Not present, 1 = Present)
  //      b3    = Energy expended status (0 = Not present, 1 = Present)
  //      b1:2  = Sensor contact status (0+1 = Not supported, 2 = Supported but
contact not detected, 3 = Supported and detected)
  //      b0    = Value format (0 = UINT8, 1 = UINT16)
  //    B1      = UINT8  - 8-bit heart rate measurement value in BPM
  //    B2:3    = UINT16 - 16-bit heart rate measurement value in BPM
  //    B4:5    = UINT16 - Energy expended in joules
  //    B6:7    = UINT16 - RR Internal (1/1024 second resolution)
  hrmc.setProperties(CHR_PROPS_NOTIFY);
  hrmc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  hrmc.setFixedLen(2);
  hrmc.setCccdWriteCallback(cccd_callback);  // Optionally capture CCCD updates
  hrmc.begin();
  uint8_t hrmdata[2] = { 0b00000110, 0x40 }; // Set the characteristic to use 8-bit
values, with the sensor connected and detected
  hrmc.notify(hrmdata, 2);                   // Use .notify instead of .write!

  // Configure the Body Sensor Location characteristic
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
  // Permission = Read
  // Min Len    = 1
  // Max Len    = 1
  //    B0      = UINT8 - Body Sensor Location
  //      0     = Other
  //      1     = Chest
  //      2     = Wrist
  //      3     = Finger
  //      4     = Hand
  //      5     = Ear Lobe
  //      6     = Foot
  //      7:255 = Reserved
  bslc.setProperties(CHR_PROPS_READ);
  bslc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
```

```
  bslc.setFixedLen(1);
  bslc.begin();
  bslc.write8(2);    // Set the characteristic to 'Wrist' (2)
}
```

Service + Characteristic Setup Code Analysis

1. The first thing to do is to call **.begin()** on the BLEService (**hrms** above). Since the UUID is set in the object declaration at the top of the sketch, there is normally nothing else to do with the BLEService instance.

You MUST call .begin() on the BLEService before adding any BLECharacteristics. Any BLECharacteristic will automatically be added to the last BLEService that was `begin()'ed!

2. Next, you can configure the **Heart Rate Measurement** characteristic (**hrmc** above). The values that you set for this will depend on the characteristic definition, but for convenience sake we've documented the key information in the comments in the code above.

- '`hrmc.setProperties(CHR_PROPS_NOTIFY);`' - This sets the PROPERTIES value for the characteristic, which determines how the characteristic can be accessed. In this case, the Bluetooth SIG has defined the characteristic as **Notify**, which means that the peripheral will receive a request ('notification') from the Central when the Central wants to receive data using this characteristic.
- `` `hrmc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);` `` - This sets the security for the characteristic, and should normally be set to the values used in this example.
- `` `hrmc.setFixedLen(2);` `` - This tells the Bluetooth stack how many bytes the characteristic contains (normally a value between 1 and 20). In this case, we will use a fixed size of two bytes, so we call **.setFixedLen**. If the characteristic has a variable length, you would need to set the max size via **.setMaxLen**.
- '`hrmc.setCccdWriteCallback(cccd_callback);`' - This optional code sets the callback that will be fired when the CCCD record is updated by the central. This is relevant because the characteristic is setup with the NOTIFY property. When the Central sets to 'Notify' bit, it will write to the CCCD record, and you can capture this write even in the CCCD callback and turn the sensor on, for example, allowing you to save power by only turning the sensor on (and back off) when it is or isn't actually being used. For the implementation of the CCCD callback handler, see the full sample code at the bottom of this page.

- '`hrmc.begin();`' Once all of the properties have been set, you must call **.begin()** which will add the characteristic definition to the last BLEService that was '.begin()ed'.

3. Optionally set an initial value for the characteristic(s), such as the following code that populates 'hrmc' with a correct values, indicating that we are providing 8-bit heart rate monitor values, that the Body Sensor Location characteristic is present, and setting the first heart rate value to 0x04:

Note that we use .notify() in the example above instead of .write(), since this characteristic is setup with the NOTIFY property which needs to be handled in a slightly different manner than other characteristics.

```
// Set the characteristic to use 8-bit values, with the sensor connected and
detected
uint8_t hrmdata[2] = { 0b00000110, 0x40 };

// Use .notify instead of .write!
hrmc.notify(hrmdata, 2);
```

The CCCD callback handler has the following signature:

```
void cccd_callback(uint16_t conn_hdl, BLECharacteristic* chr, uint16_t cccd_value)
{
    // Display the raw request packet
    Serial.print("CCCD Updated: ");
    //Serial.printBuffer(request-&gt;data, request-&gt;len);
    Serial.print(cccd_value);
    Serial.println("");

    // Check the characteristic this CCCD update is associated with in case
    // this handler is used for multiple CCCD records.
    if (chr-&gt;uuid == htmc.uuid) {
        if (chr-&gt;indicateEnabled(conn_hdl)) {
            Serial.println("Temperature Measurement 'Indicate' enabled");
        } else {
            Serial.println("Temperature Measurement 'Indicate' disabled");
        }
    }
}
```

4. Repeat the same procedure for any other BLECharacteristics in your service.

Full Sample Code

The full sample code for this example can be seen below:

```
/*********************************************************************
 This is an example for our nRF52 based Bluefruit LE modules

 Pick one up today in the adafruit shop!
```

```
 Adafruit invests time and resources providing this open source code,
 please support Adafruit and open-source hardware by purchasing
 products from Adafruit!

 MIT license, check LICENSE for more information
 All text above, and the splash screen below must be included in
 any redistribution
*********************************************************************/
#include <bluefruit.h>

/* HRM Service Definitions
 * Heart Rate Monitor Service:  0x180D
 * Heart Rate Measurement Char: 0x2A37
 * Body Sensor Location Char:   0x2A38
 */
BLEService        hrms = BLEService(UUID16_SVC_HEART_RATE);
BLECharacteristic hrmc = BLECharacteristic(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLECharacteristic bslc = BLECharacteristic(UUID16_CHR_BODY_SENSOR_LOCATION);

BLEDis bledis;    // DIS (Device Information Service) helper class instance
BLEBas blebas;    // BAS (Battery Service) helper class instance

uint8_t  bps = 0;

void setup()
{
  Serial.begin(115200);
  while ( !Serial ) delay(10);   // for nrf52840 with native usb

  Serial.println("Bluefruit52 HRM Example");
  Serial.println("-----------------------\n");

  // Initialise the Bluefruit module
  Serial.println("Initialise the Bluefruit nRF52 module");
  Bluefruit.begin();

  // Set the connect/disconnect callback handlers
  Bluefruit.Periph.setConnectCallback(connect_callback);
  Bluefruit.Periph.setDisconnectCallback(disconnect_callback);

  // Configure and Start the Device Information Service
  Serial.println("Configuring the Device Information Service");
  bledis.setManufacturer("Adafruit Industries");
  bledis.setModel("Bluefruit Feather52");
  bledis.begin();

  // Start the BLE Battery Service and set it to 100%
  Serial.println("Configuring the Battery Service");
  blebas.begin();
  blebas.write(100);

  // Setup the Heart Rate Monitor service using
  // BLEService and BLECharacteristic classes
  Serial.println("Configuring the Heart Rate Monitor Service");
  setupHRM();

  // Setup the advertising packet(s)
  Serial.println("Setting up the advertising payload(s)");
  startAdv();

  Serial.println("Ready Player One!!!");
  Serial.println("\nAdvertising");
}

void startAdv(void)
{
  // Advertising packet
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
```

```
  Bluefruit.Advertising.addTxPower();

  // Include HRM Service UUID
  Bluefruit.Advertising.addService(hrms);

  // Include Name
  Bluefruit.Advertising.addName();

  /* Start Advertising
   * - Enable auto advertising if disconnected
   * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
   * - Timeout for fast mode is 30 seconds
   * - Start(timeout) with timeout = 0 will advertise forever (until connected)
   *
   * For recommended advertising interval
   * https://developer.apple.com/library/content/qa/qa1931/_index.html
   */
  Bluefruit.Advertising.restartOnDisconnect(true);
  Bluefruit.Advertising.setInterval(32, 244);    // in unit of 0.625 ms
  Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
  Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising
after n seconds
}

void setupHRM(void)
{
  // Configure the Heart Rate Monitor service
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.service.heart_rate.xml
  // Supported Characteristics:
  // Name                         UUID    Requirement Properties
  // ---------------------------- ------  ----------- ----------
  // Heart Rate Measurement       0x2A37  Mandatory   Notify
  // Body Sensor Location         0x2A38  Optional    Read
  // Heart Rate Control Point     0x2A39  Conditional Write       <-- Not used here
  hrms.begin();

  // Note: You must call .begin() on the BLEService before calling .begin() on
  // any characteristic(s) within that service definition.. Calling .begin() on
  // a BLECharacteristic will cause it to be added to the last BLEService that
  // was 'begin()'ed!

  // Configure the Heart Rate Measurement characteristic
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
  // Properties = Notify
  // Min Len    = 1
  // Max Len    = 8
  //    B0      = UINT8  - Flag (MANDATORY)
  //      b5:7  = Reserved
  //      b4    = RR-Internal (0 = Not present, 1 = Present)
  //      b3    = Energy expended status (0 = Not present, 1 = Present)
  //      b1:2  = Sensor contact status (0+1 = Not supported, 2 = Supported but
contact not detected, 3 = Supported and detected)
  //      b0    = Value format (0 = UINT8, 1 = UINT16)
  //    B1      = UINT8  - 8-bit heart rate measurement value in BPM
  //    B2:3    = UINT16 - 16-bit heart rate measurement value in BPM
  //    B4:5    = UINT16 - Energy expended in joules
  //    B6:7    = UINT16 - RR Internal (1/1024 second resolution)
  hrmc.setProperties(CHR_PROPS_NOTIFY);
  hrmc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  hrmc.setFixedLen(2);
  hrmc.setCccdWriteCallback(cccd_callback);  // Optionally capture CCCD updates
  hrmc.begin();
  uint8_t hrmdata[2] = { 0b00000110, 0x40 }; // Set the characteristic to use 8-bit
values, with the sensor connected and detected
  hrmc.write(hrmdata, 2);

  // Configure the Body Sensor Location characteristic
```

```
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
  // Properties = Read
  // Min Len    = 1
  // Max Len    = 1
  //    B0      = UINT8 - Body Sensor Location
  //      0     = Other
  //      1     = Chest
  //      2     = Wrist
  //      3     = Finger
  //      4     = Hand
  //      5     = Ear Lobe
  //      6     = Foot
  //      7:255 = Reserved
  bslc.setProperties(CHR_PROPS_READ);
  bslc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  bslc.setFixedLen(1);
  bslc.begin();
  bslc.write8(2);    // Set the characteristic to 'Wrist' (2)
}

void connect_callback(uint16_t conn_handle)
{
  // Get the reference to current connection
  BLEConnection* connection = Bluefruit.Connection(conn_handle);

  char central_name[32] = { 0 };
  connection->getPeerName(central_name, sizeof(central_name));

  Serial.print("Connected to ");
  Serial.println(central_name);
}

/**
 * Callback invoked when a connection is dropped
 * @param conn_handle connection where this event happens
 * @param reason is a BLE_HCI_STATUS_CODE which can be found in ble_hci.h
 */
void disconnect_callback(uint16_t conn_handle, uint8_t reason)
{
  (void) conn_handle;
  (void) reason;

  Serial.print("Disconnected, reason = 0x"); Serial.println(reason, HEX);
  Serial.println("Advertising!");
}

void cccd_callback(uint16_t conn_hdl, BLECharacteristic* chr, uint16_t cccd_value)
{
    // Display the raw request packet
    Serial.print("CCCD Updated: ");
    //Serial.printBuffer(request->data, request->len);
    Serial.print(cccd_value);
    Serial.println("");

    // Check the characteristic this CCCD update is associated with in case
    // this handler is used for multiple CCCD records.
    if (chr->uuid == hrmc.uuid) {
        if (chr->notifyEnabled(conn_hdl)) {
            Serial.println("Heart Rate Measurement 'Notify' enabled");
        } else {
            Serial.println("Heart Rate Measurement 'Notify' disabled");
        }
    }
}

void loop()
{
  digitalToggle(LED_RED);
```

```
  if ( Bluefruit.connected() ) {
    uint8_t hrmdata[2] = { 0b00000110, bps++ };           // Sensor connected,
increment BPS value

    // Note: We use .notify instead of .write!
    // If it is connected but CCCD is not enabled
    // The characteristic's value is still updated although notification is not sent
    if ( hrmc.notify(hrmdata, sizeof(hrmdata)) ){
      Serial.print("Heart Rate Measurement updated to: "); Serial.println(bps);
    }else{
      Serial.println("ERROR: Notify not set in the CCCD or not connected!");
    }
  }

  // Only send update once per second
  delay(1000);
}
```

BLE Pin I/O

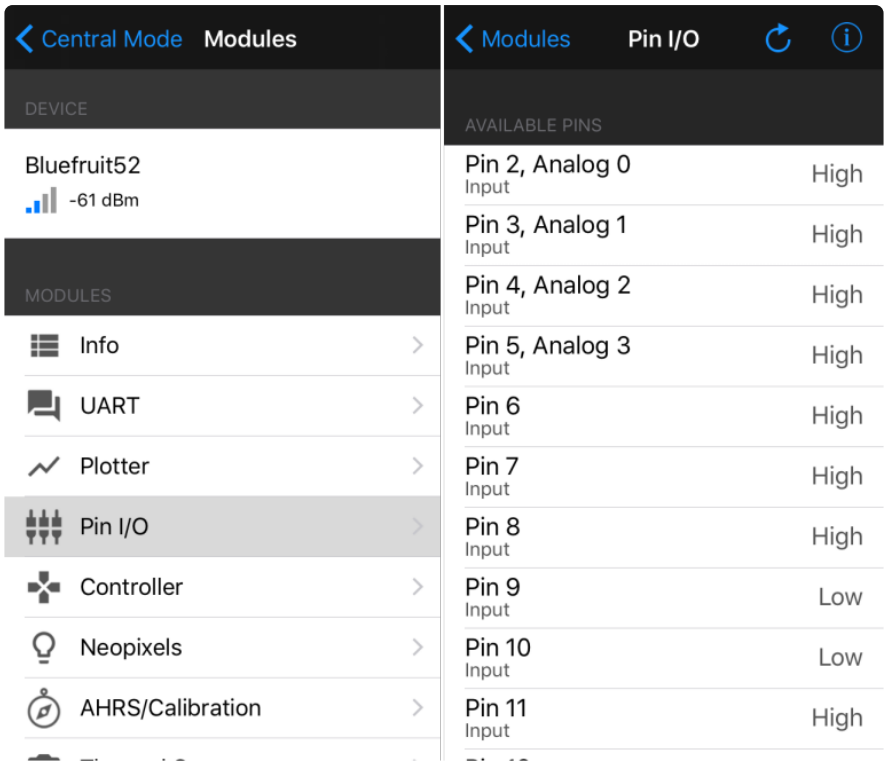
Firmata is a generic protocol for communicating with microcontrollers and controlling the board's pins such as setting the GPIO outputs and inputs, PWM output, analog reads, etc....

Setup

In order to run this demo, you will need to open Bluefruit LE Connect on your mobile device using our free iOS (https://adafru.it/f4H), Android (https://adafru.it/f4G) or OS X (https://adafru.it/o9F) applications.

- Load the **StandardFirmataBLE** example sketch (https://adafru.it/BI4) in the Arduino IDE
- Compile the sketch and flash it to your nRF52 based Feather
- Once you are done uploading, open the **Serial Monitor** (Tools > Serial Monitor)
- Open the **Bluefruit LE Connect** application on your mobile device
- Connect to the appropriate target (probably '**Bluefruit52**')
- Once connected switch to the **Pin I/O** application inside the app

For more information using Pin I/O module, you could check out this tutorial here https://learn.adafruit.com/bluefruit-le-connect-for-ios/pin-i-o



Complete Code

The latest version of this code is always available on Github (https://adafru.it/vaN), and in the **Examples** folder of the nRF52 BSP.

The code below is provided for convenience sake, but may be out of date! See the link above for the latest code.

```
/*
  Firmata is a generic protocol for communicating with microcontrollers
  from software on a host computer. It is intended to work with
  any host computer software package.

  To download a host software package, please click on the following link
  to open the list of Firmata client libraries in your default browser.

  https://github.com/firmata/arduino#firmata-client-libraries

  Copyright (C) 2006-2008 Hans-Christoph Steiner.  All rights reserved.
  Copyright (C) 2010-2011 Paul Stoffregen.  All rights reserved.
  Copyright (C) 2009 Shigeru Kobayashi.  All rights reserved.
  Copyright (C) 2009-2016 Jeff Hoefs.  All rights reserved.

  This library is free software; you can redistribute it and/or
  modify it under the terms of the GNU Lesser General Public
  License as published by the Free Software Foundation; either
  version 2.1 of the License, or (at your option) any later version.

  See file LICENSE.txt for further informations on licensing terms.

  Last updated October 16th, 2016
*/
```

```
// Adafruit nRF52 Boards require Firmata version is at least 2.5.7

#include <bluefruit.h>
#include <Servo.h>
#include <Wire.h>
#include <Firmata.h>

#define I2C_WRITE                   B00000000
#define I2C_READ                    B00001000
#define I2C_READ_CONTINUOUSLY       B00010000
#define I2C_STOP_READING            B00011000
#define I2C_READ_WRITE_MODE_MASK    B00011000
#define I2C_10BIT_ADDRESS_MODE_MASK B00100000
#define I2C_END_TX_MASK             B01000000
#define I2C_STOP_TX                 1
#define I2C_RESTART_TX              0
#define I2C_MAX_QUERIES             8
#define I2C_REGISTER_NOT_SPECIFIED  -1

// the minimum interval for sampling analog input
#define MINIMUM_SAMPLING_INTERVAL   1

// Adafruit
uint8_t ANALOG_TO_PIN(uint8_t n)
{
  switch (n)
  {
    case 0 : return PIN_A0;
    case 1 : return PIN_A1;
    case 2 : return PIN_A2;
    case 3 : return PIN_A3;
    case 4 : return PIN_A4;
    case 5 : return PIN_A5;
    case 6 : return PIN_A6;
    case 7 : return PIN_A7;
  }

  return 127;
}


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

#ifdef FIRMATA_SERIAL_FEATURE
SerialFirmata serialFeature;
#endif

BLEUart bleuart;

/* analog inputs */
int analogInputsToReport = 0; // bitwise array to store pin reporting

/* digital input ports */
byte reportPINs[TOTAL_PORTS];       // 1 = report this port, 0 = silence
byte previousPINs[TOTAL_PORTS];     // previous 8 bits sent

/* pins configuration */
byte portConfigInputs[TOTAL_PORTS]; // each bit: 1 = pin in INPUT, 0 = anything else

/* timer variables */
unsigned long currentMillis;        // store the current value from millis()
unsigned long previousMillis;       // for comparison with currentMillis
unsigned int samplingInterval = 19; // how often to run the main loop (in ms)

/* i2c data */
struct i2c_device_info {
  byte addr;
```

```
  int reg;
  byte bytes;
  byte stopTX;
};

/* for i2c read continuous more */
i2c_device_info query[I2C_MAX_QUERIES];

byte i2cRxData[64];
boolean isI2CEnabled = false;
signed char queryIndex = -1;
// default delay time between i2c read request and Wire.requestFrom()
unsigned int i2cReadDelayTime = 0;

Servo servos[MAX_SERVOS];
byte servoPinMap[TOTAL_PINS];
byte detachedServos[MAX_SERVOS];
byte detachedServoCount = 0;
byte servoCount = 0;

boolean isResetting = false;

// Forward declare a few functions to avoid compiler errors with older versions
// of the Arduino IDE.
void setPinModeCallback(byte, int);
void reportAnalogCallback(byte analogPin, int value);
void sysexCallback(byte, byte, byte*);

/* utility functions */
void wireWrite(byte data)
{
#if ARDUINO >= 100
  Wire.write((byte)data);
#else
  Wire.send(data);
#endif
}

byte wireRead(void)
{
#if ARDUINO >= 100
  return Wire.read();
#else
  return Wire.receive();
#endif
}

/*==============================================================================
 * FUNCTIONS
 *============================================================================*/

void attachServo(byte pin, int minPulse, int maxPulse)
{
  if (servoCount < MAX_SERVOS) {
    // reuse indexes of detached servos until all have been reallocated
    if (detachedServoCount > 0) {
      servoPinMap[pin] = detachedServos[detachedServoCount - 1];
      if (detachedServoCount > 0) detachedServoCount--;
    } else {
      servoPinMap[pin] = servoCount;
      servoCount++;
    }
    if (minPulse > 0 && maxPulse > 0) {
      servos[servoPinMap[pin]].attach(PIN_TO_DIGITAL(pin), minPulse, maxPulse);
    } else {
      servos[servoPinMap[pin]].attach(PIN_TO_DIGITAL(pin));
    }
  } else {
    Firmata.sendString("Max servos attached");
```

```
  }
}

void detachServo(byte pin)
{
  servos[servoPinMap[pin]].detach();
  // if we're detaching the last servo, decrement the count
  // otherwise store the index of the detached servo
  if (servoPinMap[pin] == servoCount && servoCount > 0) {
    servoCount--;
  } else if (servoCount > 0) {
    // keep track of detached servos because we want to reuse their indexes
    // before incrementing the count of attached servos
    detachedServoCount++;
    detachedServos[detachedServoCount - 1] = servoPinMap[pin];
  }

  servoPinMap[pin] = 255;
}

void enableI2CPins()
{
  byte i;
  // is there a faster way to do this? would probaby require importing
  // Arduino.h to get SCL and SDA pins
  for (i = 0; i < TOTAL_PINS; i++) {
    if (IS_PIN_I2C(i)) {
      // mark pins as i2c so they are ignore in non i2c data requests
      setPinModeCallback(i, PIN_MODE_I2C);
    }
  }

  isI2CEnabled = true;

  Wire.begin();
}

/* disable the i2c pins so they can be used for other functions */
void disableI2CPins() {
  isI2CEnabled = false;
  // disable read continuous mode for all devices
  queryIndex = -1;
}

void readAndReportData(byte address, int theRegister, byte numBytes, byte stopTX) {
  // allow I2C requests that don't require a register read
  // for example, some devices using an interrupt pin to signify new data available
  // do not always require the register read so upon interrupt you call
Wire.requestFrom()
  if (theRegister != I2C_REGISTER_NOT_SPECIFIED) {
    Wire.beginTransmission(address);
    wireWrite((byte)theRegister);
    Wire.endTransmission(stopTX); // default = true
    // do not set a value of 0
    if (i2cReadDelayTime > 0) {
      // delay is necessary for some devices such as WiiNunchuck
      delayMicroseconds(i2cReadDelayTime);
    }
  } else {
    theRegister = 0;  // fill the register with a dummy value
  }

  Wire.requestFrom(address, numBytes);  // all bytes are returned in requestFrom

  // check to be sure correct number of bytes were returned by slave
  if (numBytes < Wire.available()) {
    Firmata.sendString("I2C: Too many bytes received");
  } else if (numBytes > Wire.available()) {
    Firmata.sendString("I2C: Too few bytes received");
```

```
  }

  i2cRxData[0] = address;
  i2cRxData[1] = theRegister;

  for (int i = 0; i < numBytes && Wire.available(); i++) {
    i2cRxData[2 + i] = wireRead();
  }

  // send slave address, register and received bytes
  Firmata.sendSysex(SYSEX_I2C_REPLY, numBytes + 2, i2cRxData);
}

void outputPort(byte portNumber, byte portValue, byte forceSend)
{
  // pins not configured as INPUT are cleared to zeros
  portValue = portValue & portConfigInputs[portNumber];
  // only send if the value is different than previously sent
  if (forceSend || previousPINs[portNumber] != portValue) {
    Firmata.sendDigitalPort(portNumber, portValue);
    previousPINs[portNumber] = portValue;
  }
}

/* -----------------------------------------------------------------------------
 * check all the active digital inputs for change of state, then add any events
 * to the Serial output queue using Serial.print() */
void checkDigitalInputs(void)
{
  /* Using non-looping code allows constants to be given to readPort().
   * The compiler will apply substantial optimizations if the inputs
   * to readPort() are compile-time constants. */
  if (TOTAL_PORTS > 0 && reportPINs[0]) outputPort(0, readPort(0,
portConfigInputs[0]), false);
  if (TOTAL_PORTS > 1 && reportPINs[1]) outputPort(1, readPort(1,
portConfigInputs[1]), false);
  if (TOTAL_PORTS > 2 && reportPINs[2]) outputPort(2, readPort(2,
portConfigInputs[2]), false);
  if (TOTAL_PORTS > 3 && reportPINs[3]) outputPort(3, readPort(3,
portConfigInputs[3]), false);
  if (TOTAL_PORTS > 4 && reportPINs[4]) outputPort(4, readPort(4,
portConfigInputs[4]), false);
  if (TOTAL_PORTS > 5 && reportPINs[5]) outputPort(5, readPort(5,
portConfigInputs[5]), false);
  if (TOTAL_PORTS > 6 && reportPINs[6]) outputPort(6, readPort(6,
portConfigInputs[6]), false);
  if (TOTAL_PORTS > 7 && reportPINs[7]) outputPort(7, readPort(7,
portConfigInputs[7]), false);
  if (TOTAL_PORTS > 8 && reportPINs[8]) outputPort(8, readPort(8,
portConfigInputs[8]), false);
  if (TOTAL_PORTS > 9 && reportPINs[9]) outputPort(9, readPort(9,
portConfigInputs[9]), false);
  if (TOTAL_PORTS > 10 && reportPINs[10]) outputPort(10, readPort(10,
portConfigInputs[10]), false);
  if (TOTAL_PORTS > 11 && reportPINs[11]) outputPort(11, readPort(11,
portConfigInputs[11]), false);
  if (TOTAL_PORTS > 12 && reportPINs[12]) outputPort(12, readPort(12,
portConfigInputs[12]), false);
  if (TOTAL_PORTS > 13 && reportPINs[13]) outputPort(13, readPort(13,
portConfigInputs[13]), false);
  if (TOTAL_PORTS > 14 && reportPINs[14]) outputPort(14, readPort(14,
portConfigInputs[14]), false);
  if (TOTAL_PORTS > 15 && reportPINs[15]) outputPort(15, readPort(15,
portConfigInputs[15]), false);
}

// -----------------------------------------------------------------------------
/* sets the pin mode to the correct state and sets the relevant bits in the
 * two bit-arrays that track Digital I/O and PWM status
```

```
 */
void setPinModeCallback(byte pin, int mode)
{
  if (Firmata.getPinMode(pin) == PIN_MODE_IGNORE)
    return;

  if (Firmata.getPinMode(pin) == PIN_MODE_I2C && isI2CEnabled && mode !=
PIN_MODE_I2C) {
    // disable i2c so pins can be used for other functions
    // the following if statements should reconfigure the pins properly
    disableI2CPins();
  }
  if (IS_PIN_DIGITAL(pin) && mode != PIN_MODE_SERVO) {
    if (servoPinMap[pin] < MAX_SERVOS && servos[servoPinMap[pin]].attached()) {
      detachServo(pin);
    }
  }
  if (IS_PIN_ANALOG(pin)) {
    reportAnalogCallback(PIN_TO_ANALOG(pin), mode == PIN_MODE_ANALOG ? 1 : 0); //
turn on/off reporting
  }
  if (IS_PIN_DIGITAL(pin)) {
    if (mode == INPUT || mode == PIN_MODE_PULLUP) {
      portConfigInputs[pin / 8] |= (1 << (pin & 7));
    } else {
      portConfigInputs[pin / 8] &= ~(1 << (pin & 7));
    }
  }
  Firmata.setPinState(pin, 0);
  switch (mode) {
    case PIN_MODE_ANALOG:
      if (IS_PIN_ANALOG(pin)) {
        if (IS_PIN_DIGITAL(pin)) {
          pinMode(PIN_TO_DIGITAL(pin), INPUT);    // disable output driver
#if ARDUINO <= 100
          // deprecated since Arduino 1.0.1 - TODO: drop support in Firmata 2.6
          digitalWrite(PIN_TO_DIGITAL(pin), LOW); // disable internal pull-ups
#endif
        }
        Firmata.setPinMode(pin, PIN_MODE_ANALOG);
      }
      break;
    case INPUT:
// Adafruit: Input without pull up cause pin state changes randomly --> lots of
transmission data
//      if (IS_PIN_DIGITAL(pin)) {
//        pinMode(PIN_TO_DIGITAL(pin), INPUT);    // disable output driver
//#if ARDUINO <= 100
//        // deprecated since Arduino 1.0.1 - TODO: drop support in Firmata 2.6
//        digitalWrite(PIN_TO_DIGITAL(pin), LOW); // disable internal pull-ups
//#endif
//        Firmata.setPinMode(pin, INPUT);
//      }
//      break;
    case PIN_MODE_PULLUP:
      if (IS_PIN_DIGITAL(pin)) {
        pinMode(PIN_TO_DIGITAL(pin), INPUT_PULLUP);
        Firmata.setPinMode(pin, PIN_MODE_PULLUP);
        Firmata.setPinState(pin, 1);
      }
      break;
    case OUTPUT:
      if (IS_PIN_DIGITAL(pin)) {
        if (Firmata.getPinMode(pin) == PIN_MODE_PWM) {
          // Disable PWM if pin mode was previously set to PWM.
          digitalWrite(PIN_TO_DIGITAL(pin), LOW);
        }
        pinMode(PIN_TO_DIGITAL(pin), OUTPUT);
        Firmata.setPinMode(pin, OUTPUT);
```

```
      }
      break;
    case PIN_MODE_PWM:
      if (IS_PIN_PWM(pin)) {
        pinMode(PIN_TO_PWM(pin), OUTPUT);
        analogWrite(PIN_TO_PWM(pin), 0);
        Firmata.setPinMode(pin, PIN_MODE_PWM);
      }
      break;
    case PIN_MODE_SERVO:
      if (IS_PIN_DIGITAL(pin)) {
        Firmata.setPinMode(pin, PIN_MODE_SERVO);
        if (servoPinMap[pin] == 255 || !servos[servoPinMap[pin]].attached()) {
          // pass -1 for min and max pulse values to use default values set
          // by Servo library
          attachServo(pin, -1, -1);
        }
      }
      break;
    case PIN_MODE_I2C:
      if (IS_PIN_I2C(pin)) {
        // mark the pin as i2c
        // the user must call I2C_CONFIG to enable I2C for a device
        Firmata.setPinMode(pin, PIN_MODE_I2C);
      }
      break;
    case PIN_MODE_SERIAL:
#ifdef FIRMATA_SERIAL_FEATURE
      serialFeature.handlePinMode(pin, PIN_MODE_SERIAL);
#endif
      break;
    default:
      Firmata.sendString("Unknown pin mode"); // TODO: put error msgs in EEPROM
  }
  // TODO: save status to EEPROM here, if changed
}

/*
 * Sets the value of an individual pin. Useful if you want to set a pin value but
 * are not tracking the digital port state.
 * Can only be used on pins configured as OUTPUT.
 * Cannot be used to enable pull-ups on Digital INPUT pins.
 */
void setPinValueCallback(byte pin, int value)
{
  if (pin < TOTAL_PINS && IS_PIN_DIGITAL(pin)) {
    if (Firmata.getPinMode(pin) == OUTPUT) {
      Firmata.setPinState(pin, value);
      digitalWrite(PIN_TO_DIGITAL(pin), value);
    }
  }
}

void analogWriteCallback(byte pin, int value)
{
  if (pin < TOTAL_PINS) {
    switch (Firmata.getPinMode(pin)) {
      case PIN_MODE_SERVO:
        if (IS_PIN_DIGITAL(pin))
          servos[servoPinMap[pin]].write(value);
        Firmata.setPinState(pin, value);
        break;
      case PIN_MODE_PWM:
        if (IS_PIN_PWM(pin))
          analogWrite(PIN_TO_PWM(pin), value);
        Firmata.setPinState(pin, value);
        break;
    }
  }
```

```
}

void digitalWriteCallback(byte port, int value)
{
  byte pin, lastPin, pinValue, mask = 1, pinWriteMask = 0;

  if (port < TOTAL_PORTS) {
    // create a mask of the pins on this port that are writable.
    lastPin = port * 8 + 8;
    if (lastPin > TOTAL_PINS) lastPin = TOTAL_PINS;
    for (pin = port * 8; pin < lastPin; pin++) {
      // do not disturb non-digital pins (eg, Rx & Tx)
      if (IS_PIN_DIGITAL(pin)) {
        // do not touch pins in PWM, ANALOG, SERVO or other modes
        if (Firmata.getPinMode(pin) == OUTPUT || Firmata.getPinMode(pin) == INPUT) {
          pinValue = ((byte)value & mask) ? 1 : 0;
          if (Firmata.getPinMode(pin) == OUTPUT) {
            pinWriteMask |= mask;
          } else if (Firmata.getPinMode(pin) == INPUT && pinValue == 1 &&
Firmata.getPinState(pin) != 1) {
            // only handle INPUT here for backwards compatibility
#if ARDUINO > 100
            pinMode(pin, INPUT_PULLUP);
#else
            // only write to the INPUT pin to enable pullups if Arduino v1.0.0 or
earlier
            pinWriteMask |= mask;
#endif
          }
          Firmata.setPinState(pin, pinValue);
        }
      }
      mask = mask << 1;
    }
    writePort(port, (byte)value, pinWriteMask);
  }
}


// -----------------------------------------------------------------------------
/* sets bits in a bit array (int) to toggle the reporting of the analogIns
 */
//void FirmataClass::setAnalogPinReporting(byte pin, byte state) {
//}
void reportAnalogCallback(byte analogPin, int value)
{
  if (analogPin < TOTAL_ANALOG_PINS) {
    if (value == 0) {
      analogInputsToReport = analogInputsToReport & ~ (1 << analogPin);
    } else {
      analogInputsToReport = analogInputsToReport | (1 << analogPin);
      // prevent during system reset or all analog pin values will be reported
      // which may report noise for unconnected analog pins
      if (!isResetting) {
        // Send pin value immediately. This is helpful when connected via
        // ethernet, wi-fi or bluetooth so pin states can be known upon
        // reconnecting.
        Firmata.sendAnalog(analogPin, analogRead( ANALOG_TO_PIN(analogPin) ) );
      }
    }
  }
  // TODO: save status to EEPROM here, if changed
}

void reportDigitalCallback(byte port, int value)
{
  if (port < TOTAL_PORTS) {
    reportPINs[port] = (byte)value;
    // Send port value immediately. This is helpful when connected via
```

```
    // ethernet, wi-fi or bluetooth so pin states can be known upon
    // reconnecting.
    if (value) outputPort(port, readPort(port, portConfigInputs[port]), true);
  }
  // do not disable analog reporting on these 8 pins, to allow some
  // pins used for digital, others analog.  Instead, allow both types
  // of reporting to be enabled, but check if the pin is configured
  // as analog when sampling the analog inputs.  Likewise, while
  // scanning digital pins, portConfigInputs will mask off values from any
  // pins configured as analog
}

/*==============================================================================
 * SYSEX-BASED commands
 *============================================================================*/

void sysexCallback(byte command, byte argc, byte *argv)
{
  byte mode;
  byte stopTX;
  byte slaveAddress;
  byte data;
  int slaveRegister;
  unsigned int delayTime;

  switch (command) {
    case I2C_REQUEST:
      mode = argv[1] & I2C_READ_WRITE_MODE_MASK;
      if (argv[1] & I2C_10BIT_ADDRESS_MODE_MASK) {
        Firmata.sendString("10-bit addressing not supported");
        return;
      }
      else {
        slaveAddress = argv[0];
      }

      // need to invert the logic here since 0 will be default for client
      // libraries that have not updated to add support for restart tx
      if (argv[1] & I2C_END_TX_MASK) {
        stopTX = I2C_RESTART_TX;
      }
      else {
        stopTX = I2C_STOP_TX; // default
      }

      switch (mode) {
        case I2C_WRITE:
          Wire.beginTransmission(slaveAddress);
          for (byte i = 2; i < argc; i += 2) {
            data = argv[i] + (argv[i + 1] << 7);
            wireWrite(data);
          }
          Wire.endTransmission();
          delayMicroseconds(70);
          break;
        case I2C_READ:
          if (argc == 6) {
            // a slave register is specified
            slaveRegister = argv[2] + (argv[3] << 7);
            data = argv[4] + (argv[5] << 7);  // bytes to read
          }
          else {
            // a slave register is NOT specified
            slaveRegister = I2C_REGISTER_NOT_SPECIFIED;
            data = argv[2] + (argv[3] << 7);  // bytes to read
          }
          readAndReportData(slaveAddress, (int)slaveRegister, data, stopTX);
          break;
        case I2C_READ_CONTINUOUSLY:
```

```
        if ((queryIndex + 1) >= I2C_MAX_QUERIES) {
          // too many queries, just ignore
          Firmata.sendString("too many queries");
          break;
        }
        if (argc == 6) {
          // a slave register is specified
          slaveRegister = argv[2] + (argv[3] << 7);
          data = argv[4] + (argv[5] << 7);  // bytes to read
        }
        else {
          // a slave register is NOT specified
          slaveRegister = (int)I2C_REGISTER_NOT_SPECIFIED;
          data = argv[2] + (argv[3] << 7);  // bytes to read
        }
        queryIndex++;
        query[queryIndex].addr = slaveAddress;
        query[queryIndex].reg = slaveRegister;
        query[queryIndex].bytes = data;
        query[queryIndex].stopTX = stopTX;
        break;
      case I2C_STOP_READING:
        byte queryIndexToSkip;
        // if read continuous mode is enabled for only 1 i2c device, disable
        // read continuous reporting for that device
        if (queryIndex <= 0) {
          queryIndex = -1;
        } else {
          queryIndexToSkip = 0;
          // if read continuous mode is enabled for multiple devices,
          // determine which device to stop reading and remove it's data from
          // the array, shifiting other array data to fill the space
          for (byte i = 0; i < queryIndex + 1; i++) {
            if (query[i].addr == slaveAddress) {
              queryIndexToSkip = i;
              break;
            }
          }

          for (byte i = queryIndexToSkip; i < queryIndex + 1; i++) {
            if (i < I2C_MAX_QUERIES) {
              query[i].addr = query[i + 1].addr;
              query[i].reg = query[i + 1].reg;
              query[i].bytes = query[i + 1].bytes;
              query[i].stopTX = query[i + 1].stopTX;
            }
          }
          queryIndex--;
        }
        break;
      default:
        break;
    }
    break;
  case I2C_CONFIG:
    delayTime = (argv[0] + (argv[1] << 7));

    if (delayTime > 0) {
      i2cReadDelayTime = delayTime;
    }

    if (!isI2CEnabled) {
      enableI2CPins();
    }

    break;
  case SERVO_CONFIG:
    if (argc > 4) {
      // these vars are here for clarity, they'll optimized away by the compiler
```

```
        byte pin = argv[0];
        int minPulse = argv[1] + (argv[2] << 7);
        int maxPulse = argv[3] + (argv[4] << 7);

        if (IS_PIN_DIGITAL(pin)) {
          if (servoPinMap[pin] < MAX_SERVOS && servos[servoPinMap[pin]].attached())
{
            detachServo(pin);
          }
          attachServo(pin, minPulse, maxPulse);
          setPinModeCallback(pin, PIN_MODE_SERVO);
        }
      }
      break;
    case SAMPLING_INTERVAL:
      if (argc > 1) {
        samplingInterval = argv[0] + (argv[1] << 7);
        if (samplingInterval < MINIMUM_SAMPLING_INTERVAL) {
          samplingInterval = MINIMUM_SAMPLING_INTERVAL;
        }
      } else {
        //Firmata.sendString("Not enough data");
      }
      break;
    case EXTENDED_ANALOG:
      if (argc > 1) {
        int val = argv[1];
        if (argc > 2) val |= (argv[2] << 7);
        if (argc > 3) val |= (argv[3] << 14);
        analogWriteCallback(argv[0], val);
      }
      break;
    case CAPABILITY_QUERY:
      Firmata.write(START_SYSEX);
      Firmata.write(CAPABILITY_RESPONSE);
      for (byte pin = 0; pin < TOTAL_PINS; pin++) {
        if (IS_PIN_DIGITAL(pin)) {
          Firmata.write((byte)INPUT);
          Firmata.write(1);
          Firmata.write((byte)PIN_MODE_PULLUP);
          Firmata.write(1);
          Firmata.write((byte)OUTPUT);
          Firmata.write(1);
        }
        if (IS_PIN_ANALOG(pin)) {
          Firmata.write(PIN_MODE_ANALOG);
          Firmata.write(10); // 10 = 10-bit resolution
        }
        if (IS_PIN_PWM(pin)) {
          Firmata.write(PIN_MODE_PWM);
          Firmata.write(DEFAULT_PWM_RESOLUTION);
        }
        if (IS_PIN_DIGITAL(pin)) {
          Firmata.write(PIN_MODE_SERVO);
          Firmata.write(14);
        }
        if (IS_PIN_I2C(pin)) {
          Firmata.write(PIN_MODE_I2C);
          Firmata.write(1);  // TODO: could assign a number to map to SCL or SDA
        }
#ifdef FIRMATA_SERIAL_FEATURE
        serialFeature.handleCapability(pin);
#endif
        Firmata.write(127);
      }
      Firmata.write(END_SYSEX);
      break;
    case PIN_STATE_QUERY:
      if (argc > 0) {
```

```
        byte pin = argv[0];
        Firmata.write(START_SYSEX);
        Firmata.write(PIN_STATE_RESPONSE);
        Firmata.write(pin);
        if (pin < TOTAL_PINS) {
          Firmata.write(Firmata.getPinMode(pin));
          Firmata.write((byte)Firmata.getPinState(pin) & 0x7F);
          if (Firmata.getPinState(pin) & 0xFF80) Firmata.write((byte)
(Firmata.getPinState(pin) >> 7) & 0x7F);
          if (Firmata.getPinState(pin) & 0xC000) Firmata.write((byte)
(Firmata.getPinState(pin) >> 14) & 0x7F);
        }
        Firmata.write(END_SYSEX);
      }
      break;
    case ANALOG_MAPPING_QUERY:
      Firmata.write(START_SYSEX);
      Firmata.write(ANALOG_MAPPING_RESPONSE);
      for (byte pin = 0; pin < TOTAL_PINS; pin++) {
        Firmata.write(IS_PIN_ANALOG(pin) ? PIN_TO_ANALOG(pin) : 127);
      }
      Firmata.write(END_SYSEX);
      break;

    case SERIAL_MESSAGE:
#ifdef FIRMATA_SERIAL_FEATURE
      serialFeature.handleSysex(command, argc, argv);
#endif
      break;
  }
}

/*==============================================================================
 * SETUP()
 *============================================================================*/

void systemResetCallback()
{
  isResetting = true;

  // initialize a defalt state
  // TODO: option to load config from EEPROM instead of default

#ifdef FIRMATA_SERIAL_FEATURE
  serialFeature.reset();
#endif

  if (isI2CEnabled) {
    disableI2CPins();
  }

  for (byte i = 0; i < TOTAL_PORTS; i++) {
    reportPINs[i] = false;    // by default, reporting off
    portConfigInputs[i] = 0;  // until activated
    previousPINs[i] = 0;
  }

  for (byte i = 0; i < TOTAL_PINS; i++) {
    // pins with analog capability default to analog input
    // otherwise, pins default to digital output
    if (IS_PIN_ANALOG(i)) {
      // turns off pullup, configures everything
      setPinModeCallback(i, PIN_MODE_ANALOG);
    } else if (IS_PIN_DIGITAL(i)) {
      // sets the output to 0, configures portConfigInputs
      setPinModeCallback(i, OUTPUT);
    }

    servoPinMap[i] = 255;
```

```
  }
  // by default, do not report any analog inputs
  analogInputsToReport = 0;

  detachedServoCount = 0;
  servoCount = 0;

  /* send digital inputs to set the initial state on the host computer,
   * since once in the loop(), this firmware will only send on change */
  /*
  TODO: this can never execute, since no pins default to digital input
        but it will be needed when/if we support EEPROM stored config
  for (byte i=0; i < TOTAL_PORTS; i++) {
    outputPort(i, readPort(i, portConfigInputs[i]), true);
  }
  */
  isResetting = false;
}

void setup()
{
  Serial.begin(115200);
  while ( !Serial ) delay(10);   // for nrf52840 with native usb

  Serial.println("Bluefruit52 Standard Firmata via BLEUART Example");
  Serial.println("-----------------------------------------------\n");

  // Config the peripheral connection with maximum bandwidth
  // more SRAM required by SoftDevice
  // Note: All config***() function must be called before begin()
  Bluefruit.configPrphBandwidth(BANDWIDTH_MAX);

  Bluefruit.begin();
  Bluefruit.setTxPower(4);    // Check bluefruit.h for supported values

  // try to go as fast as possible, could be rejected by some central, increase it
if needed
  // iOS won't negotitate and will mostly use 30ms
  Bluefruit.Periph.setConnInterval(9, 24); // min = 9*1.25=11.25 ms, max =
23*1.25=30ms

  // Configure and Start BLE Uart Service
  // Firmata use several small write(1) --> buffering TXD is required to run
smoothly
  // Enable buffering TXD
  bleuart.begin();
  bleuart.bufferTXD(true);

  Firmata.setFirmwareVersion(FIRMATA_FIRMWARE_MAJOR_VERSION,
FIRMATA_FIRMWARE_MINOR_VERSION);

  Firmata.attach(ANALOG_MESSAGE, analogWriteCallback);
  Firmata.attach(DIGITAL_MESSAGE, digitalWriteCallback);
  Firmata.attach(REPORT_ANALOG, reportAnalogCallback);
  Firmata.attach(REPORT_DIGITAL, reportDigitalCallback);
  Firmata.attach(SET_PIN_MODE, setPinModeCallback);
  Firmata.attach(SET_DIGITAL_PIN_VALUE, setPinValueCallback);
  Firmata.attach(START_SYSEX, sysexCallback);
  Firmata.attach(SYSTEM_RESET, systemResetCallback);

  // use bleuart as transportation layer
  Firmata.begin(bleuart);

  // to use a port other than Serial, such as Serial1 on an Arduino Leonardo or
Mega,
  // Call begin(baud) on the alternate serial port and pass it to Firmata to begin
like this:
  // Serial1.begin(57600);
  // Firmata.begin(Serial1);
```

```
  // However do not do this if you are using SERIAL_MESSAGE

  //Firmata.begin(57600);
  //while (!Serial) {
  //  ; // wait for serial port to connect. Needed for ATmega32u4-based boards and
Arduino 101
  //}

  systemResetCallback();  // reset to default config

  // Set up and start advertising
  startAdv();
}

void startAdv(void)
{
  // Advertising packet
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
  Bluefruit.Advertising.addTxPower();

  // Include bleuart 128-bit uuid
  Bluefruit.Advertising.addService(bleuart);

  // Secondary Scan Response packet (optional)
  // Since there is no room for 'Name' in Advertising packet
  Bluefruit.ScanResponse.addName();

  /* Start Advertising
   * - Enable auto advertising if disconnected
   * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
   * - Timeout for fast mode is 30 seconds
   * - Start(timeout) with timeout = 0 will advertise forever (until connected)
   *
   * For recommended advertising interval
   * https://developer.apple.com/library/content/qa/qa1931/_index.html
   */
  Bluefruit.Advertising.restartOnDisconnect(true);
  Bluefruit.Advertising.setInterval(32, 244);    // in unit of 0.625 ms
  Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
  Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising
after n seconds
}

/*==========================================================================
 * LOOP()
 *==========================================================================*/
void loop()
{
  // Skip if not connected and bleuart notification is not enabled
  if (  !(Bluefruit.connected() && bleuart.notifyEnabled()) ) return;

  byte pin, analogPin;

  /* DIGITALREAD - as fast as possible, check for changes and output them to the
   * FTDI buffer using Serial.print()  */
  checkDigitalInputs();

  /* STREAMREAD - processing incoming messagse as soon as possible, while still
   * checking digital inputs.  */
  while (Firmata.available())
    Firmata.processInput();

  // TODO - ensure that Stream buffer doesn't go over 60 bytes

  currentMillis = millis();
  if (currentMillis - previousMillis > samplingInterval) {
    previousMillis += samplingInterval;
    /* ANALOGREAD - do all analogReads() at the configured sampling interval */
    for (pin = 0; pin < TOTAL_PINS; pin++) {
```

```
      if (IS_PIN_ANALOG(pin) && Firmata.getPinMode(pin) == PIN_MODE_ANALOG) {
        analogPin = PIN_TO_ANALOG(pin);
        if (analogInputsToReport & (1 << analogPin)) {
          Firmata.sendAnalog(analogPin, analogRead( ANALOG_TO_PIN(analogPin) ));
        }
      }
    }
    // report i2c data for all device with read continuous mode enabled
    if (queryIndex > -1) {
      for (byte i = 0; i < queryIndex + 1; i++) {
        readAndReportData(query[i].addr, query[i].reg, query[i].bytes,
query[i].stopTX);
      }
    }
  }

#ifdef FIRMATA_SERIAL_FEATURE
  serialFeature.update();
#endif

  // flush TXD since we use bufferTXD()
  bleuart.flushTXD();
}
```

Central BLEUART

This example show you how to use Feather nRF52/nRF52840 as a **Central** to talk to other Bluefruit (nRF52 or nRF51) peripherals exposing the bleuart (AKA 'NUS') service.

Client Services

Since the Central role accesses the GATT server on the peripheral, we first need to declare a client bleuart instance using the **BLEClientUart** helper class. We can also conveniently read Device Information if **BLEClientDis** is also used.

```
BLEClientDis  clientDis;
BLEClientUart clientUart;
```

Before we can configure client services, `Bluefruit.begin()` must be called with at least 1 for the number of concurrent connections supported in central mode. Since we won't be running the nRF52 as a peripheral in this instance, we will set the peripheral count to 0:

```
// Initialize Bluefruit with maximum connections as Peripheral = 0, Central = 1
Bluefruit.begin(0, 1);
```

Afterward this, the client service(s) must be initialized by calling their `begin()` function, and you can setup any callbacks that you wish to use from the helper class:

```
// Configure DIS client
clientDis.begin();

// Init BLE Central Uart Serivce
clientUart.begin();
clientUart.setRxCallback(bleuart_rx_callback);
```

Scanner

Let's start the advertising scanner to find a peripheral.

We'll hook up the scan result callback with **setRxCallback().**

Whenever advertising data is found by the scanner, it will be passed to this callback handler, and we can examine the advertising data there, and only connect to peripheral(s) that advertise the bleuart service.

Note: If the peripheral has multiple services and bleuart is not included in the UUID list in the advertising packet, you could optionally use another check such as matching the MAC address, name checking, using "another service", etc.

Once we find a peripheral that we wish to communicate with, call `Bluefruit.Central.connect()` to establish connection with it:

```
void setup()
{
  // Other set up .....

  /* Start Central Scanning
   * - Enable auto scan if disconnected
   * - Interval = 100 ms, window = 80 ms
   * - Don't use active scan
   * - Start(timeout) with timeout = 0 will scan forever (until connected)
   */
  Bluefruit.Scanner.setRxCallback(scan_callback);
  Bluefruit.Scanner.restartOnDisconnect(true);
  Bluefruit.Scanner.setInterval(160, 80); // in unit of 0.625 ms
  Bluefruit.Scanner.useActiveScan(false);
  Bluefruit.Scanner.start(0);                   // // 0 = Don't stop scanning after
n seconds
}

/**
 * Callback invoked when scanner pick up an advertising data
 * @param report Structural advertising data
 */
void scan_callback(ble_gap_evt_adv_report_t* report)
{
  // Check if advertising contain BleUart service
  if ( Bluefruit.Scanner.checkReportForService(report, clientUart) )
  {
    Serial.print("BLE UART service detected. Connecting ... ");

    // Connect to device with bleuart service in advertising
```

```
    Bluefruit.Central.connect(report);
  }
}
```

Central Role

You normally need to setup the Central mode device's **connect callback**, which fires when a connection is established/disconnected with a peripheral device. Alternatively you could poll the connection status with connected(), but callbacks help to simplify the code significantly:

```
// Callbacks for Central
Bluefruit.Central.setConnectCallback(connect_callback);
Bluefruit.Central.setDisconnectCallback(disconnect_callback);
```

In the connect callback, we will try to **discover** the bleuart service by browsing the GATT table of the peripheral. This will help to determine the handle values for characteristics (e.g TXD, RXD, etc.). This is all done by BLEClientUart's `.discover()`. Once the service is found, enable the TXD characteristic's CCCD to allow the peripheral to send data, and we are ready to send data back and forth between the devices:

```
void connect_callback(uint16_t conn_handle)
{
  Serial.println("Connected");

  Serial.print("Dicovering DIS ... ");
  if ( clientDis.discover(conn_handle) )
  {
    Serial.println("Found it");
    char buffer[32+1];

    // read and print out Manufacturer
    memset(buffer, 0, sizeof(buffer));
    if ( clientDis.getManufacturer(buffer, sizeof(buffer)) )
    {
      Serial.print("Manufacturer: ");
      Serial.println(buffer);
    }

    // read and print out Model Number
    memset(buffer, 0, sizeof(buffer));
    if ( clientDis.getModel(buffer, sizeof(buffer)) )
    {
      Serial.print("Model: ");
      Serial.println(buffer);
    }

    Serial.println();
  }

  Serial.print("Discovering BLE Uart Service ... ");

  if ( clientUart.discover(conn_handle) )
  {
    Serial.println("Found it");
```

```
    Serial.println("Enable TXD's notify");
    clientUart.enableTXD();

    Serial.println("Ready to receive from peripheral");
  }else
  {
    Serial.println("Found NONE");

    // disconect since we couldn't find bleuart service
    Bluefruit.Central.disconnect(conn_handle);
  }
}
```

Full Sample Code

The full sample code for this example can be seen below:

```
/*********************************************************************
 This is an example for our nRF52 based Bluefruit LE modules

 Pick one up today in the adafruit shop!

 Adafruit invests time and resources providing this open source code,
 please support Adafruit and open-source hardware by purchasing
 products from Adafruit!

 MIT license, check LICENSE for more information
 All text above, and the splash screen below must be included in
 any redistribution
*********************************************************************/

/*
 * This sketch demonstrate the central API(). A additional bluefruit
 * that has bleuart as peripheral is required for the demo.
 */
#include <bluefruit.h>

BLEClientBas  clientBas;  // battery client
BLEClientDis  clientDis;  // device information client
BLEClientUart clientUart; // bleuart client

void setup()
{
  Serial.begin(115200);
//  while ( !Serial ) delay(10);   // for nrf52840 with native usb

  Serial.println("Bluefruit52 Central BLEUART Example");
  Serial.println("-----------------------------------\n");

  // Initialize Bluefruit with maximum connections as Peripheral = 0, Central = 1
  // SRAM usage required by SoftDevice will increase dramatically with number of
connections
  Bluefruit.begin(0, 1);

  Bluefruit.setName("Bluefruit52 Central");

  // Configure Battery client
  clientBas.begin();

  // Configure DIS client
  clientDis.begin();

  // Init BLE Central Uart Serivce
```

```
  clientUart.begin();
  clientUart.setRxCallback(bleuart_rx_callback);

  // Increase Blink rate to different from PrPh advertising mode
  Bluefruit.setConnLedInterval(250);

  // Callbacks for Central
  Bluefruit.Central.setConnectCallback(connect_callback);
  Bluefruit.Central.setDisconnectCallback(disconnect_callback);

  /* Start Central Scanning
   * - Enable auto scan if disconnected
   * - Interval = 100 ms, window = 80 ms
   * - Don't use active scan
   * - Start(timeout) with timeout = 0 will scan forever (until connected)
   */
  Bluefruit.Scanner.setRxCallback(scan_callback);
  Bluefruit.Scanner.restartOnDisconnect(true);
  Bluefruit.Scanner.setInterval(160, 80); // in unit of 0.625 ms
  Bluefruit.Scanner.useActiveScan(false);
  Bluefruit.Scanner.start(0);                   // // 0 = Don't stop scanning after
n seconds
}

/**
 * Callback invoked when scanner pick up an advertising data
 * @param report Structural advertising data
 */
void scan_callback(ble_gap_evt_adv_report_t* report)
{
  // Check if advertising contain BleUart service
  if ( Bluefruit.Scanner.checkReportForService(report, clientUart) )
  {
    Serial.print("BLE UART service detected. Connecting ... ");

    // Connect to device with bleuart service in advertising
    Bluefruit.Central.connect(report);
  }else
  {
    // For Softdevice v6: after received a report, scanner will be paused
    // We need to call Scanner resume() to continue scanning
    Bluefruit.Scanner.resume();
  }
}

/**
 * Callback invoked when an connection is established
 * @param conn_handle
 */
void connect_callback(uint16_t conn_handle)
{
  Serial.println("Connected");

  Serial.print("Dicovering Device Information ... ");
  if ( clientDis.discover(conn_handle) )
  {
    Serial.println("Found it");
    char buffer[32+1];

    // read and print out Manufacturer
    memset(buffer, 0, sizeof(buffer));
    if ( clientDis.getManufacturer(buffer, sizeof(buffer)) )
    {
      Serial.print("Manufacturer: ");
      Serial.println(buffer);
    }

    // read and print out Model Number
    memset(buffer, 0, sizeof(buffer));
```

```
    if ( clientDis.getModel(buffer, sizeof(buffer)) )
    {
      Serial.print("Model: ");
      Serial.println(buffer);
    }

    Serial.println();
  }else
  {
    Serial.println("Found NONE");
  }

  Serial.print("Dicovering Battery ... ");
  if ( clientBas.discover(conn_handle) )
  {
    Serial.println("Found it");
    Serial.print("Battery level: ");
    Serial.print(clientBas.read());
    Serial.println("%");
  }else
  {
    Serial.println("Found NONE");
  }

  Serial.print("Discovering BLE Uart Service ... ");
  if ( clientUart.discover(conn_handle) )
  {
    Serial.println("Found it");

    Serial.println("Enable TXD's notify");
    clientUart.enableTXD();

    Serial.println("Ready to receive from peripheral");
  }else
  {
    Serial.println("Found NONE");

    // disconnect since we couldn't find bleuart service
    Bluefruit.disconnect(conn_handle);
  }
}

/**
 * Callback invoked when a connection is dropped
 * @param conn_handle
 * @param reason is a BLE_HCI_STATUS_CODE which can be found in ble_hci.h
 */
void disconnect_callback(uint16_t conn_handle, uint8_t reason)
{
  (void) conn_handle;
  (void) reason;

  Serial.print("Disconnected, reason = 0x"); Serial.println(reason, HEX);
}

/**
 * Callback invoked when uart received data
 * @param uart_svc Reference object to the service where the data
 * arrived. In this example it is clientUart
 */
void bleuart_rx_callback(BLEClientUart& uart_svc)
{
  Serial.print("[RX]: ");

  while ( uart_svc.available() )
  {
    Serial.print( (char) uart_svc.read() );
  }
```

```
  Serial.println();
}

void loop()
{
  if ( Bluefruit.Central.connected() )
  {
    // Not discovered yet
    if ( clientUart.discovered() )
    {
      // Discovered means in working state
      // Get Serial input and send to Peripheral
      if ( Serial.available() )
      {
        delay(2); // delay a bit for all characters to arrive

        char str[20+1] = { 0 };
        Serial.readBytes(str, 20);

        clientUart.print( str );
      }
    }
  }
}
```

Dual Roles BLEUART

If you are not familiar with Central Role, it is advised to look at the "Central BLEUART" example first then continue with this afterwards.

This example demonstrates how you can use a Feather nRF52/nRF52840 to connect to two other Bluefruit or BLE devices using the bleuart (AKA 'NUS') service concurrently, with the device running at both a peripheral and a central at the same time.

This dual role example acts as a BLE bridge that sits between a central and a peripheral forwarding bleuart messages back and forth, as shown in the image below:

Server & Client Service Setup

Since the Bluefruit device will act as both a central and a peripheral, we will need to declare both server and client instance of the bleuart helper class:

```
// Peripheral uart service
BLEUart bleuart;

// Central uart client
BLEClientUart clientUart;
```

Before we can configure client services, `Bluefruit.begin()` must be called with at least 1 for the number of concurrent connection for both peripheral and central mode:

```
// Initialize Bluefruit with max concurrent connections as Peripheral = 1, Central
= 1
Bluefruit.begin(1, 1);
```

After this, client services must be initialized by calling their `begin()` function, followed by any callbacks that you wish to wire up as well:

```
// Configure and Start BLE Uart Service
bleuart.begin();
bleuart.setRxCallback(prph_bleuart_rx_callback);

// Init BLE Central Uart Serivce
clientUart.begin();
clientUart.setRxCallback(cent_bleuart_rx_callback);
```

We are then ready to forward data from central to peripheral and vice versa using callbacks:

```
void cent_bleuart_rx_callback(BLEClientUart&amp;amp; cent_uart)
{
  char str[20+1] = { 0 };
  cent_uart.read(str, 20);

  Serial.print("[Cent] RX: ");
  Serial.println(str);

  if ( bleuart.notifyEnabled() )
  {
    // Forward data from our peripheral to Mobile
    bleuart.print( str );
  }else
  {
    // response with no prph message
    clientUart.println("[Cent] Peripheral role not connected");
  }
}

void prph_bleuart_rx_callback(void)
{
  // Forward data from Mobile to our peripheral
  char str[20+1] = { 0 };
  bleuart.read(str, 20);

  Serial.print("[Prph] RX: ");
  Serial.println(str);

  if ( clientUart.discovered() )
  {
    clientUart.print(str);
  }else
```

```
  {
    bleuart.println("[Prph] Central role not connected");
  }
}
```

Peripheral Role

The first thing to do for the peripheral part of our code is to setup the **connect callback**, which fires when a connection is established/disconnected with the central. Alternatively you could poll the connection status with connected(), but callbacks helps to simplify the code significantly:

```
// Callbacks for Peripheral
Bluefruit.setConnectCallback(prph_connect_callback);
Bluefruit.setDisconnectCallback(prph_disconnect_callback);
```

Central Role

Next we setup the Central mode **connect callback**, which fires when a connection is established/disconnected with a peripheral device:

```
// Callbacks for Central
Bluefruit.Central.setConnectCallback(cent_connect_callback);
Bluefruit.Central.setDisconnectCallback(cent_disconnect_callback);
```

Advertising and Scanner

It is possible to start both the scanner and advertising at the same time so that we can discover and be discovered by other BLE devices. For the scanner, we use a filter that only fires the callback if a specific UUID is found in the advertising data of the peer device:

```
/* Start Central Scanning
 * - Enable auto scan if disconnected
 * - Interval = 100 ms, window = 80 ms
 * - Filter only accept bleuart service
 * - Don't use active scan
 * - Start(timeout) with timeout = 0 will scan forever (until connected)
 */
Bluefruit.Scanner.setRxCallback(scan_callback);
Bluefruit.Scanner.restartOnDisconnect(true);
Bluefruit.Scanner.setInterval(160, 80); // in unit of 0.625 ms
Bluefruit.Scanner.filterUuid(bleuart.uuid);
Bluefruit.Scanner.useActiveScan(false);
Bluefruit.Scanner.start(0);                   // 0 = Don't stop scanning after n
seconds

// Advertising packet
```

```
Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
Bluefruit.Advertising.addTxPower();

// Include bleuart 128-bit uuid
Bluefruit.Advertising.addService(bleuart);

// Secondary Scan Response packet (optional)
// Since there is no room for 'Name' in Advertising packet
Bluefruit.ScanResponse.addName();

/* Start Advertising
 * - Enable auto advertising if disconnected
 * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
 * - Timeout for fast mode is 30 seconds
 * - Start(timeout) with timeout = 0 will advertise forever (until connected)
 *
 * For recommended advertising interval
 * https://developer.apple.com/library/content/qa/qa1931/_index.html
 */
Bluefruit.Advertising.restartOnDisconnect(true);
Bluefruit.Advertising.setInterval(32, 244);    // in unit of 0.625 ms
Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising after
n seconds
```

Full Sample Code

The full sample code for this example can be seen below:

```
/*********************************************************************
 This is an example for our nRF52 based Bluefruit LE modules

 Pick one up today in the adafruit shop!

 Adafruit invests time and resources providing this open source code,
 please support Adafruit and open-source hardware by purchasing
 products from Adafruit!

 MIT license, check LICENSE for more information
 All text above, and the splash screen below must be included in
 any redistribution
*********************************************************************/

/*
 * This sketch demonstrate how to run both Central and Peripheral roles
 * at the same time. It will act as a relay between an central (mobile)
 * to another peripheral using bleuart service.
 *
 * Mobile <--> DualRole <--> peripheral Ble Uart
 */
#include <bluefruit.h>

// OTA DFU service
BLEDfu bledfu;

// Peripheral uart service
BLEUart bleuart;

// Central uart client
BLEClientUart clientUart;

void setup()
{
  Serial.begin(115200);
```

```
  while ( !Serial ) delay(10);   // for nrf52840 with native usb

  Serial.println("Bluefruit52 Dual Role BLEUART Example");
  Serial.println("-------------------------------------\n");

  // Initialize Bluefruit with max concurrent connections as Peripheral = 1,
Central = 1
  // SRAM usage required by SoftDevice will increase with number of connections
  Bluefruit.begin(1, 1);
  Bluefruit.setTxPower(4);    // Check bluefruit.h for supported values

  // Callbacks for Peripheral
  Bluefruit.Periph.setConnectCallback(prph_connect_callback);
  Bluefruit.Periph.setDisconnectCallback(prph_disconnect_callback);

  // Callbacks for Central
  Bluefruit.Central.setConnectCallback(cent_connect_callback);
  Bluefruit.Central.setDisconnectCallback(cent_disconnect_callback);

  // To be consistent OTA DFU should be added first if it exists
  bledfu.begin();

  // Configure and Start BLE Uart Service
  bleuart.begin();
  bleuart.setRxCallback(prph_bleuart_rx_callback);

  // Init BLE Central Uart Serivce
  clientUart.begin();
  clientUart.setRxCallback(cent_bleuart_rx_callback);


  /* Start Central Scanning
   * - Enable auto scan if disconnected
   * - Interval = 100 ms, window = 80 ms
   * - Filter only accept bleuart service
   * - Don't use active scan
   * - Start(timeout) with timeout = 0 will scan forever (until connected)
   */
  Bluefruit.Scanner.setRxCallback(scan_callback);
  Bluefruit.Scanner.restartOnDisconnect(true);
  Bluefruit.Scanner.setInterval(160, 80); // in unit of 0.625 ms
  Bluefruit.Scanner.filterUuid(bleuart.uuid);
  Bluefruit.Scanner.useActiveScan(false);
  Bluefruit.Scanner.start(0);                   // 0 = Don't stop scanning after n
seconds

  // Set up and start advertising
  startAdv();
}

void startAdv(void)
{
  // Advertising packet
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
  Bluefruit.Advertising.addTxPower();

  // Include bleuart 128-bit uuid
  Bluefruit.Advertising.addService(bleuart);

  // Secondary Scan Response packet (optional)
  // Since there is no room for 'Name' in Advertising packet
  Bluefruit.ScanResponse.addName();

  /* Start Advertising
   * - Enable auto advertising if disconnected
   * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
   * - Timeout for fast mode is 30 seconds
   * - Start(timeout) with timeout = 0 will advertise forever (until connected)
   *
```

```
   * For recommended advertising interval
   * https://developer.apple.com/library/content/qa/qa1931/_index.html
   */
  Bluefruit.Advertising.restartOnDisconnect(true);
  Bluefruit.Advertising.setInterval(32, 244);    // in unit of 0.625 ms
  Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
  Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising
after n seconds
}

void loop()
{
  // do nothing, all the work is done in callback
}

/*------------------------------------------------------------------*/
/* Peripheral
 *------------------------------------------------------------------*/
void prph_connect_callback(uint16_t conn_handle)
{
  // Get the reference to current connection
  BLEConnection* connection = Bluefruit.Connection(conn_handle);

  char peer_name[32] = { 0 };
  connection->getPeerName(peer_name, sizeof(peer_name));

  Serial.print("[Prph] Connected to ");
  Serial.println(peer_name);
}

void prph_disconnect_callback(uint16_t conn_handle, uint8_t reason)
{
  (void) conn_handle;
  (void) reason;

  Serial.println();
  Serial.println("[Prph] Disconnected");
}

void prph_bleuart_rx_callback(uint16_t conn_handle)
{
  (void) conn_handle;

  // Forward data from Mobile to our peripheral
  char str[20+1] = { 0 };
  bleuart.read(str, 20);

  Serial.print("[Prph] RX: ");
  Serial.println(str);

  if ( clientUart.discovered() )
  {
    clientUart.print(str);
  }else
  {
    bleuart.println("[Prph] Central role not connected");
  }
}

/*------------------------------------------------------------------*/
/* Central
 *------------------------------------------------------------------*/
void scan_callback(ble_gap_evt_adv_report_t* report)
{
  // Since we configure the scanner with filterUuid()
  // Scan callback only invoked for device with bleuart service advertised
  // Connect to the device with bleuart service in advertising packet
  Bluefruit.Central.connect(report);
}
```

```
void cent_connect_callback(uint16_t conn_handle)
{
  // Get the reference to current connection
  BLEConnection* connection = Bluefruit.Connection(conn_handle);

  char peer_name[32] = { 0 };
  connection->getPeerName(peer_name, sizeof(peer_name));

  Serial.print("[Cent] Connected to ");
  Serial.println(peer_name);;

  if ( clientUart.discover(conn_handle) )
  {
    // Enable TXD's notify
    clientUart.enableTXD();
  }else
  {
    // disconnect since we couldn't find bleuart service
    Bluefruit.disconnect(conn_handle);
  }
}

void cent_disconnect_callback(uint16_t conn_handle, uint8_t reason)
{
  (void) conn_handle;
  (void) reason;

  Serial.println("[Cent] Disconnected");
}

/**
 * Callback invoked when uart received data
 * @param cent_uart Reference object to the service where the data
 * arrived. In this example it is clientUart
 */
void cent_bleuart_rx_callback(BLEClientUart& cent_uart)
{
  char str[20+1] = { 0 };
  cent_uart.read(str, 20);

  Serial.print("[Cent] RX: ");
  Serial.println(str);

  if ( bleuart.notifyEnabled() )
  {
    // Forward data from our peripheral to Mobile
    bleuart.print( str );
  }else
  {
    // response with no prph message
    clientUart.println("[Cent] Peripheral role not connected");
  }
}
```

Custom: Central HRM

The BLEClientService and BLEClientCharacteristic classes can be used to implement any custom or officially adopted BLE service of characteristic on the client side (most often is Central) using a set of basic properties and callback handlers.

The example below shows how to use these classes to implement the Heart Rate Monitor (https://adafru.it/vaO) service, as defined by the Bluetooth SIG. To run this example, you will need an extra nRF52 running peripheral HRM sketch (https://adafru.it/Cnf)

HRM Service Definition

UUID: 0x180D (https://adafru.it/vaO)

Only the first characteristic is mandatory, but we will also implement the optional **Body Sensor Location** characteristic. Heart Rate Control Point won't be used in this example to keep things simple.

Implementing the HRM Service and Characteristics

The core service and the first two characteristics can be implemented with the following code:

First, define the BLEService and BLECharacteristic variables that will be used in your project:

```
/* HRM Service Definitions
 * Heart Rate Monitor Service:  0x180D
 * Heart Rate Measurement Char: 0x2A37 (Mandatory)
 * Body Sensor Location Char:   0x2A38 (Optional)
 */

BLEClientService        hrms(UUID16_SVC_HEART_RATE);
BLEClientCharacteristic hrmc(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLEClientCharacteristic bslc(UUID16_CHR_BODY_SENSOR_LOCATION);
```

Then you need to initialize those variables by calling their begin().

```
// Initialize HRM client
hrms.begin();

// Initialize client characteristics of HRM.
// Note: Client Char will be added to the last service that is begin()ed.
bslc.begin();

// set up callback for receiving measurement
hrmc.setNotifyCallback(hrm_notify_callback);
hrmc.begin();
```

Client Service + Characteristic Code Analysis

1. The first thing to do is to call **.begin()** on the BLEClientService (**hrms** above). Since the UUID is set in the object declaration at the top of the sketch, there is normally nothing else to do with the BLEClientService instance.

You MUST call .begin() on the BLEClientService before adding any BLEClientCharacteristics. Any BLEClientCharacteristic will automatically be added to the last BLEClientService that was `begin()'ed!

2. Since **Heart Rate Measurement** characteristic (**clientMeasurement** above) is notifiable. You need to set up callback for it

- '`hrmc.setNotifyCallback(hrm_notify_callback);`' This sets the callback that will be fired when we receive a Notify message from peripheral. This is needed to handle notifiable characteristic since callback allow us to response to the message in timely manner. For this example is just simply printing out value to Serial.
- '`hrmc.begin();`' Once all of the properties have been set, you must call **.begin()** which will add the characteristic definition to the last BLEClientService that was '.begin()ed'.

Note for characteristic that does not support notify e.g body sensor location , we can simply use .read() to retrieve its value.

3. Next, we can start to scan and connect to peripheral that advertises HRM service. Once connected, we need to go through peripheral GATT table to find out the Gatt handle for our interest. In this example they are handle for **hrms**, **hrmc** and **bslc.** This looking up process for interested service/characteristic is called **Discovery.**

Note: Gatt handle (or just handle) is required to perform any operations at all such as read, write, enable notify. It is required that a client characteristic must be discovered before we could doing anything with it.

The service should be discovered before we could discover its characteristic. This can be done by calling `hrms.discover(conn_handle)` . Where conn_handle is the connection ID i.e peripheral that we want to discover since it is possible for Bluefruit nRF52 to connect to multiple peripherals concurrently. If the service is found, the function will return true, otherwise false.

```
// Connect Callback Part 1
void connect_callback(uint16_t conn_handle)
{
  Serial.println("Connected");
  Serial.print("Discovering HRM Service ... ");

  // If HRM is not found, disconnect and return
  if ( !hrms.discover(conn_handle) )
  {
    Serial.println("Found NONE");

    // disconect since we couldn't find HRM service
    Bluefruit.Central.disconnect(conn_handle);

    return;
  }

  // Once HRM service is found, we continue to discover its characteristic
  Serial.println("Found it");

  ............
}
```

4. Afterwards, we continue to discover all the interested characteristics within the service by calling `.discover()`. The function return true if characteristics is found, and false otherwise. You could also check with `.discovered()` function. A service could contain more characteristics but we don't need to discover them all, only those that we want to interact with.

Advanced: Alternatively, you could discover all the interested characteristics of a service within a function call by using `Bluefruit.Discovery.discoverCharacteristic()` (not used in the example). The API can take up to 5 characteristics, if you need more, the variant with passing array of characteristics is also available. The function will return the number of characteristic it found.

Note: when a characteristic is discovered by above API, all necessarily meta data such as handles, properties (read,write, notify etc ...), cccd handle will be updated automatically. You can then use BLEClientCharacteristic (https://adafru.it/Cng) API such as read(), write(), enableNotify() on it provided that its properties support such as operation.

```
// Connect Callback Part 2
void connect_callback(uint16_t conn_handle)
{
  Serial.print("Discovering Measurement characteristic ... ");
  if ( !hrmc.discover() )
  {
    // Measurement chr is mandatory, if it is not found (valid), then disconnect
    Serial.println("not found !!!");
    Serial.println("Measurement characteristic is mandatory but not found");
    Bluefruit.Central.disconnect(conn_handle);
    return;
  }
  Serial.println("Found it");
```

```
  // Measurement is found, continue to look for option Body Sensor Location
  // https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
  // Body Sensor Location is optional, print out the location in text if present
  Serial.print("Discovering Body Sensor Location characteristic ... ");
  if ( bslc.discover() )
  {
    Serial.println("Found it");

    // Body sensor location value is 8 bit
    const char* body_str[] = { "Other", "Chest", "Wrist", "Finger", "Hand", "Ear
Lobe", "Foot" };

    // Read 8-bit BSLC value from peripheral
    uint8_t loc_value = bslc.read8();

    Serial.print("Body Location Sensor: ");
    Serial.println(body_str[loc_value]);
  }else
  {
    Serial.println("Found NONE");
  }

  ..............
}
```

5. Once hrmc is discovered, you should enable its notification by calling `hrmc.enableNotify()`. If this succeeded (return true), peripheral can now send data to us using notify message. Which will trigger the callback that we setup earlier to handle incoming data.

```
// Connect Callback Part 3
void connect_callback(uint16_t conn_handle)
{
  .......

  // Reaching here means we are ready to go, let's enable notification on
measurement chr
  if ( hrmc.enableNotify() )
  {
    Serial.println("Ready to receive HRM Measurement value");
  }else
  {
    Serial.println("Couldn't enable notify for HRM Measurement. Increase DEBUG
LEVEL for troubleshooting");
  }
}
```

```
/**
 * Hooked callback that triggered when a measurement value is sent from peripheral
 * @param chr   Pointer to client characteristic that even occurred,
 *              in this example it should be hrmc
 * @param data  Pointer to received data
 * @param len   Length of received data
 */
void hrm_notify_callback(BLEClientCharacteristic* chr, uint8_t* data, uint16_t len)
{
  // https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
  // Measurement contains of control byte0 and measurement (8 or 16 bit) + optional
field
  // if byte0's bit0 is 0 --&gt; measurement is 8 bit, otherwise 16 bit.
```

```
  Serial.print("HRM Measurement: ");

  if ( data[0] &amp; bit(0) )
  {
    uint16_t value;
    memcpy(&amp;value, data+1, 2);

    Serial.println(value);
  }
  else
  {
    Serial.println(data[1]);
  }
}
```

Full Sample Code

The full sample code for this example can be seen below:

```
/*********************************************************************
 This is an example for our nRF52 based Bluefruit LE modules

 Pick one up today in the adafruit shop!

 Adafruit invests time and resources providing this open source code,
 please support Adafruit and open-source hardware by purchasing
 products from Adafruit!

 MIT license, check LICENSE for more information
 All text above, and the splash screen below must be included in
 any redistribution
*********************************************************************/
#include <bluefruit.h>

/* HRM Service Definitions
 * Heart Rate Monitor Service:  0x180D
 * Heart Rate Measurement Char: 0x2A37
 * Body Sensor Location Char:   0x2A38
 */
BLEService        hrms = BLEService(UUID16_SVC_HEART_RATE);
BLECharacteristic hrmc = BLECharacteristic(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLECharacteristic bslc = BLECharacteristic(UUID16_CHR_BODY_SENSOR_LOCATION);

BLEDis bledis;    // DIS (Device Information Service) helper class instance
BLEBas blebas;    // BAS (Battery Service) helper class instance

uint8_t  bps = 0;

void setup()
{
  Serial.begin(115200);
  while ( !Serial ) delay(10);   // for nrf52840 with native usb

  Serial.println("Bluefruit52 HRM Example");
  Serial.println("-----------------------\n");

  // Initialise the Bluefruit module
  Serial.println("Initialise the Bluefruit nRF52 module");
  Bluefruit.begin();

  // Set the connect/disconnect callback handlers
  Bluefruit.Periph.setConnectCallback(connect_callback);
  Bluefruit.Periph.setDisconnectCallback(disconnect_callback);
```

```
  // Configure and Start the Device Information Service
  Serial.println("Configuring the Device Information Service");
  bledis.setManufacturer("Adafruit Industries");
  bledis.setModel("Bluefruit Feather52");
  bledis.begin();

  // Start the BLE Battery Service and set it to 100%
  Serial.println("Configuring the Battery Service");
  blebas.begin();
  blebas.write(100);

  // Setup the Heart Rate Monitor service using
  // BLEService and BLECharacteristic classes
  Serial.println("Configuring the Heart Rate Monitor Service");
  setupHRM();

  // Setup the advertising packet(s)
  Serial.println("Setting up the advertising payload(s)");
  startAdv();

  Serial.println("Ready Player One!!!");
  Serial.println("\nAdvertising");
}

void startAdv(void)
{
  // Advertising packet
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
  Bluefruit.Advertising.addTxPower();

  // Include HRM Service UUID
  Bluefruit.Advertising.addService(hrms);

  // Include Name
  Bluefruit.Advertising.addName();

  /* Start Advertising
   * - Enable auto advertising if disconnected
   * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
   * - Timeout for fast mode is 30 seconds
   * - Start(timeout) with timeout = 0 will advertise forever (until connected)
   *
   * For recommended advertising interval
   * https://developer.apple.com/library/content/qa/qa1931/_index.html
   */
  Bluefruit.Advertising.restartOnDisconnect(true);
  Bluefruit.Advertising.setInterval(32, 244);    // in unit of 0.625 ms
  Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
  Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising
after n seconds
}

void setupHRM(void)
{
  // Configure the Heart Rate Monitor service
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.service.heart_rate.xml
  // Supported Characteristics:
  // Name                         UUID    Requirement Properties
  // ---------------------------- ------  ----------- ----------
  // Heart Rate Measurement       0x2A37  Mandatory   Notify
  // Body Sensor Location         0x2A38  Optional    Read
  // Heart Rate Control Point     0x2A39  Conditional Write       <-- Not used here
  hrms.begin();

  // Note: You must call .begin() on the BLEService before calling .begin() on
  // any characteristic(s) within that service definition.. Calling .begin() on
  // a BLECharacteristic will cause it to be added to the last BLEService that
```

```
  // was 'begin()'ed!

  // Configure the Heart Rate Measurement characteristic
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
  // Properties = Notify
  // Min Len    = 1
  // Max Len    = 8
  //    B0      = UINT8  - Flag (MANDATORY)
  //      b5:7  = Reserved
  //      b4    = RR-Internal (0 = Not present, 1 = Present)
  //      b3    = Energy expended status (0 = Not present, 1 = Present)
  //      b1:2  = Sensor contact status (0+1 = Not supported, 2 = Supported but
contact not detected, 3 = Supported and detected)
  //      b0    = Value format (0 = UINT8, 1 = UINT16)
  //    B1      = UINT8  - 8-bit heart rate measurement value in BPM
  //    B2:3    = UINT16 - 16-bit heart rate measurement value in BPM
  //    B4:5    = UINT16 - Energy expended in joules
  //    B6:7    = UINT16 - RR Internal (1/1024 second resolution)
  hrmc.setProperties(CHR_PROPS_NOTIFY);
  hrmc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  hrmc.setFixedLen(2);
  hrmc.setCccdWriteCallback(cccd_callback);  // Optionally capture CCCD updates
  hrmc.begin();
  uint8_t hrmdata[2] = { 0b00000110, 0x40 }; // Set the characteristic to use 8-bit
values, with the sensor connected and detected
  hrmc.write(hrmdata, 2);

  // Configure the Body Sensor Location characteristic
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
  // Properties = Read
  // Min Len    = 1
  // Max Len    = 1
  //    B0      = UINT8 - Body Sensor Location
  //      0     = Other
  //      1     = Chest
  //      2     = Wrist
  //      3     = Finger
  //      4     = Hand
  //      5     = Ear Lobe
  //      6     = Foot
  //      7:255 = Reserved
  bslc.setProperties(CHR_PROPS_READ);
  bslc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  bslc.setFixedLen(1);
  bslc.begin();
  bslc.write8(2);    // Set the characteristic to 'Wrist' (2)
}

void connect_callback(uint16_t conn_handle)
{
  // Get the reference to current connection
  BLEConnection* connection = Bluefruit.Connection(conn_handle);

  char central_name[32] = { 0 };
  connection->getPeerName(central_name, sizeof(central_name));

  Serial.print("Connected to ");
  Serial.println(central_name);
}

/**
 * Callback invoked when a connection is dropped
 * @param conn_handle connection where this event happens
 * @param reason is a BLE_HCI_STATUS_CODE which can be found in ble_hci.h
 */
void disconnect_callback(uint16_t conn_handle, uint8_t reason)
{
```

```
  (void) conn_handle;
  (void) reason;

  Serial.print("Disconnected, reason = 0x"); Serial.println(reason, HEX);
  Serial.println("Advertising!");
}

void cccd_callback(uint16_t conn_hdl, BLECharacteristic* chr, uint16_t cccd_value)
{
    // Display the raw request packet
    Serial.print("CCCD Updated: ");
    //Serial.printBuffer(request->data, request->len);
    Serial.print(cccd_value);
    Serial.println("");

    // Check the characteristic this CCCD update is associated with in case
    // this handler is used for multiple CCCD records.
    if (chr->uuid == hrmc.uuid) {
        if (chr->notifyEnabled(conn_hdl)) {
            Serial.println("Heart Rate Measurement 'Notify' enabled");
        } else {
            Serial.println("Heart Rate Measurement 'Notify' disabled");
        }
    }
}

void loop()
{
  digitalToggle(LED_RED);

  if ( Bluefruit.connected() ) {
    uint8_t hrmdata[2] = { 0b00000110, bps++ };           // Sensor connected,
increment BPS value

    // Note: We use .notify instead of .write!
    // If it is connected but CCCD is not enabled
    // The characteristic's value is still updated although notification is not sent
    if ( hrmc.notify(hrmdata, sizeof(hrmdata)) ){
      Serial.print("Heart Rate Measurement updated to: "); Serial.println(bps);
    }else{
      Serial.println("ERROR: Notify not set in the CCCD or not connected!");
    }
  }

  // Only send update once per second
  delay(1000);
}
```

Arduino Bluefruit nRF52 API

The Adafruit nRF52 core defines a number of custom classes that aim to make it easy to work with BLE in your projects.

The key classes are listed below, and examined in more detail elsewhere in this learning guide:

- **AdafruitBluefruit** is the main entry point to the Adafruit Bluefruit nRF52 API. This class exposes a number of essential functions and classes, such as advertising,

the list of GATT services and characteristics defined on your device, and connection status.

- **BLEService** is a wrapper class for BLE GATT service records, and can be used to define custom service definitions, or acts as the base class for any service helper classes.
- **BLECharacteristic** is a wrapper class for a BLE GATT characteristic record, and can be used to define custom characteristics, or acts as the base class for any characteristic helper classes.
- **BLEDis** is a helper class for the DIS or 'Device Information Service'.
- **BLEUart** is a helper class for the NUS or 'Nordic UART Service'.
- **BLEBeacon** is a helper class to configure your nRF52 as a beacon using the advertising packet to send out properly formatted beacon data.
- **BLEMidi** is a helper class to work with MIDI data over BLE.
- **BLEHidAdafruit** is a helper class to emulate an HID mouse or keyboard over BLE.

Details on each of these helper classes are found further in this learning guide.

AdafruitBluefruit

Adafruit's nRF52 BSP codebase is still undergoing active development based on customer feedback and testing. As such, the class documentation here may be incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

This base class is the main entry point to the Adafruit Bluefruit nRF52 API, and exposes most of the helper classes and functions that you use to configure your device.

API

AdafruitBluefruit has the following public API:

```
// Constructor
AdafruitBluefruit(void);

/*------------------------------------------------------------------*/
/* Lower Level Classes (Bluefruit.Advertising.*, etc.)
 *------------------------------------------------------------------*/
BLEGap             Gap;
BLEGatt            Gatt;

BLEAdvertising     Advertising;
```

```
BLEAdvertisingData ScanResponse;
BLEScanner         Scanner;
BLECentral         Central;
BLEDiscovery       Discovery;

/*------------------------------------------------------------------*/
/* SoftDevice Configure Functions, must call before begin().
 * These function affect the SRAM consumed by SoftDevice.
 *------------------------------------------------------------------*/
void     configServiceChanged (bool     changed);
void     configUuid128Count   (uint8_t  uuid128_max);
void     configAttrTableSize  (uint32_t attr_table_size);

// Config Bandwidth for connections
void     configPrphConn        (uint16_t mtu_max, uint8_t event_len, uint8_t
hvn_qsize, uint8_t wrcmd_qsize);
void     configCentralConn     (uint16_t mtu_max, uint8_t event_len, uint8_t
hvn_qsize, uint8_t wrcmd_qsize);

// Convenient function to config connection
void     configPrphBandwidth   (uint8_t bw);
void     configCentralBandwidth(uint8_t bw);

err_t    begin(uint8_t prph_count = 1, uint8_t central_count = 0);

/*------------------------------------------------------------------*/
/* General Functions
 *------------------------------------------------------------------*/
void     setName            (const char* str);
uint8_t  getName            (char* name, uint16_t bufsize);

bool     setTxPower         (int8_t power);
int8_t   getTxPower         (void);

bool     setApperance       (uint16_t appear);
uint16_t getApperance       (void);

void     autoConnLed        (bool enabled);
void     setConnLedInterval (uint32_t ms);

/*------------------------------------------------------------------*/
/* GAP, Connections and Bonding
 *------------------------------------------------------------------*/
bool     connected         (void);
bool     disconnect        (void);

bool     setConnInterval   (uint16_t min, uint16_t max);
bool     setConnIntervalMS (uint16_t min_ms, uint16_t max_ms);

uint16_t connHandle        (void);
bool     connPaired        (void);
uint16_t connInterval      (void);

bool     requestPairing    (void);
void     clearBonds        (void);

ble_gap_addr_t getPeerAddr (void);
uint8_t        getPeerAddr (uint8_t addr[6]);

void     printInfo(void);

/*------------------------------------------------------------------*/
/* Callbacks
 *------------------------------------------------------------------*/
void setConnectCallback   ( BLEGap::connect_callback_t    fp);
void setDisconnectCallback( BLEGap::disconnect_callback_t fp);
```

These functions are generally available via '**Bluefruit.***'. For example, to check the connection status in your sketch you could run '`if (Bluefruit.connected()) { ... }`'.

Examples

For examples of how to work with the parent **Bluefruit** class, see the **Examples** section later in this guide. It's better to examine this class in the context of a real world use case.

You can also browse the latest example code online via Github:

Browse the latest example code on Github

https://adafru.it/vaK

BLEGap

This page is a work in progress as the API is changing as we migrate to S132v5 (nRF52832) and S140 (nRF52840) and add better Central mode support.

This GAP API for Bluefruit is accessible via `Bluefruit.Gap.***` and has the following public functions:

```
typedef void (*connect_callback_t    ) (uint16_t conn_handle);
typedef void (*disconnect_callback_t ) (uint16_t conn_handle, uint8_t reason);

uint8_t        getAddr               (uint8_t mac[6]);
bool           setAddr               (uint8_t mac[6], uint8_t type);

bool           connected             (uint16_t conn_handle);

uint8_t        getRole               (uint16_t conn_handle);

uint8_t        getPeerAddr           (uint16_t conn_handle, uint8_t addr[6]);
ble_gap_addr_t getPeerAddr           (uint16_t conn_handle);
uint16_t       getPeerName           (uint16_t conn_handle, char* buf, uint16_t
bufsize);

uint16_t       getMTU                (uint16_t conn_handle);
uint16_t       getMaxMtuByConnCfg    (uint8_t conn_cfg);
uint16_t       getMaxMtu             (uint8_t conn_handle);
```

BLEAdvertising

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

'Advertising' is what makes your Bluetooth Low Energy devices visible to other devices in listening range. The radio sends out specially formatter advertising packets that contain information like the device name, whether you can connect to the device (or if it only advertises), etc.

You can also include custom data in the advertising packet, which is essential how beacons work.

The BLEAdvertisingData and BLEAdvertising classes exposes a number of helper functions to make it easier to create well-formatted advertising packets, as well as to use the **Scan Response** option, which is an optional secondary advertising packet that can be requested by a Central device. (This gives you another 27 bytes of advertising data, but isn't sent out automatically like the main advertising packet.).

This two advertising packets are accessible via the parent AdafruitBluefruit class, calling '`Bluefruit.Advertising.*`' and '`Bluefruit.ScanResponse.*`' from your user sketches.

For examples of using these helper classes, see any of the **examples** later on in this guide, since all devices will advertise as part of the startup process.

API

The **BLEAdvertisingData** class has the following public API:

```
/*------------- Adv Data -------------*/
bool addData(uint8_t type, const void* data, uint8_t len);
bool addFlags(uint8_t flags);
bool addTxPower(void);
bool addName(void);
bool addAppearance(uint16_t appearance);
bool addManufacturerData(const void* data, uint8_t count);

/*------------- UUID -------------*/
bool addUuid(BLEUuid bleuuid);
bool addUuid(BLEUuid bleuuid1, BLEUuid bleuuid2);
bool addUuid(BLEUuid bleuuid1, BLEUuid bleuuid2, BLEUuid bleuuid3);
bool addUuid(BLEUuid bleuuid1, BLEUuid bleuuid2, BLEUuid bleuuid3, BLEUuid
```

```
bleuuid4);

bool addUuid(BLEUuid bleuuid[], uint8_t count);

/*------------- Service -------------*/
bool addService(BLEService&amp; service);
bool addService(BLEService&amp; service1, BLEService&amp; service2);
bool addService(BLEService&amp; service1, BLEService&amp; service2, BLEService&amp;
service3);
bool addService(BLEService&amp; service1, BLEService&amp; service2, BLEService&amp;
service3, BLEService&amp; service4);

/*------------- Client Service -------------*/
bool addService(BLEClientService&amp; service);

// Functions to work with the raw advertising packet
uint8_t  count(void);
uint8_t* getData(void);
bool     setData(const uint8_t* data, uint8_t count);
void     clearData(void);

bool     setData(Advertisable&amp; adv_able) { return adv_able.setAdv(*this); }
```

In addition to API from BLEAdvertisingData, The **BLEAdvertising** class also has functions that dictate the behavior of advertising such as slow/fast timeout, adv intervals, and callbacks etc...

```
typedef void (*stop_callback_t) (void);
typedef void (*slow_callback_t) (void);

void setType(uint8_t adv_type);
void setFastTimeout(uint16_t sec);

void setSlowCallback(slow_callback_t fp);
void setStopCallback(stop_callback_t fp);

void setInterval  (uint16_t fast, uint16_t slow);
void setIntervalMS(uint16_t fast, uint16_t slow);

uint16_t getInterval(void);

bool setBeacon(BLEBeacon&amp; beacon);
bool setBeacon(EddyStoneUrl&amp; eddy_url);

bool isRunning(void);

void restartOnDisconnect(bool enable);
bool start(uint16_t timeout = 0);
bool stop (void);
```

Related Information

- Generic Access Profile (https://adafru.it/vaL): This page contains the official list of assigned numbers for the 'Data' type field. Data is inserted into the advertising packet by supplying a valid 'data' type, optionally followed by a properly formatted payload corresponding to the selected value.

Example

For practical example code, see the **Examples** section later on in this guide. The snippet below is provided for illustration purposes, but advertising should be examined in the context of a real use case since it varies from one setup to the next!

```
void setup(void)
{
  // Other startup code here
  // ...

  // Set up Advertising Packet
  setupAdv();

  // Start Advertising
  Bluefruit.Advertising.start();
}

void startAdv(void)
{
  // Advertising packet
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
  Bluefruit.Advertising.addTxPower();

  // Include bleuart 128-bit uuid
  Bluefruit.Advertising.addService(bleuart);

  // Secondary Scan Response packet (optional)
  // Since there is no room for 'Name' in Advertising packet
  Bluefruit.ScanResponse.addName();

  /* Start Advertising
   * - Enable auto advertising if disconnected
   * - Interval:  fast mode = 20 ms, slow mode = 152.5 ms
   * - Timeout for fast mode is 30 seconds
   * - Start(timeout) with timeout = 0 will advertise forever (until connected)
   *
   * For recommended advertising interval
   * https://developer.apple.com/library/content/qa/qa1931/_index.html
   */
  Bluefruit.Advertising.restartOnDisconnect(true);
  Bluefruit.Advertising.setInterval(32, 244);    // in unit of 0.625 ms
  Bluefruit.Advertising.setFastTimeout(30);      // number of seconds in fast mode
  Bluefruit.Advertising.start(0);                // 0 = Don't stop advertising
after n seconds
}
```

BLEScanner

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

This documentation is based on BSP 0.7.0 and higher. Please make sure you have an up to date version before using the code below.

The BLEScanner class is used in **Central Mode**, and facilitates scanning for BLE peripherals in range and parsing the advertising data that is being sent out by the peripherals.

The BLEScanner class is normally accessed via the Bluefruit class (instantiated at startup), as shown below:

```
/* Start Central Scanning
 * - Enable auto scan if disconnected
 * - Filter for devices with a min RSSI of -80 dBm
 * - Interval = 100 ms, window = 50 ms
 * - Use active scan (requests the optional scan response packet)
 * - Start(0) = will scan forever since no timeout is given
 */
Bluefruit.Scanner.setRxCallback(scan_callback);
Bluefruit.Scanner.restartOnDisconnect(true);
Bluefruit.Scanner.filterRssi(-80);            // Only invoke callback when RSSI
&gt;= -80 dBm
Bluefruit.Scanner.setInterval(160, 80);       // in units of 0.625 ms
Bluefruit.Scanner.useActiveScan(true);        // Request scan response data
Bluefruit.Scanner.start(0);                   // 0 = Don't stop scanning after n
seconds
```

API

BLEScanner has the following public API:

```
typedef void (*rx_callback_t) (ble_gap_evt_adv_report_t*);
typedef void (*stop_callback_t) (void);

BLEScanner(void);

ble_gap_scan_params_t* getParams(void);
bool isRunning(void);

void useActiveScan(bool enable);
void setInterval(uint16_t interval, uint16_t window);
void setIntervalMS(uint16_t interval, uint16_t window);
void restartOnDisconnect(bool enable);

void filterRssi(int8_t min_rssi);
void filterMSD(uint16_t manuf_id);

void filterUuid(BLEUuid ble_uuid);
void filterUuid(BLEUuid ble_uuid1, BLEUuid ble_uuid2);
void filterUuid(BLEUuid ble_uuid1, BLEUuid ble_uuid2, BLEUuid ble_uuid3);
void filterUuid(BLEUuid ble_uuid1, BLEUuid ble_uuid2, BLEUuid ble_uuid3, BLEUuid
ble_uuid4);
void filterUuid(BLEUuid ble_uuid[], uint8_t count);

void clearFilters(void);

bool start(uint16_t timeout = 0);
```

```
bool stop(void);

/*------------- Callbacks -------------*/
void setRxCallback(rx_callback_t fp);
void setStopCallback(stop_callback_t fp);

/*------------- Data Parser -------------*/
uint8_t parseReportByType(const uint8_t* scandata, uint8_t scanlen, uint8_t type,
uint8_t* buf, uint8_t bufsize = 0);
uint8_t parseReportByType(const ble_gap_evt_adv_report_t* report, uint8_t type,
uint8_t* buf, uint8_t bufsize = 0);

bool    checkReportForUuid(const ble_gap_evt_adv_report_t* report, BLEUuid
ble_uuid);
bool    checkReportForService(const ble_gap_evt_adv_report_t* report,
BLEClientService svc);
bool    checkReportForService(const ble_gap_evt_adv_report_t* report, BLEService
svc);
```

setRxCallback(rx_callback_t fp)

Whenever a valid advertising packet is detected (based on any optional filters that are applied in the BLEScanner class), a dedicated callback function (see `rx_callback_t`) will be called.

The callback function has the following signature:

NOTE: `ble_gap_evt_adv_report_t` is part of the Nordic nRF52 SD and is defined in ble_gap.h

```
void scan_callback(ble_gap_evt_adv_report_t* report)
{
  /* Display the timestamp and device address */
  if (report-&gt;scan_rsp)
  {
    /* This is a Scan Response packet */
    Serial.printf("[SR%10d] Packet received from ", millis());
  }
  else
  {
    /* This is a normal advertising packet */
    Serial.printf("[ADV%9d] Packet received from ", millis());
  }
  Serial.printBuffer(report-&gt;peer_addr.addr, 6, ':');
  Serial.print("\n");

  /* Raw buffer contents */
  Serial.printf("%14s %d bytes\n", "PAYLOAD", report-&gt;dlen);
  if (report-&gt;dlen)
  {
    Serial.printf("%15s", " ");
    Serial.printBuffer(report-&gt;data, report-&gt;dlen, '-');
    Serial.println();
  }

  /* RSSI value */
  Serial.printf("%14s %d dBm\n", "RSSI", report-&gt;rssi);

  /* Adv Type */
  Serial.printf("%14s ", "ADV TYPE");
```

```
  switch (report-&gt;type)
  {
    case BLE_GAP_ADV_TYPE_ADV_IND:
      Serial.printf("Connectable undirected\n");
      break;
    case BLE_GAP_ADV_TYPE_ADV_DIRECT_IND:
      Serial.printf("Connectable directed\n");
      break;
    case BLE_GAP_ADV_TYPE_ADV_SCAN_IND:
      Serial.printf("Scannable undirected\n");
      break;
    case BLE_GAP_ADV_TYPE_ADV_NONCONN_IND:
      Serial.printf("Non-connectable undirected\n");
      break;
  }

  /* Check for BLE UART UUID */
  if ( Bluefruit.Scanner.checkReportForUuid(report, BLEUART_UUID_SERVICE) )
  {
    Serial.printf("%14s %s\n", "BLE UART", "UUID Found!");
  }

  /* Check for DIS UUID */
  if ( Bluefruit.Scanner.checkReportForUuid(report, UUID16_SVC_DEVICE_INFORMATION) )
  {
    Serial.printf("%14s %s\n", "DIS", "UUID Found!");
  }

  Serial.println();
}
```

void useActiveScan(bool enable);

Enabling 'Active Scan' by setting the `enable` parameter to 1 will cause the device to request the optional **Scan Response** advertising packet, which is a second 31 byte advertising packet that can be used to transmit additional information.

By default active scanning is disabled, so no Scan Response packets will be received by BLEScanner unless this function is called and set to 1 before calling `Bluefruit.Scanner.start(0)`.

void filterRssi(int8_t min_rssi);
void filterMSD(uint16_t manuf_id);
void filterUuid(BLEUuid ble_uuid);
void filterUuid(BLEUuid ble_uuid1, BLEUuid ble_uuid2);
void filterUuid(BLEUuid ble_uuid1, BLEUuid ble_uuid2, BLEUuid ble_uuid3);
void filterUuid(BLEUuid ble_uuid1, BLEUuid ble_uuid2, BLEUuid ble_uuid3, BLEUuid ble_uuid4);
void filterUuid(BLEUuid ble_uuid[], uint8_t count);

Filters can be applied to BLEScanner to narrow down the data sent to the callback handler, and make processing advertising packets easier for you.

As of BSP 0.7.0 the following three filters are present:

- `filterRssi(int8_t min_rssi)` : Filters advertising results to devices with at least the specified RSSI value, which allows you to ignore devices that are too far away or whose signal is too weak. The higher the number, the strong the signal so -90 is a very weak signal, and -60 is a much stronger one.
- `filterUuid(BLEUuid ble_uuid)` : Filters advertising results to devices that advertise themselves as having the specified service UUID. If multiple UUIDs are entered, they will be filtered with boolean OR logic, meaning any single UUID present will be considered a match.
- `void filterMSD(uint16_t manuf_id)` : Fitlers advertising results to devices that contain a Manufacturer Specific Data data type, and who use the specifed Bluetooth Customer ID (`manuf_id)` . This can be useful to filter iBeacon versus Eddystone devices, for example, which both used the MSD field, or to look for custom MSD data matching your own CID.

When multiple UUIDs are added via one of the .filterUuid(...) functions, they UUIDs will be filtered using boolean 'OR' logic, meaning that the callback will fire when ANY of the specified UUIDs are detected in the advertising packet.

void clearFilters(void);

This function clears and filter values set using the functions above.

bool start(uint16_t timeout = 0);
bool stop(void);

The `.start` and `.stop` functions can be used to start and stop scanning, and should be called after all of the main parameters (timing, filters, etc.) have been set.

The `.start` function has a single parameter called timeout, which sets the number of seconds to scan for advertising packets. Setting this to '0' (the default value) will cause the device to scan forever.

Make sure you set any filters of BLEScanner parameters before calling .start!

void restartOnDisconnect(bool enable);

Setting this function to '1' will cause the scanning process to start again as soon as you disconnect from a peripheral device. The default behaviour is to automatically restart scanning on disconnect.

Examples

For an example that uses almost all of the BLEScanner and advertising API in Central mode, see central_scan_advanced.ino (https://adafru.it/y5a) in the Central examples folder.

central_scan_advanced.ino on Github

https://adafru.it/y5a

This example is only available in BSP 0.7.0 and higher!

BLEService

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

This base class is used when defining custom BLE Gatt Services, such as the various service helper classes that make up the Adafruit Bluefruit nRF52 API described here.

Unless you are implementing a custom GATT service and characteristic, you normally won't use this base class directly, and would instantiate and call a higher level helper service or characteristic included in the Bluefruit nRF52 API.

Basic Usage

There are normally only two operation required to use the BLEService class:

You need to declare and instantiate the class with an appropriate 16-bit or 128-bit UUID in the constructor:

```
BLEService myService = BLEService(0x1234);
```

You then need to call the **.begin()** method on the instance before adding any BLECharacteristics to it (via the BLECharacteristic's respective .begin() function call):

```
myService.begin();
```

Order of Operations (Important!)

One very important thing to take into consideration when working with BLEService and BLECharacteristic, is that any BLECharacteristic will automatically be added to the last BLEService that had it's `.begin()` function called. As such, you **must call yourService.begin() before adding any characteristics!**

See the example at the bottom of this page for a concrete example of how this works in practice.

API

BLEService has the following overall class structure:

This documentation may be slightly out of date as bugs are fixed, and the API develops. You should always consult the Github repo for the definitive latest code release and class definitions!

```
BLEUuid uuid;

static BLEService* lastService;

BLEService(void);
BLEService(uint16_t uuid16);
BLEService(uint8_t const  uuid128[]);

void setUuid(uint16_t uuid16);
void setUuid(uint8_t const  uuid128[]);

virtual err_t begin(void);
```

Example

The following example declares a HRM (Heart Rate Monitor) service, and assigns some characteristics to it:

Note that this example code is incomplete. For the full example open the 'custom_hrm' example that is part of the nRF52 BSP! The code below is for illustration purposes only.

```
/* HRM Service Definitions
 * Heart Rate Monitor Service:  0x180D
 * Heart Rate Measurement Char: 0x2A37
 * Body Sensor Location Char:   0x2A38
 */
BLEService        hrms = BLEService(UUID16_SVC_HEART_RATE);
BLECharacteristic hrmc = BLECharacteristic(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLECharacteristic bslc = BLECharacteristic(UUID16_CHR_BODY_SENSOR_LOCATION);

void setupHRM(void)
{
  // Configure the Heart Rate Monitor service
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.service.heart_rate.xml
  // Supported Characteristics:
  // Name                         UUID    Requirement Properties
  // ---------------------------- ------  ----------- ----------
  // Heart Rate Measurement       0x2A37  Mandatory   Notify
  // Body Sensor Location         0x2A38  Optional    Read
  // Heart Rate Control Point     0x2A39  Conditional Write       &lt;-- Not used
here
  hrms.begin();

  // Note: You must call .begin() on the BLEService before calling .begin() on
  // any characteristic(s) within that service definition.. Calling .begin() on
  // a BLECharacteristic will cause it to be added to the last BLEService that
  // was 'begin()'ed!

  // Configure the Heart Rate Measurement characteristic
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
  // Permission = Notify
  // Min Len    = 1
  // Max Len    = 8
  //    B0      = UINT8  - Flag (MANDATORY)
  //      b5:7  = Reserved
  //      b4    = RR-Internal (0 = Not present, 1 = Present)
```

```
  //      b3    = Energy expended status (0 = Not present, 1 = Present)
  //      b1:2  = Sensor contact status (0+1 = Not supported, 2 = Supported but
contact not detected, 3 = Supported and detected)
  //      b0    = Value format (0 = UINT8, 1 = UINT16)
  //    B1      = UINT8  - 8-bit heart rate measurement value in BPM
  //    B2:3    = UINT16 - 16-bit heart rate measurement value in BPM
  //    B4:5    = UINT16 - Energy expended in joules
  //    B6:7    = UINT16 - RR Internal (1/1024 second resolution)
  hrmc.setProperties(CHR_PROPS_NOTIFY);
  hrmc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  hrmc.setFixedLen(2);
  hrmc.setCccdWriteCallback(cccd_callback);  // Optionally capture CCCD updates
  hrmc.begin();
  uint8_t hrmdata[2] = { 0b00000110, 0x40 }; // Set the characteristic to use 8-bit
values, with the sensor connected and detected
  hrmc.notify(hrmdata, 2);                   // Use .notify instead of .write!

  // Configure the Body Sensor Location characteristic
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
  // Permission = Read
  // Min Len    = 1
  // Max Len    = 1
  //    B0      = UINT8 - Body Sensor Location
  //      0     = Other
  //      1     = Chest
  //      2     = Wrist
  //      3     = Finger
  //      4     = Hand
  //      5     = Ear Lobe
  //      6     = Foot
  //      7:255 = Reserved
  bslc.setProperties(CHR_PROPS_READ);
  bslc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  bslc.setFixedLen(1);
  bslc.begin();
  bslc.write8(2);    // Set the characteristic to 'Wrist' (2)
}

void cccd_callback(BLECharacteristic&amp; chr, uint16_t cccd_value)
{
    // Display the raw request packet
    Serial.print("CCCD Updated: ");
    //Serial.printBuffer(request-&gt;data, request-&gt;len);
    Serial.print(cccd_value);
    Serial.println("");

    // Check the characteristic this CCCD update is associated with in case
    // this handler is used for multiple CCCD records.
    if (chr.uuid == hrmc.uuid) {
        if (chr.notifyEnabled()) {
            Serial.println("Heart Rate Measurement 'Notify' enabled");
        } else {
            Serial.println("Heart Rate Measurement 'Notify' disabled");
        }
    }
}
```

BLECharacteristic

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is

incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

This base class is used when defining custom BLE GATT characteristics, and is used throughput the Adafruit Bluefruit nRF52 API and helper classes.

Unless you are implementing a custom GATT service and characteristic, you normally won't use this base class directly, and would instantiate and call a higher level helper service or characteristic included in the Bluefruit nRF52 API.

Basic Usage

There are two main steps to using the BLECharacteristic class.

First, you need to declare and instantiate your BLECharacteristic class with a 16-bit or 128-bit UUID:

```
BLECharacteristic myChar = BLECharacteristic(0xABCD);
```

Then you need to set the relevant properties for the characteristic, with the following values at minimum:

```
myChar.setProperties(CHR_PROPS_READ);
myChar.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
myChar.setFixedLen(1); // Alternatively .setMaxLen(uint16_t len)
myChar.begin();
```

- **.setProperties** can be set to one or more of the following macros, which correspond to a single bit in the eight bit 'properties' field for the characteristic definition:
    - CHR_PROPS_BROADCAST = bit(0),
    - CHR_PROPS_READ = bit(1),
    - CHR_PROPS_WRITE_WO_RESP = bit(2),
    - CHR_PROPS_WRITE = bit(3),
    - CHR_PROPS_NOTIFY = bit(4),
    - CHR_PROPS_INDICATE = bit(5)
- **.setPermission** sets the security level for the characteristic, where the first value sets the read permissions, and the second value sets the write permissions, where both fields can have one of the following values:
    - SECMODE_NO_ACCESS = 0x00,
    - SECMODE_OPEN = 0x11,

    - SECMODE_ENC_NO_MITM = 0x21,
    - SECMODE_ENC_WITH_MITM = 0x31,
    - SECMODE_SIGNED_NO_MITM = 0x12,
    - SECMODE_SIGNED_WITH_MITM = 0x22

- **.setFixedLen()** indicates how many bytes this characteristic has. For characteristics that use 'notify' or 'indicate' this value can be from 1..20, other characteristic types can be set from 1..512 and values >20 bytes will be sent across multiple 20 byte packets. If the characteristic has a variable len, you set the **.setMaxLen()** value to the maximum value it will hold (up to 20 bytes).
- **.begin()** will cause this characteristic to be added to the last **BLEService** that had it's **.begin()** method called.

Order of Operations (Important!)

One very important thing to take into consideration when working with BLEService and BLECharacteristic, is that any BLECharacteristic will automatically be added to the last BLEService that had it's `.begin()` function called. As such, you **must call yourService.begin() before adding any characteristics!**

See the example at the bottom of this page for a concrete example of how this works in practice.

API

BLECharacteristic has the following overall class structure:

This documentation may be slightly out of date as bugs are fixed, and the API develops. You should always consult the Github repo for the definitive latest code release and class definitions!

```
/*--------- Callback Signatures ----------*/
typedef void (*read_authorize_cb_t)  (BLECharacteristic&amp; chr,
ble_gatts_evt_read_t * request);
typedef void (*write_authorize_cb_t) (BLECharacteristic&amp; chr,
ble_gatts_evt_write_t* request);
typedef void (*write_cb_t)           (BLECharacteristic&amp; chr, uint8_t* data,
uint16_t len, uint16_t offset);
typedef void (*write_cccd_cb_t)      (BLECharacteristic&amp; chr, uint16_t value);

BLEUuid uuid;

// Constructors
BLECharacteristic(void);
BLECharacteristic(BLEUuid bleuuid);
```

```
// Destructor
virtual ~BLECharacteristic();

BLEService&amp;amp; parentService(void);

void setTempMemory(void);

/*------------- Configure -------------*/
void setUuid(BLEUuid bleuuid);
void setProperties(uint8_t prop);
void setPermission(BleSecurityMode read_perm, BleSecurityMode write_perm);
void setMaxLen(uint16_t max_len);
void setFixedLen(uint16_t fixed_len);

/*------------- Descriptors -------------*/
void setUserDescriptor(const char* descriptor); // aka user descriptor
void setReportRefDescriptor(uint8_t id, uint8_t type); // TODO refactor to use
addDescriptor()
void setPresentationFormatDescriptor(uint8_t type, int8_t exponent, uint16_t unit,
uint8_t name_space = 1, uint16_t descritpor = 0);

/*------------- Callbacks -------------*/
void setWriteCallback        (write_cb_t fp);
void setCccdWriteCallback    (write_cccd_cb_t fp);
void setReadAuthorizeCallback(read_authorize_cb_t fp);
void setWriteAuthorizeCallbak(write_authorize_cb_t fp);

virtual err_t begin(void);

// Add Descriptor function must be called right after begin()
err_t addDescriptor(BLEUuid bleuuid, void const * content, uint16_t len,
BleSecurityMode read_perm = SECMODE_OPEN, BleSecurityMode write_perm =
SECMODE_NO_ACCESS);

ble_gatts_char_handles_t handles(void);

/*------------- Write -------------*/
uint16_t write(const void* data, uint16_t len);
uint16_t write(const char* str);

uint16_t write8    (uint8_t  num);
uint16_t write16   (uint16_t num);
uint16_t write32   (uint32_t num);
uint16_t write32   (int      num);


/*------------- Read -------------*/
uint16_t read(void* buffer, uint16_t bufsize);

uint8_t  read8 (void);
uint16_t read16(void);
uint32_t read32(void);

/*------------- Notify -------------*/
uint16_t getCccd(void);

bool notifyEnabled(void);

bool notify(const void* data, uint16_t len);
bool notify(const char* str);

bool notify8    (uint8_t  num);
bool notify16   (uint16_t num);
bool notify32   (uint32_t num);
bool notify32   (int      num);

/*------------- Indicate -------------*/
bool indicateEnabled(void);
```

```
bool indicate(const void* data, uint16_t len);
bool indicate(const char* str);

bool indicate8    (uint8_t  num);
bool indicate16   (uint16_t num);
bool indicate32   (uint32_t num);
bool indicate32   (int      num);
```

Example

The following example configures an instance of the Heart Rate Monitor (HRM) Service and it's related characteristics:

Note that this example code is incomplete. For the full example open the 'custom_hrm' example that is part of the nRF52 BSP! The code below is for illustration purposes only.

```
/* HRM Service Definitions
 * Heart Rate Monitor Service:  0x180D
 * Heart Rate Measurement Char: 0x2A37
 * Body Sensor Location Char:   0x2A38
 */
BLEService        hrms = BLEService(UUID16_SVC_HEART_RATE);
BLECharacteristic hrmc = BLECharacteristic(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLECharacteristic bslc = BLECharacteristic(UUID16_CHR_BODY_SENSOR_LOCATION);

void setupHRM(void)
{
  // Configure the Heart Rate Monitor service
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.service.heart_rate.xml
  // Supported Characteristics:
  // Name                         UUID    Requirement Properties
  // ---------------------------- ------  ----------- ----------
  // Heart Rate Measurement       0x2A37  Mandatory   Notify
  // Body Sensor Location         0x2A38  Optional    Read
  // Heart Rate Control Point     0x2A39  Conditional Write       &lt;-- Not used
here
  hrms.begin();

  // Note: You must call .begin() on the BLEService before calling .begin() on
  // any characteristic(s) within that service definition.. Calling .begin() on
  // a BLECharacteristic will cause it to be added to the last BLEService that
  // was 'begin()'ed!

  // Configure the Heart Rate Measurement characteristic
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
  // Permission = Notify
  // Min Len    = 1
  // Max Len    = 8
  //    B0      = UINT8  - Flag (MANDATORY)
  //      b5:7  = Reserved
  //      b4    = RR-Internal (0 = Not present, 1 = Present)
  //      b3    = Energy expended status (0 = Not present, 1 = Present)
  //      b1:2  = Sensor contact status (0+1 = Not supported, 2 = Supported but
contact not detected, 3 = Supported and detected)
  //      b0    = Value format (0 = UINT8, 1 = UINT16)
  //    B1      = UINT8  - 8-bit heart rate measurement value in BPM
  //    B2:3    = UINT16 - 16-bit heart rate measurement value in BPM
```

```
  //    B4:5    = UINT16 - Energy expended in joules
  //    B6:7    = UINT16 - RR Internal (1/1024 second resolution)
  hrmc.setProperties(CHR_PROPS_NOTIFY);
  hrmc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  hrmc.setFixedLen(2);
  hrmc.setCccdWriteCallback(cccd_callback);  // Optionally capture CCCD updates
  hrmc.begin();
  uint8_t hrmdata[2] = { 0b00000110, 0x40 }; // Set the characteristic to use 8-bit
values, with the sensor connected and detected
  hrmc.notify(hrmdata, 2);                   // Use .notify instead of .write!

  // Configure the Body Sensor Location characteristic
  // See: https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
  // Permission = Read
  // Min Len    = 1
  // Max Len    = 1
  //    B0      = UINT8 - Body Sensor Location
  //      0     = Other
  //      1     = Chest
  //      2     = Wrist
  //      3     = Finger
  //      4     = Hand
  //      5     = Ear Lobe
  //      6     = Foot
  //      7:255 = Reserved
  bslc.setProperties(CHR_PROPS_READ);
  bslc.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  bslc.setFixedLen(1);
  bslc.begin();
  bslc.write8(2);    // Set the characteristic to 'Wrist' (2)
}

void cccd_callback(BLECharacteristic&amp;amp; chr, uint16_t cccd_value)
{
    // Display the raw request packet
    Serial.print("CCCD Updated: ");
    //Serial.printBuffer(request-&amp;gt;data, request-&amp;gt;len);
    Serial.print(cccd_value);
    Serial.println("");

    // Check the characteristic this CCCD update is associated with in case
    // this handler is used for multiple CCCD records.
    if (chr.uuid == hrmc.uuid) {
        if (chr.notifyEnabled()) {
            Serial.println("Heart Rate Measurement 'Notify' enabled");
        } else {
            Serial.println("Heart Rate Measurement 'Notify' disabled");
        }
    }
}
```

BLEClientService

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

This base class is used when defining custom BLE Gatt Clients.

Unless you are implementing a custom GATT client service and characteristic, you normally won't use this base class directly, and would instantiate and call a higher level helper service or characteristic included in the Bluefruit nRF52 API.

Basic Usage

There are normally only threes operations required to use the BLEClientService class:

1.) You need to declare and instantiate the class with an appropriate 16-bit or 128-bit UUID in the constructor:

```
BLEClientService myService = BLEService(0x1234);
```

2.) You then need to call the **.begin()** method on the instance before adding any BLEClientCharacteristics to it (via the BLEClientCharacteristic's respective **.begin()** function call):

```
myService.begin();
```

3) When connected e.g in connect callback, you should call **.discover()** to discover the service

```
myService.discover();
```

API

BLEClientService has the following overall class structure:

This documentation may be slightly out of date as bugs are fixed, and the API develops. You should always consult the Github repo for the definitive latest code release and class definitions!

```
BLEUuid uuid;

// Constructors
BLEClientService(void);
BLEClientService(BLEUuid bleuuid);

virtual bool     begin(void);

virtual bool     discover  (uint16_t conn_handle);
        bool     discovered(void);

        uint16_t connHandle(void);
```

```
        void             setHandleRange(ble_gattc_handle_range_t handle_range);
ble_gattc_handle_range_t getHandleRange(void);
```

Example

The following example declares a HRM (Heart Rate Monitor) service, and assigns some characteristics to it:

```
/*********************************************************************
 This is an example for our nRF52 based Bluefruit LE modules

 Pick one up today in the adafruit shop!

 Adafruit invests time and resources providing this open source code,
 please support Adafruit and open-source hardware by purchasing
 products from Adafruit!

 MIT license, check LICENSE for more information
 All text above, and the splash screen below must be included in
 any redistribution
*********************************************************************/

/* This sketch show how to use BLEClientService and BLEClientCharacteristic
 * to implement a custom client that is used to talk with Gatt server on
 * peripheral.
 *
 * Note: you will need another feather52 running peripheral/custom_HRM sketch
 * to test with.
 */

#include &lt;bluefruit.h&gt;

/* HRM Service Definitions
 * Heart Rate Monitor Service:  0x180D
 * Heart Rate Measurement Char: 0x2A37 (Mandatory)
 * Body Sensor Location Char:   0x2A38 (Optional)
 */

BLEClientService        hrms(UUID16_SVC_HEART_RATE);
BLEClientCharacteristic hrmc(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLEClientCharacteristic bslc(UUID16_CHR_BODY_SENSOR_LOCATION);

void setup()
{
  Serial.begin(115200);

  Serial.println("Bluefruit52 Central Custom HRM Example");
  Serial.println("--------------------------------------\n");

  // Initialize Bluefruit with maximum connections as Peripheral = 0, Central = 1
  // SRAM usage required by SoftDevice will increase dramatically with number of
connections
  Bluefruit.begin(0, 1);

  Bluefruit.setName("Bluefruit52 Central");

  // Initialize HRM client
  hrms.begin();

  // Initialize client characteristics of HRM.
  // Note: Client Char will be added to the last service that is begin()ed.
  bslc.begin();
```

```
  // set up callback for receiving measurement
  hrmc.setNotifyCallback(hrm_notify_callback);
  hrmc.begin();

  // Increase Blink rate to different from PrPh advertising mode
  Bluefruit.setConnLedInterval(250);

  // Callbacks for Central
  Bluefruit.Central.setDisconnectCallback(disconnect_callback);
  Bluefruit.Central.setConnectCallback(connect_callback);

  /* Start Central Scanning
   * - Enable auto scan if disconnected
   * - Interval = 100 ms, window = 80 ms
   * - Don't use active scan
   * - Filter only accept HRM service
   * - Start(timeout) with timeout = 0 will scan forever (until connected)
   */
  Bluefruit.Scanner.setRxCallback(scan_callback);
  Bluefruit.Scanner.restartOnDisconnect(true);
  Bluefruit.Scanner.setInterval(160, 80); // in unit of 0.625 ms
  Bluefruit.Scanner.filterUuid(hrms.uuid);
  Bluefruit.Scanner.useActiveScan(false);
  Bluefruit.Scanner.start(0);                   // // 0 = Don't stop scanning after
n seconds
}

void loop()
{
  // do nothing
}

/**
 * Callback invoked when scanner pick up an advertising data
 * @param report Structural advertising data
 */
void scan_callback(ble_gap_evt_adv_report_t* report)
{
  // Connect to device with HRM service in advertising
  Bluefruit.Central.connect(report);
}

/**
 * Callback invoked when an connection is established
 * @param conn_handle
 */
void connect_callback(uint16_t conn_handle)
{
  Serial.println("Connected");
  Serial.print("Discovering HRM Service ... ");

  // If HRM is not found, disconnect and return
  if ( !hrms.discover(conn_handle) )
  {
    Serial.println("Found NONE");

    // disconect since we couldn't find HRM service
    Bluefruit.Central.disconnect(conn_handle);

    return;
  }

  // Once HRM service is found, we continue to discover its characteristic
  Serial.println("Found it");


  Serial.print("Discovering Measurement characteristic ... ");
  if ( !hrmc.discover() )
```

```
  {
    // Measurement chr is mandatory, if it is not found (valid), then disconnect
    Serial.println("not found !!!");
    Serial.println("Measurement characteristic is mandatory but not found");
    Bluefruit.Central.disconnect(conn_handle);
    return;
  }
  Serial.println("Found it");

  // Measurement is found, continue to look for option Body Sensor Location
  // https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
  // Body Sensor Location is optional, print out the location in text if present
  Serial.print("Discovering Body Sensor Location characteristic ... ");
  if ( bslc.discover() )
  {
    Serial.println("Found it");

    // Body sensor location value is 8 bit
    const char* body_str[] = { "Other", "Chest", "Wrist", "Finger", "Hand", "Ear
Lobe", "Foot" };

    // Read 8-bit BSLC value from peripheral
    uint8_t loc_value = bslc.read8();

    Serial.print("Body Location Sensor: ");
    Serial.println(body_str[loc_value]);
  }else
  {
    Serial.println("Found NONE");
  }

  // Reaching here means we are ready to go, let's enable notification on
measurement chr
  if ( hrmc.enableNotify() )
  {
    Serial.println("Ready to receive HRM Measurement value");
  }else
  {
    Serial.println("Couldn't enable notify for HRM Measurement. Increase DEBUG
LEVEL for troubleshooting");
  }
}

/**
 * Callback invoked when a connection is dropped
 * @param conn_handle
 * @param reason
 */
void disconnect_callback(uint16_t conn_handle, uint8_t reason)
{
  (void) conn_handle;
  (void) reason;

  Serial.println("Disconnected");
}


/**
 * Hooked callback that triggered when a measurement value is sent from peripheral
 * @param chr   Pointer client characteristic that even occurred,
 *              in this example it should be hrmc
 * @param data  Pointer to received data
 * @param len   Length of received data
 */
void hrm_notify_callback(BLEClientCharacteristic* chr, uint8_t* data, uint16_t len)
{
  // https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
```

```
  // Measurement contains of control byte0 and measurement (8 or 16 bit) + optional
field
  // if byte0's bit0 is 0 --&gt; measurement is 8 bit, otherwise 16 bit.

  Serial.print("HRM Measurement: ");

  if ( data[0] &amp; bit(0) )
  {
    uint16_t value;
    memcpy(&amp;value, data+1, 2);

    Serial.println(value);
  }
  else
  {
    Serial.println(data[1]);
  }
}
```

BLEClientCharacteristic

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

This base class is used when defining custom client for BLE GATT characteristics, and is used throughout the Adafruit Bluefruit nRF52 API and helper classes.

Unless you are implementing a custom client for GATT service and characteristic, you normally won't use this base class directly, and would instantiate and call a higher level helper service or characteristic included in the Bluefruit nRF52 API.

Basic Usage

There are three main steps to using the BLECharacteristic class.

1.) First, you need to declare and instantiate your BLECharacteristic class with a 16-bit or 128-bit UUID:

```
BLEClientCharacteristic myChar = BLEClientCharacteristic(0xABCD);
```

2.) Then you need to set the relevant callback for the characteristic if it supports notify or indicate.

```
myChar.setNotifyCallback(notify_callback);
myChar.begin();
```

- **.setNotifyCallback** This sets the callback that will be fired when we receive a Notify message from peripheral. This is needed to handle notifiable characteristic since callback allow us to response to the message in timely manner
- **.begin()** will cause this characteristic to be added to the last **BLEClientService** that had it's **.begin()** method called.

3) Discover the characteristic after connected to peripheral by calling `.discover()` It is a must in order to perform any operation such as .read(), .write(), .enableNotify().

```
if ( myChar.discover() )
{
  uint32_t value = myChar.read32();
}
```

API

BLEClientCharacteristic has the following overall class structure:

This documentation may be slightly out of date as bugs are fixed, and the API develops. You should always consult the Github repo for the definitive latest code release and class definitions!

```
/*--------- Callback Signatures ----------*/
typedef void (*notify_cb_t  ) (BLEClientCharacteristic* chr, uint8_t* data,
uint16_t len);
typedef void (*indicate_cb_t) (BLEClientCharacteristic* chr, uint8_t* data,
uint16_t len);

BLEUuid uuid;

// Constructors
BLEClientCharacteristic(void);
BLEClientCharacteristic(BLEUuid bleuuid);

// Destructor
virtual ~BLEClientCharacteristic();

void     begin(BLEClientService* parent_svc = NULL);

bool     discover(void);
bool     discovered(void);

uint16_t connHandle(void);
uint16_t valueHandle(void);
uint8_t  properties(void);
```

```
BLEClientService&amp; parentService(void);

/*------------- Read -------------*/
uint16_t read(void* buffer, uint16_t bufsize);
uint8_t  read8 (void);
uint16_t read16(void);
uint32_t read32(void);

/*------------- Write without Response-------------*/
uint16_t write     (const void* data, uint16_t len);
uint16_t write8    (uint8_t value);
uint16_t write16   (uint16_t value);
uint16_t write32   (uint32_t value);

/*------------- Write with Response-------------*/
uint16_t write_resp(const void* data, uint16_t len);
uint16_t write8_resp    (uint8_t value);
uint16_t write16_resp   (uint16_t value);
uint16_t write32_resp   (uint32_t value);

/*------------- Notify -------------*/
bool     writeCCCD       (uint16_t value);

bool     enableNotify    (void);
bool     disableNotify   (void);

bool     enableIndicate  (void);
bool     disableIndicate (void);

/*------------- Callbacks -------------*/
void     setNotifyCallback(notify_cb_t fp, bool useAdaCallback = true);
void     setIndicateCallback(indicate_cb_t fp, bool useAdaCallback = true);
```

Example

The following example configures an instance of the Heart Rate Monitor (HRM) Service and it's related characteristics:

```
/*********************************************************************
 This is an example for our nRF52 based Bluefruit LE modules

 Pick one up today in the adafruit shop!

 Adafruit invests time and resources providing this open source code,
 please support Adafruit and open-source hardware by purchasing
 products from Adafruit!

 MIT license, check LICENSE for more information
 All text above, and the splash screen below must be included in
 any redistribution
*********************************************************************/

/* This sketch show how to use BLEClientService and BLEClientCharacteristic
 * to implement a custom client that is used to talk with Gatt server on
 * peripheral.
 *
 * Note: you will need another feather52 running peripheral/custom_HRM sketch
 * to test with.
 */

#include &lt;bluefruit.h&gt;
```

```
/* HRM Service Definitions
 * Heart Rate Monitor Service:  0x180D
 * Heart Rate Measurement Char: 0x2A37 (Mandatory)
 * Body Sensor Location Char:   0x2A38 (Optional)
 */

BLEClientService        hrms(UUID16_SVC_HEART_RATE);
BLEClientCharacteristic hrmc(UUID16_CHR_HEART_RATE_MEASUREMENT);
BLEClientCharacteristic bslc(UUID16_CHR_BODY_SENSOR_LOCATION);

void setup()
{
  Serial.begin(115200);

  Serial.println("Bluefruit52 Central Custom HRM Example");
  Serial.println("--------------------------------------\n");

  // Initialize Bluefruit with maximum connections as Peripheral = 0, Central = 1
  // SRAM usage required by SoftDevice will increase dramatically with number of
connections
  Bluefruit.begin(0, 1);

  Bluefruit.setName("Bluefruit52 Central");

  // Initialize HRM client
  hrms.begin();

  // Initialize client characteristics of HRM.
  // Note: Client Char will be added to the last service that is begin()ed.
  bslc.begin();

  // set up callback for receiving measurement
  hrmc.setNotifyCallback(hrm_notify_callback);
  hrmc.begin();

  // Increase Blink rate to different from PrPh advertising mode
  Bluefruit.setConnLedInterval(250);

  // Callbacks for Central
  Bluefruit.Central.setDisconnectCallback(disconnect_callback);
  Bluefruit.Central.setConnectCallback(connect_callback);

  /* Start Central Scanning
   * - Enable auto scan if disconnected
   * - Interval = 100 ms, window = 80 ms
   * - Don't use active scan
   * - Filter only accept HRM service
   * - Start(timeout) with timeout = 0 will scan forever (until connected)
   */
  Bluefruit.Scanner.setRxCallback(scan_callback);
  Bluefruit.Scanner.restartOnDisconnect(true);
  Bluefruit.Scanner.setInterval(160, 80); // in unit of 0.625 ms
  Bluefruit.Scanner.filterUuid(hrms.uuid);
  Bluefruit.Scanner.useActiveScan(false);
  Bluefruit.Scanner.start(0);                   // // 0 = Don't stop scanning after
n seconds
}

void loop()
{
  // do nothing
}

/**
 * Callback invoked when scanner pick up an advertising data
 * @param report Structural advertising data
 */
void scan_callback(ble_gap_evt_adv_report_t* report)
{
```

```
  // Connect to device with HRM service in advertising
  Bluefruit.Central.connect(report);
}

/**
 * Callback invoked when an connection is established
 * @param conn_handle
 */
void connect_callback(uint16_t conn_handle)
{
  Serial.println("Connected");
  Serial.print("Discovering HRM Service ... ");

  // If HRM is not found, disconnect and return
  if ( !hrms.discover(conn_handle) )
  {
    Serial.println("Found NONE");

    // disconect since we couldn't find HRM service
    Bluefruit.Central.disconnect(conn_handle);

    return;
  }

  // Once HRM service is found, we continue to discover its characteristic
  Serial.println("Found it");


  Serial.print("Discovering Measurement characteristic ... ");
  if ( !hrmc.discover() )
  {
    // Measurement chr is mandatory, if it is not found (valid), then disconnect
    Serial.println("not found !!!");
    Serial.println("Measurement characteristic is mandatory but not found");
    Bluefruit.Central.disconnect(conn_handle);
    return;
  }
  Serial.println("Found it");

  // Measurement is found, continue to look for option Body Sensor Location
  // https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.body_sensor_location.xml
  // Body Sensor Location is optional, print out the location in text if present
  Serial.print("Discovering Body Sensor Location characteristic ... ");
  if ( bslc.discover() )
  {
    Serial.println("Found it");

    // Body sensor location value is 8 bit
    const char* body_str[] = { "Other", "Chest", "Wrist", "Finger", "Hand", "Ear
Lobe", "Foot" };

    // Read 8-bit BSLC value from peripheral
    uint8_t loc_value = bslc.read8();

    Serial.print("Body Location Sensor: ");
    Serial.println(body_str[loc_value]);
  }else
  {
    Serial.println("Found NONE");
  }

  // Reaching here means we are ready to go, let's enable notification on
measurement chr
  if ( hrmc.enableNotify() )
  {
    Serial.println("Ready to receive HRM Measurement value");
  }else
  {
```

```
    Serial.println("Couldn't enable notify for HRM Measurement. Increase DEBUG
LEVEL for troubleshooting");
  }
}

/**
 * Callback invoked when a connection is dropped
 * @param conn_handle
 * @param reason
 */
void disconnect_callback(uint16_t conn_handle, uint8_t reason)
{
  (void) conn_handle;
  (void) reason;

  Serial.println("Disconnected");
}


/**
 * Hooked callback that triggered when a measurement value is sent from peripheral
 * @param chr   Pointer client characteristic that even occurred,
 *              in this example it should be hrmc
 * @param data  Pointer to received data
 * @param len   Length of received data
 */
void hrm_notify_callback(BLEClientCharacteristic* chr, uint8_t* data, uint16_t len)
{
  // https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.heart_rate_measurement.xml
  // Measurement contains of control byte0 and measurement (8 or 16 bit) + optional
field
  // if byte0's bit0 is 0 --&gt; measurement is 8 bit, otherwise 16 bit.

  Serial.print("HRM Measurement: ");

  if ( data[0] &amp; bit(0) )
  {
    uint16_t value;
    memcpy(&amp;value, data+1, 2);

    Serial.println(value);
  }
  else
  {
    Serial.println(data[1]);
  }
}
```

BLEDiscovery

This page is a work in progress as the API is changing as we migrate to S132v5 (nRF52832) and S140 (nRF52840) and add better Central mode support.

BLEDiscovery is a helper class to make finding characteristics on a Gatt server (hosted on a BLE peripheral) easier. For service discovery, the BLEClientService's `discover()` API must be used, as shown below:

API

```
BLEDiscovery(void); // Constructor

void     begin(void);
bool     begun(void);

void                      setHandleRange(ble_gattc_handle_range_t handle_range);
ble_gattc_handle_range_t getHandleRange(void);

uint8_t  discoverCharacteristic(uint16_t conn_handle, BLEClientCharacteristic*
chr[], uint8_t count);
uint8_t  discoverCharacteristic(uint16_t conn_handle, BLEClientCharacteristic&amp;
chr1);
uint8_t  discoverCharacteristic(uint16_t conn_handle, BLEClientCharacteristic&amp;
chr1, BLEClientCharacteristic&amp; chr2);
uint8_t  discoverCharacteristic(uint16_t conn_handle, BLEClientCharacteristic&amp;
chr1, BLEClientCharacteristic&amp; chr2, BLEClientCharacteristic&amp; chr3);
uint8_t  discoverCharacteristic(uint16_t conn_handle, BLEClientCharacteristic&amp;
chr1, BLEClientCharacteristic&amp; chr2, BLEClientCharacteristic&amp; chr3,
BLEClientCharacteristic&amp; chr4);
uint8_t  discoverCharacteristic(uint16_t conn_handle, BLEClientCharacteristic&amp;
chr1, BLEClientCharacteristic&amp; chr2, BLEClientCharacteristic&amp; chr3,
BLEClientCharacteristic&amp; chr4, BLEClientCharacteristic&amp; chr5);
```

For concrete examples of how to use this API see the 'Central' folder in the examples that are part of the BSP.

BLEDis

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

This helper class acts as a wrapper for the Bluetooth Device Information Service (https://adafru.it/q9E) (0x180A). This official GATT service allows you to publish basic information about your device in a generic manner.

The Bluefruit BLEDis helper class exposes the following characteristics:

- Model Number String (https://adafru.it/vav) (0x2A24), exposed via **.setModel(const char*)**
- Serial Number String (https://adafru.it/vaw) (0x2A25), private
- Firmware Revision String (https://adafru.it/vax) (0x2A26), private
- Hardware Revision String (https://adafru.it/vay) (0x2A27), exposed via **.setHardwareRev(const char*)**

- Software Revision String (https://adafru.it/vaz) (0x2A28), exposed via **.setSoftwareRev(const char*)**
- Manufacturer Name String (https://adafru.it/vaA) (0x2A29), exposed via **.setManufacturer(const char*)**

The **Serial Number String** is private and is populated with a unique device ID that nRF52832 SoCs are programmed with during manufacturing.

The **Firmware Revision String** is also private and is populated with the following fields (to help us track issues and offer better feedback in the support forums):

  - Softdevice Name (Sxxx)
  - Softdevice Version (x.x.x)
  - Bootloader Version (x.x.x)

Note: The Softdevice and Bootloader fields are separated by a single comma, meaning the final output will resemble the following string: 'S132 2.0.1, 0.5.0'

The remaining characteristics are all public and can be set to an value (up to 20 chars in length) using the appropriate helper function, but they have the following default values (for the nRF52832):

- **Model Number String**: Bluefruit Feather 52
- **Hardware Revision String**: NULL
- **Software Revision String**: The nRF52 BSP version number
- **Manufacturer Name String**: Adafruit Industries

Setting a public value to NULL will prevent the characteristic from being present in the DIS service.

API

The following functions and constructors are defined in the BLEDis class:

```
BLEDis(void);

void setModel(const char* model);
void setHardwareRev(const char* hw_rev);
void setSoftwareRev(const char* sw_rev);
void setManufacturer(const char* manufacturer);
```

```
err_t begin(void);
```

The individual characteristic values are set via the **.set*()** functions above, and when all values have been set you call the **.begin()** function to add the service to the device's internal GATT registry.

Example

The following bare bones examples show how to setup the device information service with user-configurable strings for values:

```
#include &lt;bluefruit.h&gt;

BLEDis bledis;

void setup()
{
  Serial.begin(115200);

  Serial.println("Bluefruit52 DIS Example");

  Bluefruit.begin();
  Bluefruit.setName("Bluefruit52");

  // Configure and Start Device Information Service
  bledis.setManufacturer("Adafruit Industries");
  bledis.setModel("Bluefruit Feather52");
  bledis.begin();

  // Set up Advertising Packet
  setupAdv();

  // Start Advertising
  Bluefruit.Advertising.start();
}

void setupAdv(void)
{
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
  Bluefruit.Advertising.addTxPower();

  // There isn't enough room in the advertising packet for the
  // name so we'll place it on the secondary Scan Response packet
  Bluefruit.ScanResponse.addName();
}

void loop()
{
}
```

Output

If you examine the device using the Bluefruit LE Connect app on iOS, Android or OS X you should see something resembling the following output:

UUID	Value
▼ Device Information	
Model Number	Bluefruit Feather52
Serial Number	8B9CE51B850F75A7
Firmware Revision	0.5.0,S132,2.0.1
Software Revision	0.4.5
Manufacturer Name	Adafruit Industries

BLEUart

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

BLEUart is a wrapper class for NUS (Nordic UART Service), which is a proprietary service defined by Nordic Semiconductors that we use as a baseline transport mechanism between Bluefruit modules and our mobile and desktop Bluefruit LE Connect applications. You can use it to easily send ASCII or binary data in both directions, between the peripheral and the central device.

API

BLEUart has the following public API:

```
// RX Callback signature (fires when data was written by the central)
typedef void (*rx_callback_t) (void);

// Constructor
BLEUart(uint16_t fifo_depth = BLE_UART_DEFAULT_FIFO_DEPTH);

virtual err_t begin(void);

bool notifyEnabled(void);

void setRxCallback( rx_callback_t fp);

// Stream API
virtual int       read       ( void );
virtual int       read       ( uint8_t * buf, size_t size );
virtual size_t    write      ( uint8_t b );
virtual size_t    write      ( const uint8_t *content, size_t len );
virtual int       available  ( void );
virtual int       peek       ( void );
virtual void      flush      ( void );

// Pull in write(str) and write(buf, size) from Print
using Print::write;
```

Example

The following example shows how to use the BLEUart helper class.

This example may be out of date, and you should always consult the latest example code in the nRF52 BSP!

```
#include &lt;bluefruit.h&gt;

BLEDis  bledis;
BLEUart bleuart;
BLEBas  blebas;

#define STATUS_LED  (17)
#define BLINKY_MS   (2000)

uint32_t blinkyms;

void setup()
{
  Serial.begin(115200);

  Serial.println("Bluefruit52 BLEUART Example");

  // Setup LED pins and reset blinky counter
  pinMode(STATUS_LED, OUTPUT);
  blinkyms = millis();

  // Setup the BLE LED to be enabled on CONNECT
  // Note: This is actually the default behaviour, but provided
  // here in case you want to control this manually via PIN 19
  Bluefruit.autoConnLed(true);

  Bluefruit.begin();
  Bluefruit.setName("Bluefruit52");
  Bluefruit.setConnectCallback(connect_callback);
  Bluefruit.setDisconnectCallback(disconnect_callback);

  // Configure and Start Device Information Service
  bledis.setManufacturer("Adafruit Industries");
  bledis.setModel("Bluefruit Feather52");
  bledis.begin();

  // Configure and Start BLE Uart Service
  bleuart.begin();

  // Start BLE Battery Service
  blebas.begin();
  blebas.update(100);

  // Set up Advertising Packet
  setupAdv();

  // Start Advertising
  Bluefruit.Advertising.start();
}

void setupAdv(void)
{
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
  Bluefruit.Advertising.addTxPower();

  // Include bleuart 128-bit uuid
```

```
  Bluefruit.Advertising.addService(bleuart);

  // There is no room for Name in Advertising packet
  // Use Scan response for Name
  Bluefruit.ScanResponse.addName();
}

void loop()
{
  // Blinky!
  if (blinkyms+BLINKY_MS &lt; millis()) {
    blinkyms = millis();
    digitalToggle(STATUS_LED);
  }

  // Forward from Serial to BLEUART
  if (Serial.available())
  {
    // Delay to get enough input data since we have a
    // limited amount of space in the transmit buffer
    delay(2);

    uint8_t buf[64];
    int count = Serial.readBytes(buf, sizeof(buf));
    bleuart.write( buf, count );
  }

  // Forward from BLEUART to Serial
  if ( bleuart.available() )
  {
    uint8_t ch;
    ch = (uint8_t) bleuart.read();
    Serial.write(ch);
  }
}

void connect_callback(void)
{
  Serial.println("Connected");
}

void disconnect_callback(uint8_t reason)
{
  (void) reason;

  Serial.println();
  Serial.println("Disconnected");
  Serial.println("Bluefruit will start advertising again");
}
```

BLEClientUart

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

BLEClientUart is a wrapper class for the client side of the NUS or 'Nordic UART Service' (aka 'BLE UART'). It is only required when your Bluefruit nRF52 board is

acting as Central communicating to other BLE peripherals that expose the BLEUart (https://adafru.it/yud) service.

API

BLEClientUart has the following public API:

```
// Callback Signatures
typedef void (*rx_callback_t) (BLEClientUart&amp;amp; svc);

BLEClientUart(uint16_t fifo_depth = BLE_UART_DEFAULT_FIFO_DEPTH);

virtual bool  begin(void);
virtual bool  discover(uint16_t conn_handle);

void setRxCallback( rx_callback_t fp);

bool enableTXD(void);
bool disableTXD(void);

// Stream API
virtual int       read       ( void );
virtual int       read       ( uint8_t * buf, size_t size );
        int       read       ( char    * buf, size_t size ) { return
read( (uint8_t*) buf, size); }
virtual size_t    write      ( uint8_t b );
virtual size_t    write      ( const uint8_t *content, size_t len );
virtual int       available  ( void );
virtual int       peek       ( void );
virtual void      flush      ( void );
```

Examples

The following example shows how to use the BLEClientUart helper class.

```
#include &lt;bluefruit.h&gt;

BLEClientDis  clientDis;
BLEClientUart clientUart;

void setup()
{
  Serial.begin(115200);

  Serial.println("Bluefruit52 Central BLEUART Example");
  Serial.println("-----------------------------------\n");

  // Initialize Bluefruit with maximum connections as Peripheral = 0, Central = 1
  // SRAM usage required by SoftDevice will increase dramatically with number of
connections
  Bluefruit.begin(0, 1);

  Bluefruit.setName("Bluefruit52 Central");

  // Configure DIS client
  clientDis.begin();

  // Init BLE Central Uart Serivce
```

```
  clientUart.begin();
  clientUart.setRxCallback(bleuart_rx_callback);

  // Increase Blink rate to different from PrPh advertising mode
  Bluefruit.setConnLedInterval(250);

  // Callbacks for Central
  Bluefruit.Central.setConnectCallback(connect_callback);
  Bluefruit.Central.setDisconnectCallback(disconnect_callback);

  /* Start Central Scanning
   * - Enable auto scan if disconnected
   * - Interval = 100 ms, window = 80 ms
   * - Don't use active scan
   * - Start(timeout) with timeout = 0 will scan forever (until connected)
   */
  Bluefruit.Scanner.setRxCallback(scan_callback);
  Bluefruit.Scanner.restartOnDisconnect(true);
  Bluefruit.Scanner.setInterval(160, 80); // in unit of 0.625 ms
  Bluefruit.Scanner.useActiveScan(false);
  Bluefruit.Scanner.start(0);                   // // 0 = Don't stop scanning after
n seconds
}

/**
 * Callback invoked when scanner pick up an advertising data
 * @param report Structural advertising data
 */
void scan_callback(ble_gap_evt_adv_report_t* report)
{
  // Check if advertising contain BleUart service
  if ( Bluefruit.Scanner.checkReportForService(report, clientUart) )
  {
    Serial.print("BLE UART service detected. Connecting ... ");

    // Connect to device with bleuart service in advertising
    Bluefruit.Central.connect(report);
  }
}

/**
 * Callback invoked when an connection is established
 * @param conn_handle
 */
void connect_callback(uint16_t conn_handle)
{
  Serial.println("Connected");

  Serial.print("Dicovering DIS ... ");
  if ( clientDis.discover(conn_handle) )
  {
    Serial.println("Found it");
    char buffer[32+1];

    // read and print out Manufacturer
    memset(buffer, 0, sizeof(buffer));
    if ( clientDis.getManufacturer(buffer, sizeof(buffer)) )
    {
      Serial.print("Manufacturer: ");
      Serial.println(buffer);
    }

    // read and print out Model Number
    memset(buffer, 0, sizeof(buffer));
    if ( clientDis.getModel(buffer, sizeof(buffer)) )
    {
      Serial.print("Model: ");
      Serial.println(buffer);
    }
```

```
    Serial.println();
  }

  Serial.print("Discovering BLE Uart Service ... ");

  if ( clientUart.discover(conn_handle) )
  {
    Serial.println("Found it");

    Serial.println("Enable TXD's notify");
    clientUart.enableTXD();

    Serial.println("Ready to receive from peripheral");
  }else
  {
    Serial.println("Found NONE");

    // disconect since we couldn't find bleuart service
    Bluefruit.Central.disconnect(conn_handle);
  }
}

/**
 * Callback invoked when a connection is dropped
 * @param conn_handle
 * @param reason
 */
void disconnect_callback(uint16_t conn_handle, uint8_t reason)
{
  (void) conn_handle;
  (void) reason;

  Serial.println("Disconnected");
}

/**
 * Callback invoked when uart received data
 * @param uart_svc Reference object to the service where the data
 * arrived. In this example it is clientUart
 */
void bleuart_rx_callback(BLEClientUart&amp;amp; uart_svc)
{
  Serial.print("[RX]: ");

  while ( uart_svc.available() )
  {
    Serial.print( (char) uart_svc.read() );
  }

  Serial.println();
}

void loop()
{
  if ( Bluefruit.Central.connected() )
  {
    // Not discovered yet
    if ( clientUart.discovered() )
    {
      // Discovered means in working state
      // Get Serial input and send to Peripheral
      if ( Serial.available() )
      {
        delay(2); // delay a bit for all characters to arrive

        char str[20+1] = { 0 };
        Serial.readBytes(str, 20);
```

```
        clientUart.print( str );
      }
    }
  }
}
```

BLEBeacon

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

The BLEBeacon helper class allows you to easily configure the nRF52 as a 'Beacon', which uses the advertising packet to send out a specifically format chunk of data to any devices in listening range.

The following values must be set in order to generate a valid 'Beacon' packet:

- **Manufacturer ID**: A 16-bit value (registered with the Bluetooth SIG (https://adafru.it/vaB)!) that identifies the manufacturer.
- **Major**: A 16-bit 'Major' number, used to differentiate beacon nodes.
- **Minor**: A 16-bit 'Minor' number, used to differentiate beacon nodes.
- **RSSI @ 1M**: A signed 8-bit value (int8_t) indicating the RSSI measurement at 1m distance from the node, used to estimate distance to the beacon itself.

These values can either be set in the constructor, or via the individual functions exposed as part of this helper class.

API

BLEBeacon has the following public API:

```
// Constructors
BLEBeacon(void);
BLEBeacon(uint8_t const uuid128[16]);
BLEBeacon(uint8_t const uuid128[16], uint16_t major, uint16_t minor, int8_t rssi);

// Set the beacon payload values
void setManufacturer(uint16_t manfacturer);
void setUuid(uint8_t const uuid128[16]);
void setMajorMinor(uint16_t major, uint16_t minor);
void setRssiAt1m(int8_t rssi);

// Start advertising
```

```
bool start(void);
bool start(BLEAdvertising&amp;amp; adv);
```

In addition to these functions, the BLEAdvertising class (accessible via `Bluefruit.Advertising.*`) exposes the following function to assign Beacon payload to the advertising payload:

```
bool setBeacon(BLEBeacon&amp;amp; beacon);
```

See the example below for a concrete usage example.

Example

The following example will configure the nRF52 to advertise a 'Beacon' payload:

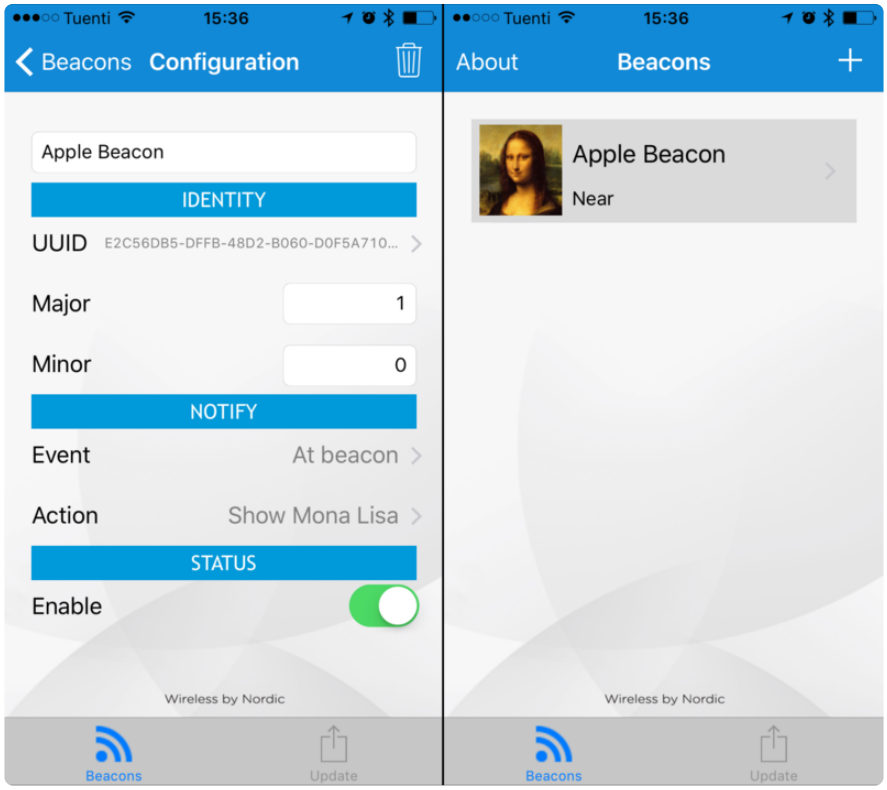
```
#include &amp;lt;bluefruit.h&amp;gt;

// Beacon uses the Manufacturer Specific Data field in the advertising
// packet, which means you must provide a valid Manufacturer ID. Update
// the field below to an appropriate value. For a list of valid IDs see:
// https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers
// 0x004C is Apple (for example)
#define MANUFACTURER_ID   0x004C

// AirLocate UUID: E2C56DB5-DFFB-48D2-B060-D0F5A71096E0
uint8_t beaconUuid[16] =
{
  0xE2, 0xC5, 0x6D, 0xB5, 0xDF, 0xFB, 0x48, 0xD2,
  0xB0, 0x60, 0xD0, 0xF5, 0xA7, 0x10, 0x96, 0xE0,
};

// A valid Beacon packet consists of the following information:
// UUID, Major, Minor, RSSI @ 1M
BLEBeacon beacon(beaconUuid, 0x0001, 0x0000, -54);

void setup()
{
  Serial.begin(115200);

  Serial.println("Bluefruit52 Beacon Example");

  Bluefruit.begin();
  Bluefruit.setName("Bluefruit52");

  // Manufacturer ID is required for Manufacturer Specific Data
  beacon.setManufacturer(MANUFACTURER_ID);

  // Setup the advertising packet
  setupAdv();

  // Start advertising
  Bluefruit.Advertising.start();
}

void setupAdv(void)
{
  // Set the beacon payload using the BLEBeacon class populated
  // earlier in this example
  Bluefruit.Advertising.setBeacon(beacon);
```

```
  // char* adv = Bluefruit.Advertising.getData();

  // There is no room left for 'Name' in the advertising packet
  // Use the optinal secondary Scan Response packet for 'Name' instead
  Bluefruit.ScanResponse.addName();
}

void loop()
{
  // Toggle both LEDs every second
  digitalToggle(LED_BUILTIN);
  delay(1000);
}
```

Testing

If you test with the nRF Beacons application (iOS (https://adafru.it/vaC) or Android (https://adafru.it/vaD)) you can configure the app to look for the UUID, Manufacturer ID, Major and Minor values you provided, and you should be able to see the beacon, as shown in the two screenshots below:

Make sure that the UUID, Major and Minor values match or the application won't detect your beacon node!



BLEMidi

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

BLEMidi is a helper class that adds support for sending and receiving MIDI Messages using the MIDI over Bluetooth LE specification. BLEMidi supports the full standard MIDI protocol (including SysEx messages), and it also can act as the hardware interface for the Arduino MIDI Library.

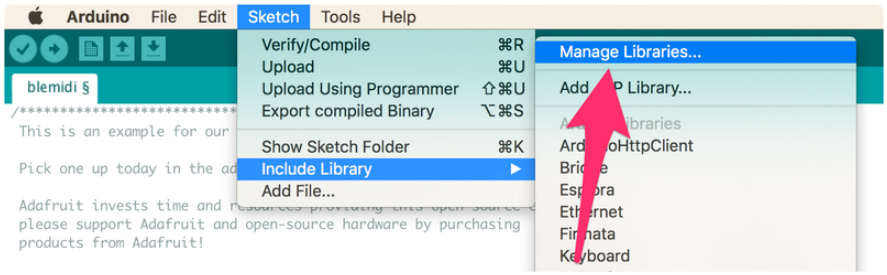
API

BLEMidi has the following public API.

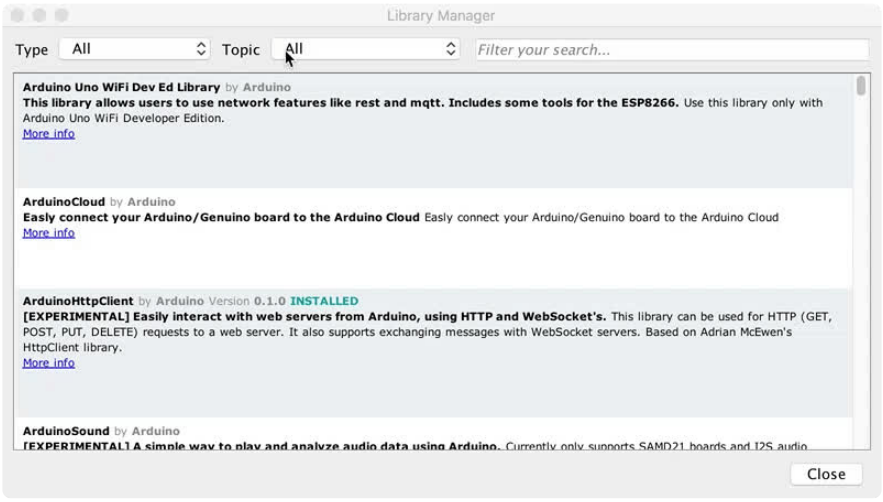
```
// Constructor
BLEMidi(uint16_t fifo_depth = 128);

err_t     begin         (void);
bool      notifyEnabled (void);

// Stream API for Arduino MIDI Library Interface
int       read       (void);
size_t    write      (uint8_t b);
int       available  (void);
int       peek       (void);
void      flush      (void);

size_t    write      (const char *str);
size_t    write      (const uint8_t *buffer, size_t size);
```

Installing the Arduino MIDI Library

BLEMidi is easiest to use when combined with the Arduino MIDI Library by Francois Best, lathoub. You will need version 4.3.1 installed before continuing with the example code.

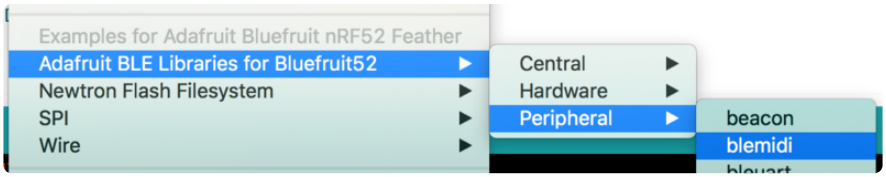


Next, select **Communication** from the topic dropdown, and enter **MIDI Library** into the search box. Click the **Install** button to install version 4.3.0 or higher of the **MIDI Library**.



Example

The **blemidi** example demonstrates how to use the BLEMidi helper class with the **Arduino MIDI Library**. The example sends a looping arpeggio, and prints any incoming MIDI note on and note off messages to the Arduino Serial Monitor.



This example may be out of date, and you should always consult the latest example code in the Bluefruit52 example folder!

```
/*********************************************************************
 This is an example for our nRF52 based Bluefruit LE modules

 Pick one up today in the adafruit shop!

 Adafruit invests time and resources providing this open source code,
 please support Adafruit and open-source hardware by purchasing
 products from Adafruit!

 MIT license, check LICENSE for more information
 All text above, and the splash screen below must be included in
 any redistribution
*********************************************************************/
#include &lt;bluefruit.h&gt;
#include &lt;MIDI.h&gt;

BLEDis bledis;
BLEMidi blemidi;

// Create a new instance of the Arduino MIDI Library,
```

```
// and attach BluefruitLE MIDI as the transport.
MIDI_CREATE_BLE_INSTANCE(blemidi);

// Variable that holds the current position in the sequence.
int position = 0;

// Store example melody as an array of note values
byte note_sequence[] = {
  74,78,81,86,90,93,98,102,57,61,66,69,73,78,81,85,88,92,97,100,97,92,88,85,81,78,
  74,69,66,62,57,62,66,69,74,78,81,86,90,93,97,102,97,93,90,85,81,78,73,68,64,61,
  56,61,64,68,74,78,81,86,90,93,98,102
};

void setup()
{
  Serial.begin(115200);
  Serial.println("Adafruit Bluefruit52 MIDI over Bluetooth LE Example");

  Bluefruit.begin();
  Bluefruit.setName("Bluefruit52 MIDI");

  // Setup the on board blue LED to be enabled on CONNECT
  Bluefruit.autoConnLed(true);

  // Configure and Start Device Information Service
  bledis.setManufacturer("Adafruit Industries");
  bledis.setModel("Bluefruit Feather52");
  bledis.begin();

  // Initialize MIDI, and listen to all MIDI channels
  // This will also call blemidi service's begin()
  MIDI.begin(MIDI_CHANNEL_OMNI);

  // Attach the handleNoteOn function to the MIDI Library. It will
  // be called whenever the Bluefruit receives MIDI Note On messages.
  MIDI.setHandleNoteOn(handleNoteOn);

  // Do the same for MIDI Note Off messages.
  MIDI.setHandleNoteOff(handleNoteOff);

  // Set General Discoverable Mode flag
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);

  // Advertise TX Power
  Bluefruit.Advertising.addTxPower();

  // Advertise BLE MIDI Service
  Bluefruit.Advertising.addService(blemidi);

  // Advertise device name in the Scan Response
  Bluefruit.ScanResponse.addName();

  // Start Advertising
  Bluefruit.Advertising.start();

  // Start MIDI read loop
  Scheduler.startLoop(midiRead);
}

void handleNoteOn(byte channel, byte pitch, byte velocity)
{
  // Log when a note is pressed.
  Serial.printf("Note on: channel = %d, pitch = %d, velocity - %d", channel, pitch,
velocity);
  Serial.println();
}

void handleNoteOff(byte channel, byte pitch, byte velocity)
{
```

```
  // Log when a note is released.
  Serial.printf("Note off: channel = %d, pitch = %d, velocity - %d", channel,
pitch, velocity);
  Serial.println();
}

void loop()
{
  // Don't continue if we aren't connected.
  if (! Bluefruit.connected()) {
    return;
  }

  // Don't continue if the connected device isn't ready to receive messages.
  if (! blemidi.notifyEnabled()) {
    return;
  }

  // Setup variables for the current and previous
  // positions in the note sequence.
  int current = position;
  int previous  = position - 1;

  // If we currently are at position 0, set the
  // previous position to the last note in the sequence.
  if (previous &lt; 0) {
    previous = sizeof(note_sequence) - 1;
  }

  // Send Note On for current position at full velocity (127) on channel 1.
  MIDI.sendNoteOn(note_sequence[current], 127, 1);

  // Send Note Off for previous note.
  MIDI.sendNoteOff(note_sequence[previous], 0, 1);

  // Increment position
  position++;

  // If we are at the end of the sequence, start over.
  if (position &gt;= sizeof(note_sequence)) {
    position = 0;
  }

  delay(286);

}

void midiRead()
{
  // Don't continue if we aren't connected.
  if (! Bluefruit.connected()) {
    return;
  }

  // Don't continue if the connected device isn't ready to receive messages.
  if (! blemidi.notifyEnabled()) {
    return;
  }

  // read any new MIDI messages
  MIDI.read();
}
```

Usage

You will need to do a small bit of setup on your selected platform to connect to the BLE MIDI enabled Bluefruit52.

Click on a platform below to view BLE MIDI setup instructions for your device:

- macOS (OS X)
- iOS
- Android
- Windows

The arpeggio should automatically play once the Bluefruit52 is connected to your software synth. The video below shows the Bluefruit52 connected to Moog's Animoog on iOS.

Note: The board used in the video was a pre-release prototype. The production boards are standard Adafruit Black.

BLEHidAdafruit

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

BLEHidAdafruit allows you to simulate a mouse or keyboard using the HID (Human Interface Device) profile that is part of the Bluetooth Low Energy standard.

Most modern mobile devices with Bluetooth Low Energy support, and the latest operating systems generally support Bluetooth Low Energy mice and keyboards out of the box, once you pair your Bluefruit nRF52/nRF52840 Feather and run an appropriate sketch.

API

The BLEHidAdafruit helper class has the following public API:

```
// Constructor
BLEHidAdafruit(void);

// Call this once to start the HID service
virtual err_t begin(void);

// Keyboard
err_t keyboardReport(hid_keyboard_report_t* report);
err_t keyboardReport(uint8_t modifier, uint8_t keycode[6]);
err_t keyboardReport(uint8_t modifier, uint8_t keycode0, uint8_t keycode1=0,
uint8_t keycode2=0, uint8_t keycode3=0, uint8_t keycode4=0, uint8_t keycode5=0);

err_t keyPress(char ch);
err_t keyRelease(void);
err_t keySequence(const char* str, int interal=5);

// Consumer Media Keys
err_t consumerReport(uint16_t usage_code);
err_t consumerKeyPress(uint16_t usage_code);
err_t consumerKeyRelease(void);

// Mouse
err_t mouseReport(hid_mouse_report_t* report);
err_t mouseReport(uint8_t buttons, int8_t x, int8_t y, int8_t wheel=0, int8_t
pan=0);

err_t mouseButtonPress(uint8_t buttons);
err_t mouseButtonRelease(void);

err_t mouseMove(int8_t x, int8_t y);
err_t mouseScroll(int8_t scroll);
err_t mousePan(int8_t pan);
```

Example Sketches

There are a variety of example sketches showing how to use the BLEHidAdafruit class. You can browse the latest source code on Github with the following links:

- hid_keyboard (https://adafru.it/vb8): This example will simulate an HID keyboard, waiting for data to arrive via the nRF52's serial port (via USB serial), and send that data over the air to the bonded Central device.
- hid_mouse (https://adafru.it/vb9): This example will simulate an HID mouse. To use it run the sketch and open the Serial Monitor, then enter the appropriate characters to move the mouse or trigger/release the mouse buttons.

Bonding HID Devices

In order to use your HID mouse or keyboard, you will first need to **bond** the two devices. The bonding process involves the following steps:

- The two devices will connect to each other normally

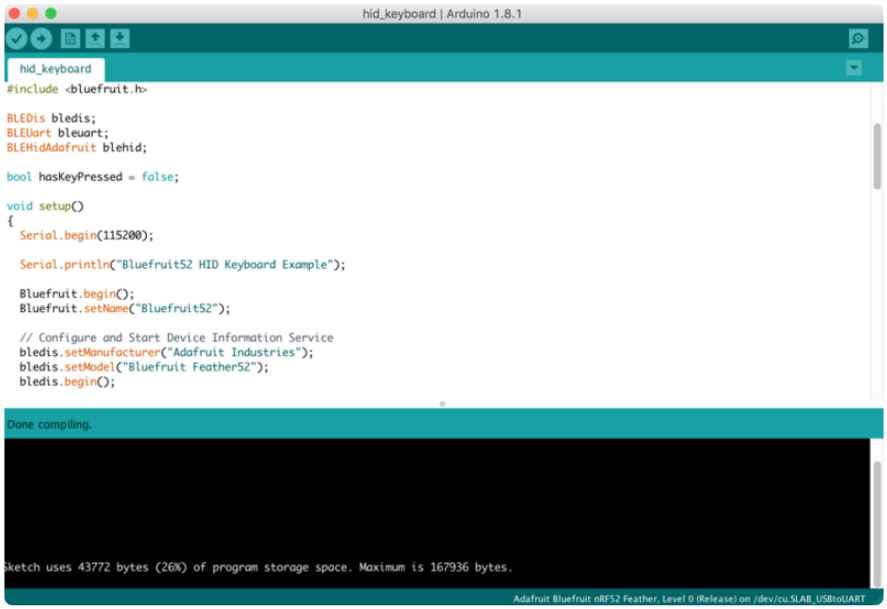
- A set of security keys are exchanged between the two devices, and stores in non-volatile memory on each side. This is to ensure that each side is reasonably confident it is talking to the device it thinks it is for future connections, and to encrypt over the air communication between the devices (so that people can 'sniff' your keyboard data, etc.).
- On the nRF52 side this key data will be stored in a section of flash memory reserved for this purpose using an internal file system.
- The process of storing these security keys is referred to as **bonding**, and allows bonded devices to securely communicate without user interaction in the future.
- To cancel the bonding agreement, you can simply delete the keys on the nRF52 via the clearbonds (https://adafru.it/vba) sketch, or delete the bonding data on your mobile device of computer.

If you run into any bonding problems, try running the clearbonds sketch to remove and old bonding data from local non-volatile memory!

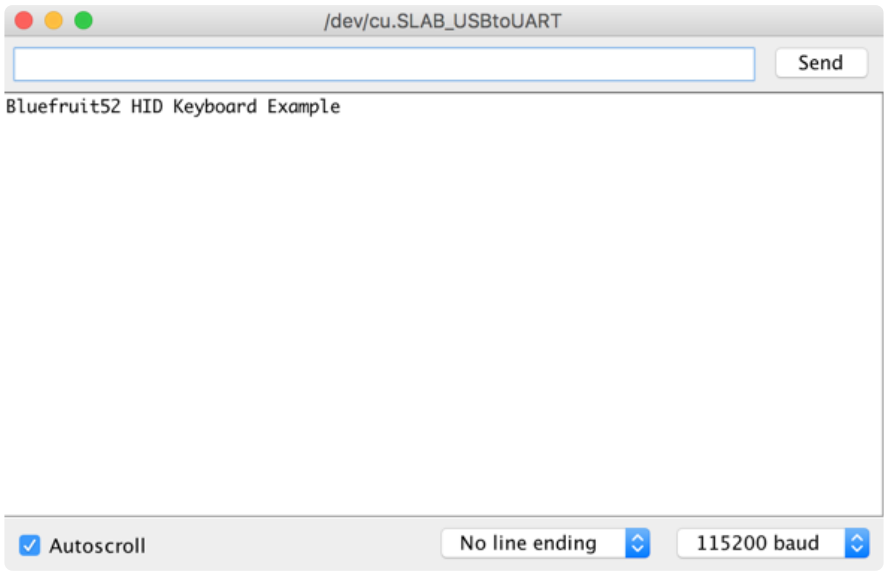
Setting up your Bluefruit device for bonding

To bond an device, run an appropriate HID sketch on the nRF52 to emulate either an HID mouse or an HID keyboard. In the event that you use the HID mouse example you may need to open the Serial Monitor to use it.

In this example we'll run the **hid_keyboard** example sketch, flashing it to the nRF52, which should give you the following results:



Opening the Serial Monitor will give you the following output (though it may differ depending on the debug level you have selected):

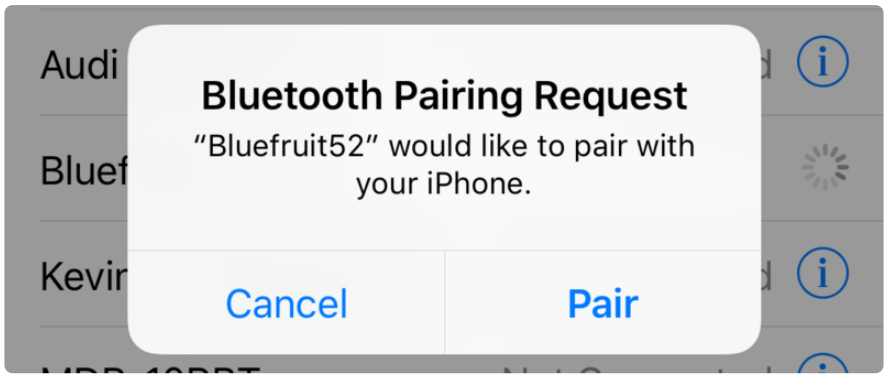


Bonding on iOS

To bond to an iOS device, make sure the sketch is running (as described above) and go into your **Settings** app and Select **Bluetooth**.

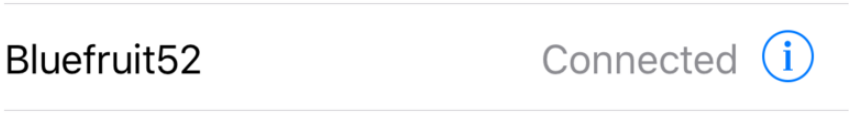
You should see a device at the bottom of this page called **Bluefruit52** (this may vary depending on the version of the sketch you are using!):

Click the device, and you will get a pairing request like this:



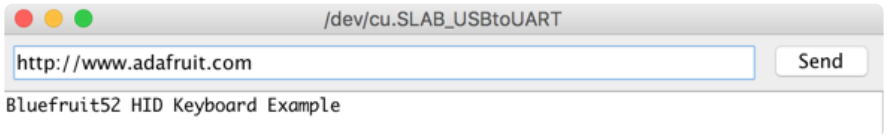
Click the **Pair** button, and the devices will be paired and bonded, and will automatically connect to each other in the future.

If everything went will, you will see the device in your **MY DEVICES** list, as follows:



Testing the HID Keyboard and Bonding

To test the HID keyboard sketch and bonding process, open the **Serial Monitor** (or your favorite terminal emulator), enter some text, and if you are using the Serial Monitor click the **Send** button. This will send some text over the air to whatever textbox or text control has focus in your app.



The text will then appear in your mobile app or bonded device.

If the characters don't match exactly what you send, be sure to check your **keyboard language** settings, since you may be sending data to a device with a different keyboard setup!

BLEAncs

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

BLEAncs is a helper class that enables you to receive notifications from the Apple Notification Center Service (https://adafru.it/ErH) from devices such as an iPhone or iPad. It can be used to receive alerts such as incoming or missed calls, email messages, or most alerts that appear on the mobile device's screen when locked.

API

Because the BLEAncs class is a work in progress, the latest public API for the BLEAncs helper class should be viewed here (https://adafru.it/xen).

ANCS OLED Example

The ancs_oled **(https://adafru.it/xeo)** example uses the Adafruit FeatherWing OLED (http://adafru.it/2900) to display any incoming alerts.

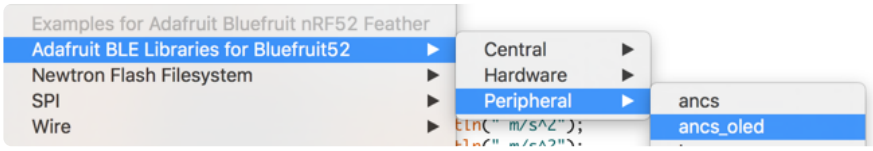
Sketch Requirements

In order to use this example sketch the following libraries must be installed on your system:

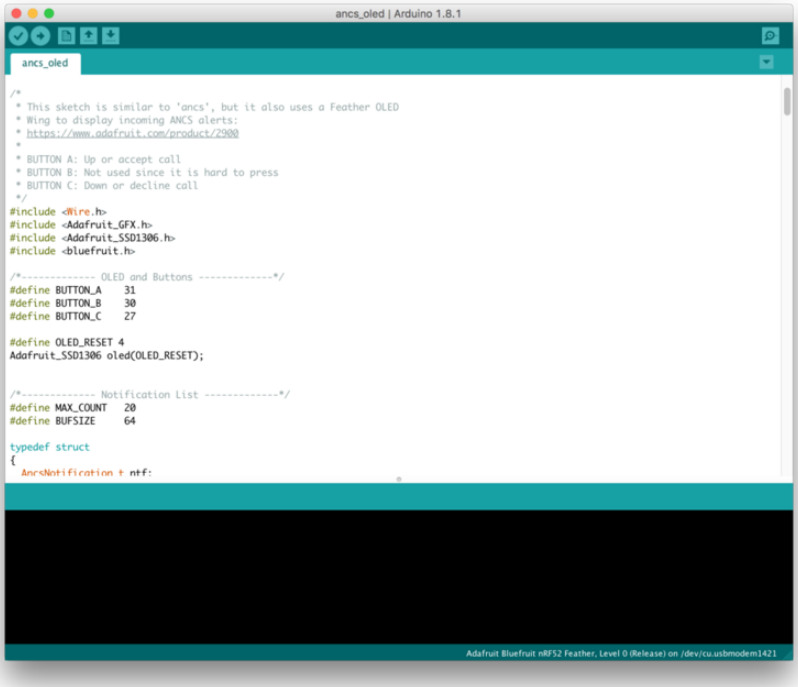
- Adafruit_GFX (https://adafru.it/xep) (Github source (https://adafru.it/aJa))
- Adafruit_SSD1306 (https://adafru.it/xep) (Github source (https://adafru.it/aHq))
- Version 0.6.0 or higher of the Bluefruit nRF52 BSP

Loading the Sketch

The ancs_oled sketch can be loaded via the examples menu under **Peripheral** > **ancs_oled**:



With the sketch loaded, you can build the firmware and then flash it to your device via the **Upload** button or menu option:



Make sure that the Adafruit_SSD1306.h file has the 'SSD1306_128_32' macro enabled. Running the sketch with 'SSD1306_128_64' set will cause corrupted data to appear on the OLED display.

Once the sketch is running on the nRF52 Feather you can proceed with the one-time pairing process, described below.

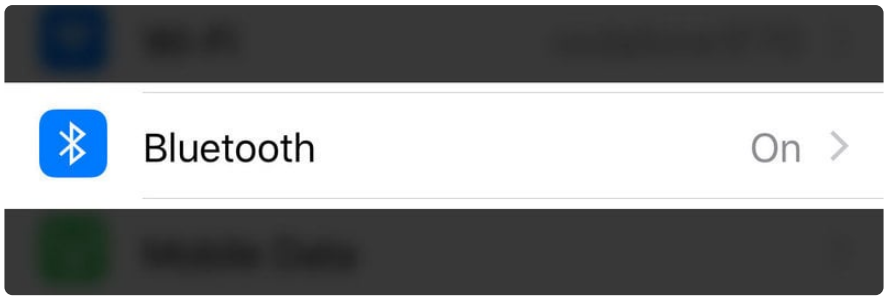
Pairing to your Mobile Device

Before you can start receiving notifications, you will need to 'pair' the nRF52 Feather and the mobile device.

The pairing process causes a set of keys to be exchanged and stored on the two devices so that each side knows it is talking to the same device it originally bonded with, and preventing any devices in the middle from eavesdropping on potentially sensitive data.

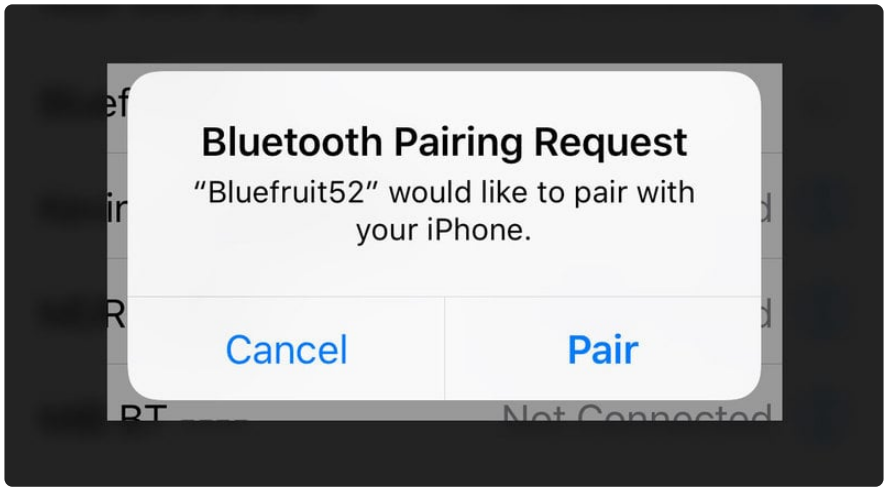
The one-time pairing process is described below, and assumes you are already running the ancs_oled sketch on your nRF52 device.

1. In the **Settings** app go to **Bluetooth**:

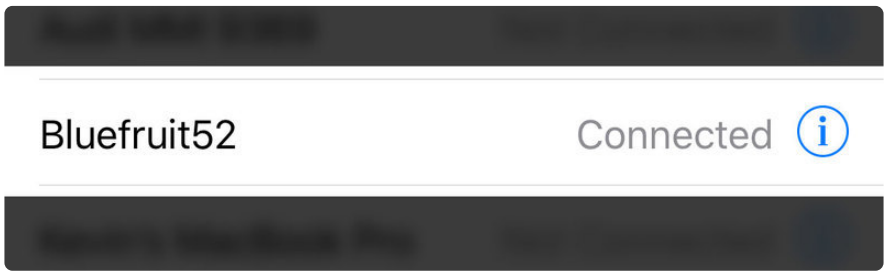


2. Scroll to the bottom of the list of 'My Devices' and click on **Bluefruit52** under **Other Devices**:

3. When the pairing dialog box comes up, click the **Pair** button:



4. Wait for the pairing process to complete, at which point **Bluefruit52** should appear in the **My Devices** list with the **Connected** status:



Once two devices have been paired, they will automatically reconnect to each other whenever they are in range and have their Bluetooth radios enabled.

Wait for Alerts

At this point, any alerts that the mobile device generates will be displayed on the OLED display along with the notification category and date:

Certain alerts (such as incoming calls) can also have actions associated with them, making use of the three buttons on the left-hand side of the display to decide which action to take.

In the ancs_oled example, we have a special section of code for incoming calls where you can accept or decline a call with an appropriate button press:

```
// Check buttons
uint32_t presedButtons = readPressedButtons();

if ( myNotifs[activeIndex].ntf.categoryID == ANCS_CAT_INCOMING_CALL )
{
  /* Incoming call event
   * - Button A to accept call
   * - Button C to decline call
   */
  if ( presedButtons &amp; bit(BUTTON_A) )
  {
    bleancs.actPositive(myNotifs[activeIndex].ntf.uid);
  }

  if ( presedButtons &amp; bit(BUTTON_C) )
  {
    bleancs.actNegative(myNotifs[activeIndex].ntf.uid);
  }
}
```

BLEClientCts

The Bluefruit nRF52 BSP codebase is undergoing active development based on customer feedback and testing. As such, the class documentation here is incomplete, and you should consult the Github repo for the latest code and API developments: https://goo.gl/LdEx62

BLEClientCts is a helper class that implements adopted Current Time Service (https://adafru.it/BiT) , which enables you to receive time from devices such as an iPhone or iPad.

API

```
// Callback Signatures
typedef void (*adjust_callback_t) (uint8_t reason);

BLEClientCts(void);

virtual bool  begin(void);
virtual bool  discover(uint16_t conn_handle);

bool getCurrentTime(void);
bool getLocalTimeInfo(void);

bool enableAdjust(void);
void setAdjustCallback(adjust_callback_t fp);

// https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.current_time.xml
struct ATTR_PACKED {
  uint16_t year;
  uint8_t  month;
  uint8_t  day;
  uint8_t  hour;
  uint8_t  minute;
  uint8_t  second;
  uint8_t  weekday;
  uint8_t  subsecond;
  uint8_t  adjust_reason;
} Time;

// https://www.bluetooth.com/specifications/gatt/viewer?
attributeXmlFile=org.bluetooth.characteristic.local_time_information.xml
struct ATTR_PACKED {
  int8_t  timezone;
  uint8_t dst_offset;
}LocalInfo;
```

Client CTS OLED Example

The client_cts_oled **(https://adafru.it/BiU)** example uses the Adafruit FeatherWing OLED (http://adafru.it/2900) to display received time.

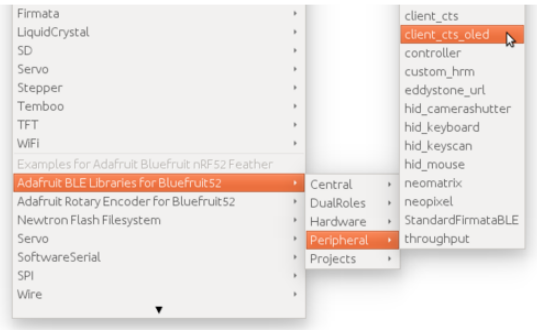
Sketch Requirements

In order to use this example sketch the following libraries must be installed on your system:

- Adafruit_GFX (https://adafru.it/xep) (Github source (https://adafru.it/aJa))
- Adafruit_SSD1306 (https://adafru.it/xep) (Github source (https://adafru.it/aHq))

Loading the Sketch

The client_cts_oled sketch can be loaded via the examples menu under **Peripheral** > **client_cts_oled**:



With the sketch loaded, you can build the firmware and then flash it to your device via the **Upload** button or menu option. Once the sketch is running on the nRF52 Feather you can proceed with the one-time pairing process, described below.

Make sure that the Adafruit_SSD1306.h file has the 'SSD1306_128_32' macro enabled. Running the sketch with 'SSD1306_128_64' set will cause corrupted data to appear on the OLED display.

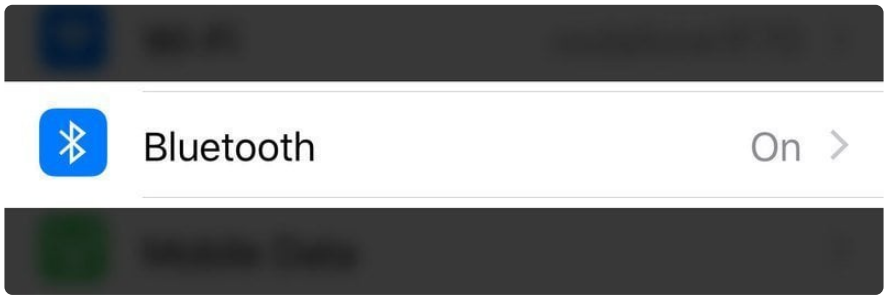
Pairing to your Mobile Device

Before you can start receiving notifications, you will need to 'pair' the nRF52 Feather and the mobile device.

The pairing process causes a set of keys to be exchanged and stored on the two devices so that each side knows it is talking to the same device it originally bonded with, and preventing any devices in the middle from eavesdropping on potentially sensitive data.

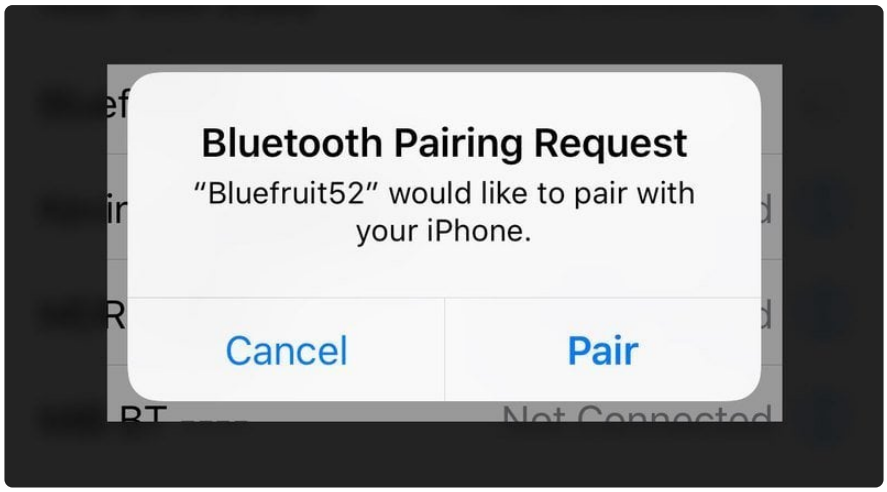
The one-time pairing process is described below, and assumes you are already running the ancs_oled sketch on your nRF52 device.

1. In the **Settings** app go to **Bluetooth**:

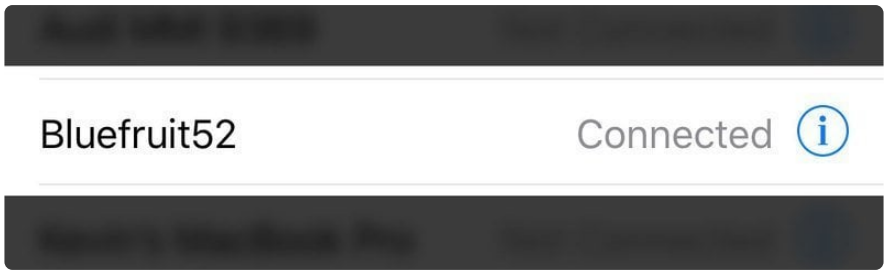


2. Scroll to the bottom of the list of 'My Devices' and click on **Bluefruit52** under **Other Devices**:

3. When the pairing dialog box comes up, click the **Pair** button:



4. Wait for the pairing process to complete, at which point **Bluefruit52** should appear in the **My Devices** list with the **Connected** status:



Once two devices have been paired, they will automatically reconnect to each other whenever they are in range and have their Bluetooth radios enabled.

Wait for Time Data

At this point, time data from the mobile device will be read and display on the the OLED. For demo purpose the sketch will read time data from mobile once every second. However, in reality, nRF52 should have an internal timer that keep track of second, and only read/sync with mobile after several hours or days, similar to how IP device got time from NTP server.

BLECentral

This page is a work in progress as the API is changing as we migrate to S132v5 (nRF52832) and S140 (nRF52840) and add better Central mode support.

The Central mode API is accessible via `Bluefruit.Central.*` and has the following public functions:

```
void begin(void);

/*-----------------------------------------------------------------*/
/* GAP
 *-----------------------------------------------------------------*/
bool     setConnInterval(uint16_t min, uint16_t max);
bool     setConnIntervalMS (uint16_t min_ms, uint16_t max_ms);

bool     connect(const ble_gap_evt_adv_report_t* adv_report);
bool     connect(const ble_gap_addr_t *peer_addr);
bool     disconnect(uint16_t conn_handle);

bool     connected (uint16_t conn_handle); // If connected to a specific peripheral
bool     connected (void);                 // If connected to any peripherals

/*------------- Callbacks -------------*/
void setConnectCallback   ( BLEGap::connect_callback_t    fp);
void setDisconnectCallback( BLEGap::disconnect_callback_t fp);
```

For examples of how to use the Central mode API, see the Central examples folder (https://adafru.it/BiV).

---

nRF52 ADC

The nRF52 family includes an adjustable 'successive-approximation ADC' which can be configured to convert data with up to 14-bit resolution (0..16383), and the reference voltage can be adjusted up to 3.6V internally.

The default values for the ADC are **10-bit resolution (0..1023)** with a **3.6V reference voltage**, meaning every digit returned from the ADC = 3600mV/1024 = **3.515625mV**.

Analog Reference Voltage

The internal reference voltage is 0.6V with a variable gain setting, and can be adjust via the **analogReference(...)** function, providing one of the following values:

- **AR_INTERNAL** (0.6V Ref * 6 = 0..3.6V) <-- DEFAULT
- **AR_INTERNAL_3_0** (0.6V Ref * 5 = 0..3.0V)
- **AR_INTERNAL_2_4** (0.6V Ref * 4 = 0..2.4V)
- **AR_INTERNAL_1_8** (0.6V Ref * 3 = 0..1.8V)
- **AR_INTERNAL_1_2** (0.6V Ref * 2 = 0..1.6V)
- **AR_VDD4** (VDD/4 REF * 4 = 0..VDD)

For example:

```
// Set the analog reference to 3.0V (default = 3.6V)
analogReference(AR_INTERNAL_3_0);
```

Analog Resolution

The ADC resolution can be set to 8, 10, 12 or 14 bits using the **analogReadResolution(...)** function, with the default value being 10-bit:

```
// Set the resolution to 12-bit (0..4095)
analogReadResolution(12); // Can be 8, 10, 12 or 14
```

Default ADC Example (10-bit, 3.6V Reference)

The original source for this code is included in the nRF52 BSP and can be viewed online here (https://adafru.it/zod).

```
#include <Arduino.h>
#include <Adafruit_TinyUSB.h> // for Serial

int adcin    = A5;
int adcvalue = 0;
float mv_per_lsb = 3600.0F/1024.0F; // 10-bit ADC with 3.6V input range

void setup() {
  Serial.begin(115200);
  while ( !Serial ) delay(10);   // for nrf52840 with native usb
}

void loop() {
  // Get a fresh ADC value
  adcvalue = analogRead(adcin);

  // Display the results
  Serial.print(adcvalue);
  Serial.print(" [");
  Serial.print((float)adcvalue * mv_per_lsb);
  Serial.println(" mV]");

  delay(100);
}
```

Advanced Example (12-bit, 3.0V Reference)

The original source for this code is included in the nRF52 BSP and can be viewed online here (https://adafru.it/zoe).

```
#include <Arduino.h>
#include <Adafruit_TinyUSB.h> // for Serial

#if defined ARDUINO_NRF52840_CIRCUITPLAY
#define  PIN_VBAT          A8   // this is just a mock read, we'll use the light
sensor, so we can run the test
#endif
```

```
uint32_t vbat_pin = PIN_VBAT;             // A7 for feather nRF52832, A6 for
nRF52840

#define VBAT_MV_PER_LSB   (0.73242188F)   // 3.0V ADC range and 12-bit ADC
resolution = 3000mV/4096

#ifdef NRF52840_XXAA
#define VBAT_DIVIDER      (0.5F)          // 150K + 150K voltage divider on VBAT
#define VBAT_DIVIDER_COMP (2.0F)          // Compensation factor for the VBAT
divider
#else
#define VBAT_DIVIDER      (0.71275837F)   // 2M + 0.806M voltage divider on VBAT =
(2M / (0.806M + 2M))
#define VBAT_DIVIDER_COMP (1.403F)        // Compensation factor for the VBAT
divider
#endif

#define REAL_VBAT_MV_PER_LSB (VBAT_DIVIDER_COMP * VBAT_MV_PER_LSB)


float readVBAT(void) {
  float raw;

  // Set the analog reference to 3.0V (default = 3.6V)
  analogReference(AR_INTERNAL_3_0);

  // Set the resolution to 12-bit (0..4095)
  analogReadResolution(12); // Can be 8, 10, 12 or 14

  // Let the ADC settle
  delay(1);

  // Get the raw 12-bit, 0..3000mV ADC value
  raw = analogRead(vbat_pin);

  // Set the ADC back to the default settings
  analogReference(AR_DEFAULT);
  analogReadResolution(10);

  // Convert the raw value to compensated mv, taking the resistor-
  // divider into account (providing the actual LIPO voltage)
  // ADC range is 0..3000mV and resolution is 12-bit (0..4095)
  return raw * REAL_VBAT_MV_PER_LSB;
}

uint8_t mvToPercent(float mvolts) {
  if(mvolts<3300)
    return 0;

  if(mvolts <3600) {
    mvolts -= 3300;
    return mvolts/30;
  }

  mvolts -= 3600;
  return 10 + (mvolts * 0.15F );  // thats mvolts /6.66666666
}

void setup() {
  Serial.begin(115200);
  while ( !Serial ) delay(10);   // for nrf52840 with native usb

  // Get a single ADC sample and throw it away
  readVBAT();
}

void loop() {
  // Get a raw ADC reading
```

```
  float vbat_mv = readVBAT();

  // Convert from raw mv to percentage (based on LIPO chemistry)
  uint8_t vbat_per = mvToPercent(vbat_mv);

  // Display the results

  Serial.print("LIPO = ");
  Serial.print(vbat_mv);
  Serial.print(" mV (");
  Serial.print(vbat_per);
  Serial.println("%)");

  delay(1000);
}
```

Memory Map

This page applies to BSP 0.8.0 and higher, which introduced bootloader v5.1.0 and S132 v5.x.x. For earlier releases (BSP release < 0.8.0) see bootloader v0.5.0 and S132 v2.x.x at the bottom of this page.

BSP release & Bootloader version

The memory usage depends on the version of the Softdevice and/or bootloader (single/dual bank). Following is the Bootloader and Softdevice version included with BSP release

- **0.9.x** :nRF52832 S132 v6.1.1 single bank andnRF52840 S140 v6.1.1 single bank

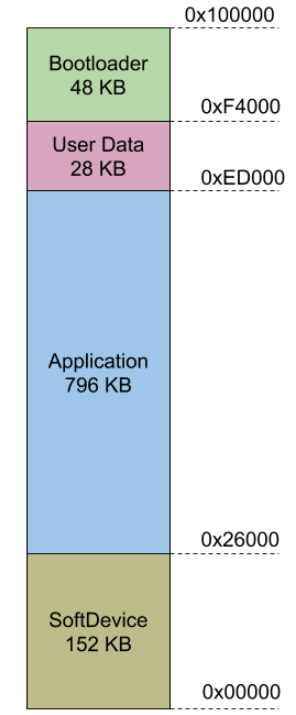
- **0.8.x** :nRF52832 S132 v2.0.1 dual banks and S132 v5.1.0 dual banks

- **0.7.x and older**:nRF52832 S132 v2.0.1 dual banks

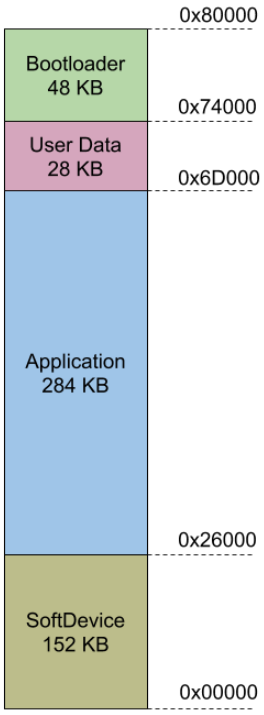
Starting from BSP 0.9.x only single banks will be used to maximize the flash storage

Flash Memory

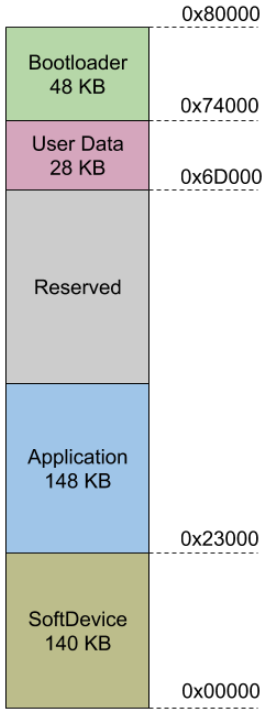
The nRF52832 has 512 KB flash, nRF52840 has 1024 KB flash. The flash layout varies as follows:



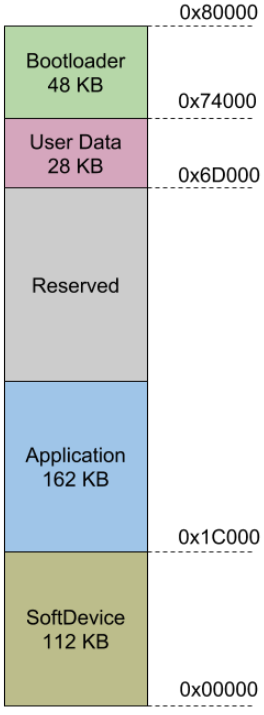
nRF52840 S140 v6



nRF52832 S132 v6



nRF52832 S132 v5



nRF52832 S132 v2

- **SoftDevice**: This section of flash memory contains the **Soft Device**, which is Nordic's black box Bluetooth Low Energy stack.
- **Application**: This section of flash memory stores the user sketches that you compile in the Arduino IDE.

- **Reserved**: This section of flash memory is kept empty to enable **dual banks** safe firmware updates. Whenever you try to update the Application are, the new application data will first be written to the free memory section, and the verified before it is swapped out with the current application code. This is to ensure that DFU complete successfully and that the entire image is safely store on the device to avoid losing the application code. This region is not used by **single bank** bootloader
- **User Data**: This 28KB section of flash memory is reserved for config settings. It uses an open source file system called Little File System (https://adafru.it/C-n), which is a part of ARM Mbed OpenSource to store bonding data. For example, when you bond the nRF52 with another Central device.
- **DFU Bootloader**: This section of flash memory stores the actual bootloader code that will be executed by the MBR described earlier.

SRAM Layout

The nRF52832 has 64KB of SRAM available, and actual memory use will depend on your project, although the stack and heap memory locations are described below:

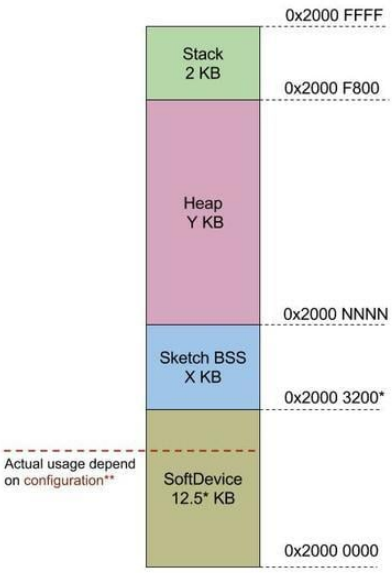
- **Soft Device:** amount of SRAM exclusive allocated for SoftDevice by Linker script. The size is subject to change and varies by releases. For BSP release **0.8.0** it is **12.5 KB**. However, the actual memory required by the SoftDevice depends on the run-time configuration determined by Bluefruit's configNNN() API.
- **Sketch BSS:** static and global data used by your sketch.
- **Heap Memory:** The largest memory region, this is used for allocating the real time operating systems (RTOS) thread stack, malloc() etc. The size, based on the variables shown below, is **Y = 64 - 12.5 - X - 2 (KB),** where 12.5KB will vary depending on the SoftDevice used.
- **Stack Memory**: Used by non RTOS thread code, this is mostly for Interrupt Service Routines (ISRs) and SoftDevice API calls. The current size is **2 KB**.

Functions affecting SoftDevice SRAM usage

The Bluefruit nRF52 **configNNN**() functions set the behavior of SoftDevice, thus determining the total SRAM usage. **These functions must be called before begin().**

- **configUuid128Count**() : Defines the number of UUID128 entries that the SoftDevice supports, e.g Bleuart, BleMidi, or other services and characteristics. **Default value is 10.**

- **configAttrTableSize():** The total size of the attribute table, which holds services and characteristics. If your application needs lots of characteristics you may need to increase this. **Default value is 2048 bytes.**
- **configPrphConn(), configPrphBandwidth():** These function set the parameters that determine the bandwidth for peripheral's connections. configPrphBandwidth() is a convenient helper that calls configPrphConn() with appropriate parameters.
- **configCentralConn(), configCentralBandwidth():** These functions set the parameters that determine the bandwidth for central mode connections. configCentralBandwidth() is a convenient helper that calls configCentralConn() with appropriate parameters.
- **begin():** Bluefruit nRF52's **begin()** function also affects the bandwidth since it takes 2 (optional) parameters. The first one is the number of concurrent connections for peripheral links (to mobile phones, your computer, etc.), the second one is the number of central links (to BLE accessories, or another feather52 running in peripheral mode). **The maximum number of concurrent connections for SoftDevice v5.x is 20**.



Bluefruit nRF52 S132 v5.x.x SRAM Usage

If you run into an error message saying "SoftDevice require more SRAM than provided by linker", try altering your system config -- for ex. lower bandwidth, fewer connection or a smaller attribute table size. Another advanced option is to modify the linker script, but this should be done with care and knowledge of what you are changing.

```
[CFG] SoftDevice config requires more SRAM than provided by the linker.
App Ram Start must be at least 0x20004180 (provided 0x20003200).
Please update linker file or re-config SoftDevice.
```

Software Resources

To help you get your Bluefruit LE module talking to other Central devices, we've put together a number of open source tools for most of the major platforms supporting Bluetooth Low Energy.

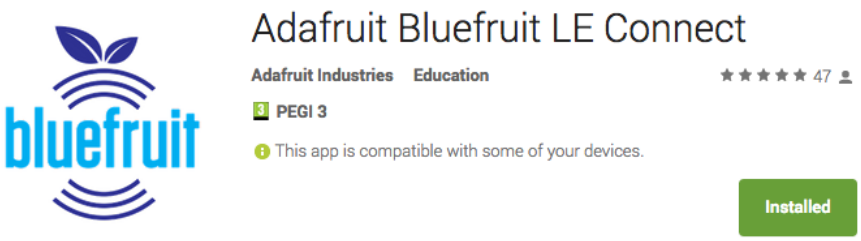
Bluefruit LE Client Apps and Libraries

Adafruit has put together the following mobile or desktop apps and libraries to make it as easy as possible to get your Bluefruit LE module talking to your mobile device or laptop, with full source available where possible:

Bluefruit LE Connect (https://adafru.it/f4G) (Android/Java)

Bluetooth Low Energy support was added to Android starting with Android 4.3 (though it was only really stable starting with 4.4), and we've already released Bluefruit LE Connect to the Play Store (https://adafru.it/f4G).

The full source code (https://adafru.it/fY9) for Bluefruit LE Connect for Android is also available on Github to help you get started with your own Android apps. You'll need a recent version of Android Studio (https://adafru.it/fYa) to use this project.



Bluefruit LE Connect (https://adafru.it/f4H) (iOS/Swift)

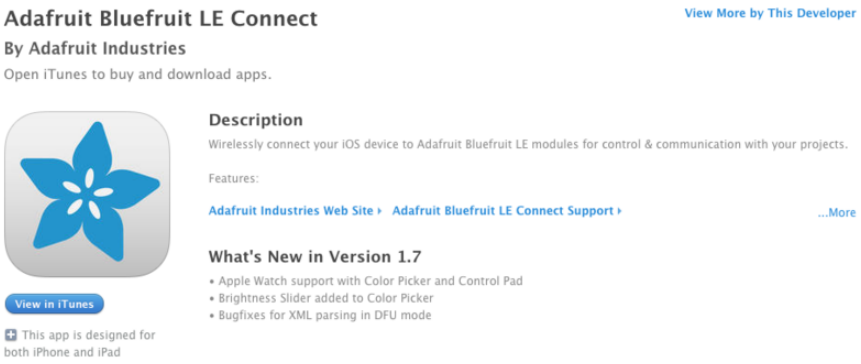
Apple was very early to adopt Bluetooth Low Energy, and we also have an iOS version of the Bluefruit LE Connect (https://adafru.it/f4H) app available in Apple's app store.

The full swift source code for Bluefruit LE Connect for iOS is also available on Github. You'll need XCode and access to Apple's developper program to use this project:

- Version 1.x source code: https://github.com/adafruit/Bluefruit_LE_Connect (https://adafru.it/ddv)

- Version 2.x source code: https://github.com/adafruit/Bluefruit_LE_Connect_v2 (https://adafru.it/o9E)

Version 2.x of the app is a complete rewrite that includes iOS, OS X GUI and OS X command-line tools in a single codebase.

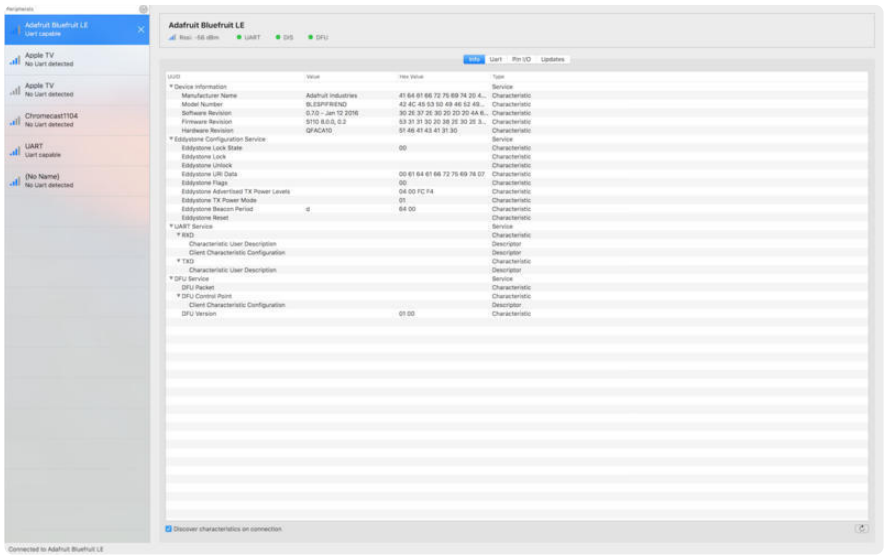


Bluefruit LE Connect for OS X (https://adafru.it/o9F) (Swift)

This OS X desktop application is based on the same V2.x codebase as the iOS app, and gives you access to BLE UART, basic Pin I/O and OTA DFU firmware updates from the convenience of your laptop or mac.

This is a great choice for logging sensor data locally and exporting it as a CSV, JSON or XML file for parsing in another application, and uses the native hardware on your computer so no BLE dongle is required on any recent mac.

The full source is also available on Github (https://adafru.it/o9E).



Bluefruit LE Command Line Updater for OS X (https://adafru.it/pLF) (Swift)

This experimental command line tool is unsupported and provided purely as a proof of concept, but can be used to allow firmware updates for Bluefruit devices from the command line.

This utility performs automatic firmware updates similar to the way that the GUI application does, by checking the firmware version on your Bluefruit device (via the Device Information Service), and comparing this against the firmware versions available online, downloading files in the background if appropriate.

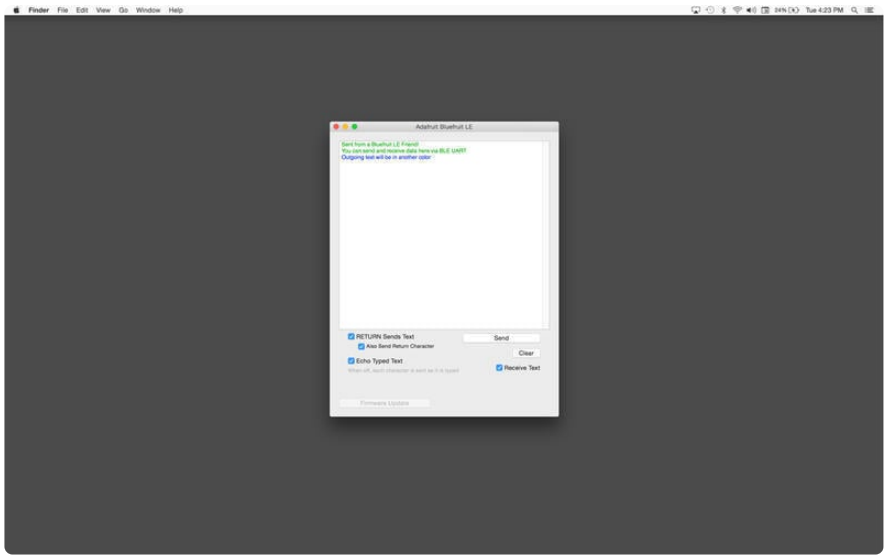
Simply install the pre-compiled tool via the DMG file (https://adafru.it/pLF) and place it somewhere in the system path, or run the file locally via './bluefruit' to see the help menu:

```
$ ./bluefruit
bluefruit v0.3
Usage:
    bluefruit &lt;command&gt; [options...]

Commands:
    Scan peripherals:   scan
    Automatic update:   update [--enable-beta] [--uuid &lt;uuid&gt;]
    Custom firmware:    dfu --hex &lt;filename&gt; [--init &lt;filename&gt;] [--
uuid &lt;uuid&gt;]
    Show this screen:   --help
    Show version:       --version

Options:
    --uuid &lt;uuid&gt;    If present the peripheral with that uuid is used. If not
present a list of peripherals is displayed
    --enable-beta    If not present only stable versions are used

Short syntax:
    -u = --uuid, -b = --enable-beta, -h = --hex, -i = --init, -v = --version, -? =
--help
```

Deprecated: Bluefruit Buddy (https://adafru.it/mCn) (OS X)

This native OS X application is a basic proof of concept app that allows you to connect to your Bluefruit LE module using most recent macbooks or iMacs. You can get basic information about the modules and use the UART service to send and receive data.

The full source for the application is available in the github repo at Adafruit_BluefruitLE_OSX (https://adafru.it/mCo).



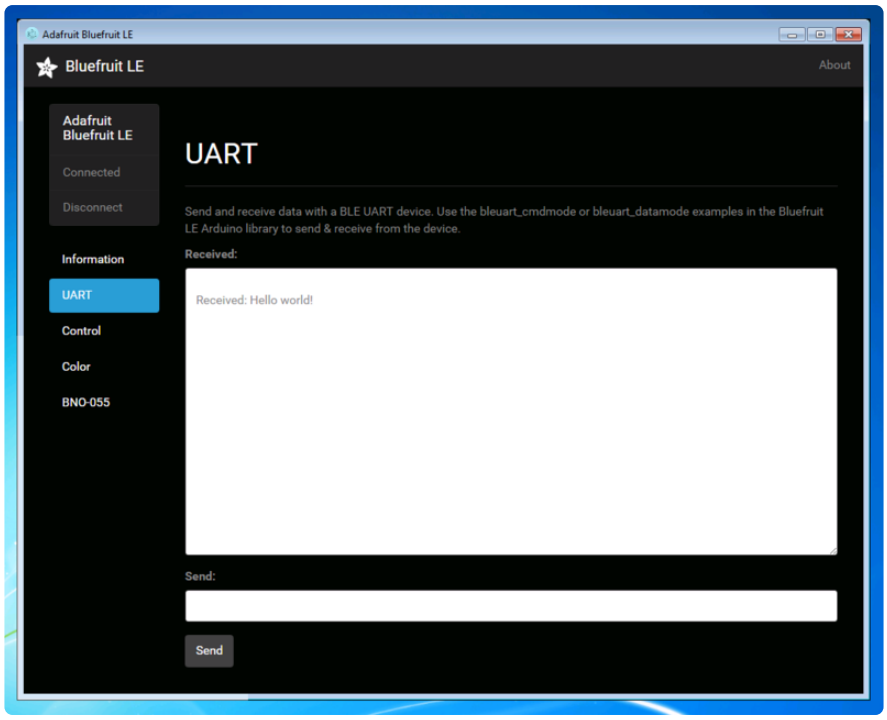
ABLE (https://adafru.it/ijB) (Cross Platform/Node+Electron)

ABLE (https://adafru.it/ijB) (Adafruit Bluefruit LE Desktop) is a cross-platform desktop application based on Sandeep Misty's noble library (https://adafru.it/ijC) and the Electron (https://adafru.it/ijD) project from Github (used by Atom).

It runs on OS X, Windows 7+ and select flavours of Linux (Ubuntu tested locally). Windows 7 support is particularly interesting since Windows 7 has no native support for Bluetooth Low Energy but the noble library talks directly to supported Bluetooth 4.0 USB dongles (http://adafru.it/1327) to emulate BLE on the system (though at this stage it's still in early BETA and drops the connection and takes more care to work with).

This app allows you to collect sensor data or perform many of the same functionality offered by the mobile Bluefruit LE Connect apps, but on the desktop.

The app is still in BETA, but full source (https://adafru.it/ijE) is available in addition to the easy to use pre-compiled binaries (https://adafru.it/ijB).



Bluefruit LE Python Wrapper (https://adafru.it/fQF)

As a proof of concept, we've played around a bit with getting Python working with the native Bluetooth APIs on OS X and the latest version of Bluez on certain Linux targets.

There are currently example sketches showing how to retreive BLE UART data as well as some basic details from the Device Information Service (DIS).

This isn't an actively support project and was more of an experiment, but if you have a recent Macbook or a Raspberry Pi and know Python, you might want to look at Adafruit_Python_BluefruitLE (https://adafru.it/fQF) in our github account.

Debug Tools

If your sense of adventure gets the better of you, and your Bluefruit LE module goes off into the weeds, the following tools might be useful to get it back from unknown lands.

These debug tools are provided purely as a convenience for advanced users for device recovery purposes, and are not recommended unless you're OK with potentially bricking your board. Use them at your own risk.

AdaLink (https://adafru.it/fPq) (Python)

This command line tool is a python-based wrapper for programming ARM MCUs using either a Segger J-Link (https://adafru.it/fYU) or an STLink/V2 (http://adafru.it/2548). You can use it to reflash your Bluefruit LE module using the latest firmware from the Bluefruit LE firmware repo (https://adafru.it/edX).

Details on how to use the tool are available in the readme.md file on the main Adafruit_Adalink (https://adafru.it/fPq) repo on Github.

Completely reprogramming a Bluefruit LE module with AdaLink would require four files, and would look something like this (using a JLink):

```
adalink nrf51822 --programmer jlink --wipe
  --program-hex "Adafruit_BluefruitLE_Firmware/softdevice/
s110_nrf51_8.0.0_softdevice.hex"
  --program-hex "Adafruit_BluefruitLE_Firmware/bootloader/bootloader_0002.hex"
  --program-hex "Adafruit_BluefruitLE_Firmware/0.6.7/blefriend32/
blefriend32_s110_xxac_0_6_7_150917_blefriend32.hex"
  --program-hex "Adafruit_BluefruitLE_Firmware/0.6.7/blefriend32/
blefriend32_s110_xxac_0_6_7_150917_blefriend32_signature.hex"
```

You can also use the AdaLink tool to get some basic information about your module, such as which SoftDevice is currently programmed or the IC revision (16KB SRAM or 32KB SRAM) via the --info command:

```
$ adalink nrf51822 -p jlink --info
Hardware ID : QFACA10 (32KB)
Segger ID   : nRF51822_xxAC
SD Version  : S110 8.0.0
Device Addr : **:**:**:**:**:**
Device ID   : ****************
```

Adafruit nRF51822 Flasher (https://adafru.it/fVL) (Python)

Adafruit's nRF51822 Flasher is an internal Python tool we use in production to flash boards as they go through the test procedures and off the assembly line, or just testing against different firmware releases when debugging.

It relies on AdaLink or OpenOCD beneath the surface (see above), but you can use this command line tool to flash your nRF51822 with a specific SoftDevice, Bootloader and Bluefruit firmware combination.

It currently supports using either a Segger J-Link or STLink/V2 via AdaLink, or GPIO on a Raspberry Pi (https://adafru.it/fVL) if you don't have access to a traditional ARM

SWD debugger. (A pre-built version of OpenOCD for the RPi is included in the repo since building it from scratch takes a long time on the original RPi.)

We don't provide active support for this tool since it's purely an internal project, but made it public just in case it might help an adventurous customer debrick a board on their own.

```
$ python flash.py --jtag=jlink --board=blefriend32 --softdevice=8.0.0 --
bootloader=2 --firmware=0.6.7
jtag        : jlink
softdevice  : 8.0.0
bootloader  : 2
board       : blefriend32
firmware    : 0.6.7
Writing Softdevice + DFU bootloader + Application to flash memory
adalink -v nrf51822 --programmer jlink --wipe --program-hex
"Adafruit_BluefruitLE_Firmware/softdevice/s110_nrf51_8.0.0_softdevice.hex" --
program-hex "Adafruit_BluefruitLE_Firmware/bootloader/bootloader_0002.hex" --
program-hex "Adafruit_BluefruitLE_Firmware/0.6.7/blefriend32/
blefriend32_s110_xxac_0_6_7_150917_blefriend32.hex" --program-hex
"Adafruit_BluefruitLE_Firmware/0.6.7/blefriend32/
blefriend32_s110_xxac_0_6_7_150917_blefriend32_signature.hex"
...
```

Downloads

The following resources may be useful working with the Bluefruit nRF52 Feather:

- Adafruit_nRF52_Arduino (https://adafru.it/vaF): The core code for this device (hosted on Github)
- nRF52 Example Sketches (https://adafru.it/vaK): Browse the example code from the core repo on Github
- nRF52832 Product Specification v1.4 (https://adafru.it/ErI): Key technical documentation for the nRF52832 SoC
- EagleCAD PCB files on GitHub (https://adafru.it/vbH)

Feather Bluefruit NRF52 Pinout Diagram

https://adafru.it/z4c

MyNewt Bootloader

This is if you want to program in MyNewt support - the bootloader will not work with Arduino!

feather52_boot_bleuart.bin

https://adafru.it/Ep4

You can use adalink and a jlink to program this in via SWD to address 0 with something like:

```
adalink -v nrf52832 -p jlink -b feather52_boot_bleuart.bin 0
```

(Don't forget you may have to erase the chip)

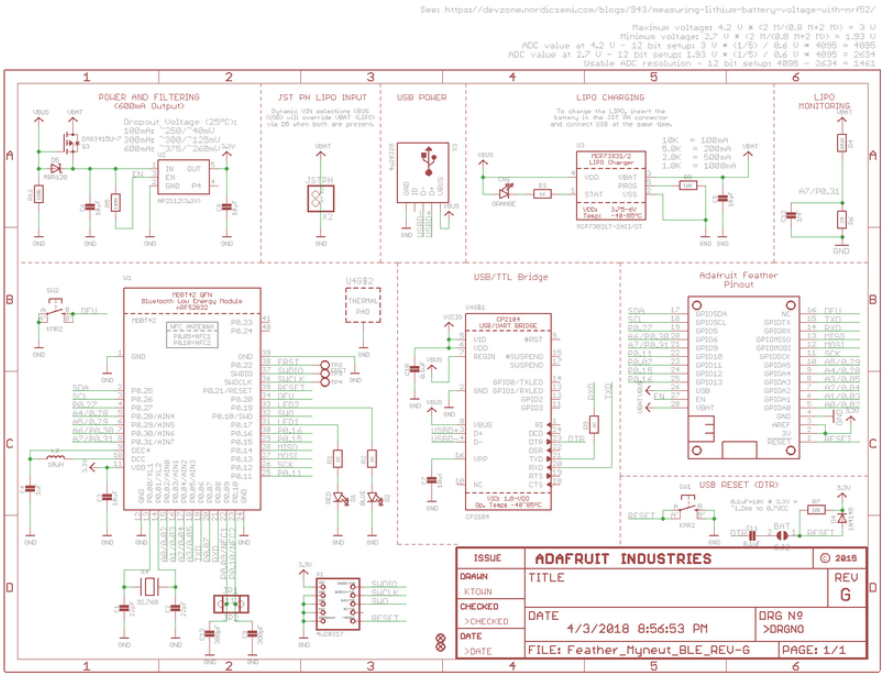
Module Details

The Bluefruit nRF52 Feather uses the MDBT42Q module from Raytac. Details on the module, including FCC and other certifications are available in the document below:

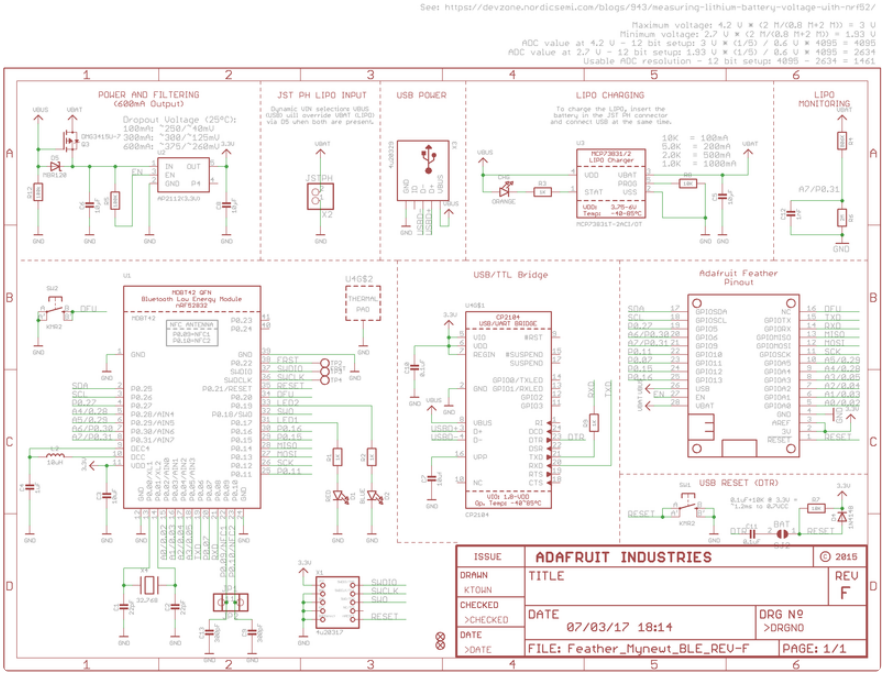
MDBT42Q-Version_B.pdf

https://adafru.it/vbb

Schematic



Schematic (pre March 16 2018)



FAQs

NOTE: For FAQs relating to the **BSP**, see the dedicated BSP FAQ list (https://adafru.it/vnF).

What are the differences between the nRF51 and nRF52 Bluefruit boards? Which one should I be using?

The two board families take very different design approaches.

All of the nRF51 based modules are based on an AT command set (over UART or SPI), and require two MCUs to run: the nRF51 hosting the AT command parser, and an external MCU sending AT style commands.

The nRF52 boards run code directly on the nRF52, executing natively and calling the Nordic S132 SoftDevice (their proprietary Bluetooth Low Energy stack) directly. This allows for more efficient code since there is no intermediate AT layer or transport, and also allows for lower overall power consumption since only a single device is involved.

The nRF52 will generally give you better performance, but for situation where you need to use an MCU with a feature the nRF52 doesn't have (such as USB), the nRF51 based boards will still be the preferable solution.

Can I run nRF51 Bluefruit sketches on the nRF52?

No. The two board families are fundamentally different, and have entirely separate APIs and programming models. If you are migrating from the nRF51 to the nRF52, you will need to redesign your sketches to use the newer API, enabling you to build code that runs natively on the nRF52832 MCU.

Can I use the nRF52 as a Central to connect to other BLE peripherals?

The S132 Soft Device and the nRF52832 HW support Central mode, so yes this is possible. At this early development stage, though, there is only bare bones support for Central mode in the Adafruit nRF52 codebase, simply to test the HW and S132 and make sure that everything is configured properly. An example is provided of listening for incoming advertising packets, printing the packet contents and meta-data out to the Serial Monitor. We hope to add further Central mode examples in the future, but priority has been given to the Peripheral API and examples for the initial release.

How are Arduino sketches executed on the nRF52? Can I do hard real time processing (bit-banging NeoPixels, Software Serial etc.)?

In order to run Arduino code on the nRF52 at the same time as the low level Bluetooth Low Energy stack, the Bluefruit nRF52 Feather uses FreeRTOS as a task scheduler. The scheduler will automatically switch between tasks, assigning clock cycles to the highest priority task at a given moment. This process is generally transparent to you, although it can have implications if you have hard real time requirements. There is no guarantee on the nRF52 to meet hard timing requirements when the radio is enabled an being actively used for Bluetooth Low Energy. This isn't possible on the nRF52 even without FreeRTOS, though, since the SoftDevice (Nordic's propietary binary blob stack) has higher priority than any user code, including control over interrupt handlers.

Can I use GDB to debug my nRF52?

You can, yes, but it will require a Segger J-Link (that's what we've tested against anyway, other options exist), and it's an advanced operation. But if you're asking about it, you probably know that.

Assuming you have the Segger J-Link drivers installed, you can start Segger's GDB Server from the command line as follows (OSX/Linux used here):

```
$ JLinkGDBServer -device nrf52832_xxaa -if swd -speed auto
```

Then open a new terminal window, making sure that you have access to `gcc-arm-none-eabi-gdb` from the command line, and enter the following command:

```
$ ./arm-none-eabi-gdb something.ino.elf
```

`` `something.ino.elf` `` is the name of the .elf file generated when you built your sketch. You can find this by enabling 'Show verbose output during: [x] compilation' in the Arduino IDE preferences. You CAN run GDB without the .elf file, but pointing to the .elf file will give you all of the meta data like displaying the actual source code at a specific address, etc.

Once you have the `(gdb)` prompt, enter the following command to connect to the Segger GDB server (updating your IP address accordingly, since the HW isn't necessarily local!):

```
(gdb) target remote 127.0.0.1:2331
```

If everything went well, you should see the current line of code where the device is halted (normally execution on the nRF52 will halt as soon as you start the Segger GDB Server).

At this point, you can send GDB debug commands, which is a tutorial in itself! As a crash course, though:

- To continue execution, type '`monitor go`' then '`continue`'
- To stop execution (to read register values, for example.), type '`monitor halt`'
- To display the current stack trace (when halted) enter '`bt`'
- To get information on the current stack frame (normally the currently executing function), try these:
    - `info frame`: Display info on the current stack frame
    - `info args`: Display info on the arguments passed into the stack frame
    - `info locals`: Display local variables in the stack frame
    - `info registers`: Dump the core ARM register values, which can be useful for debugging specific fault conditions

---

Are there any other cross platform or free debugging options other than GDB?

If you have a Segger J-Link (https://adafru.it/w5c), you can also use Segger's OZone debugger GUI (https://adafru.it/w5d) to interact with the device, though check the license terms since there are usage restrictions depending on the J-Link module you have.

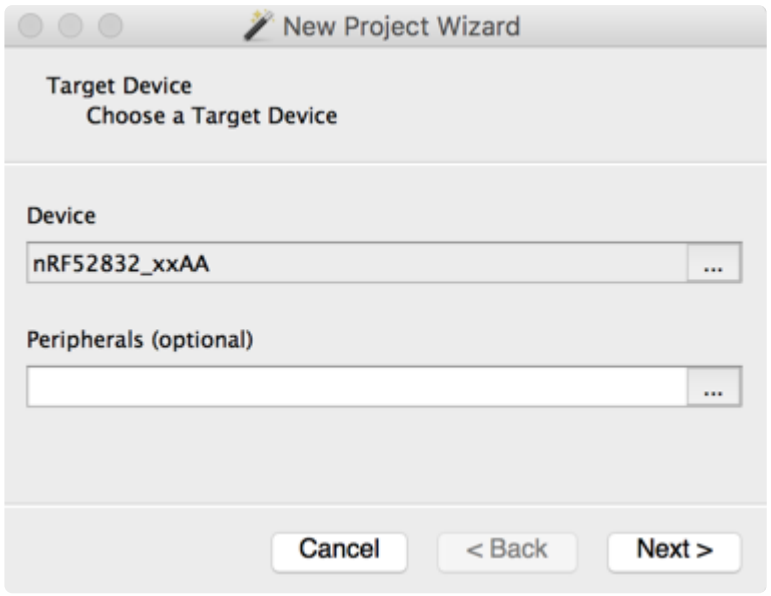
You will need to connect your nRF52 to the J-Link via the SWD and SWCLK pins on the bottom of the PCB, or if you are OK with fine pitch soldering via the SWD header.

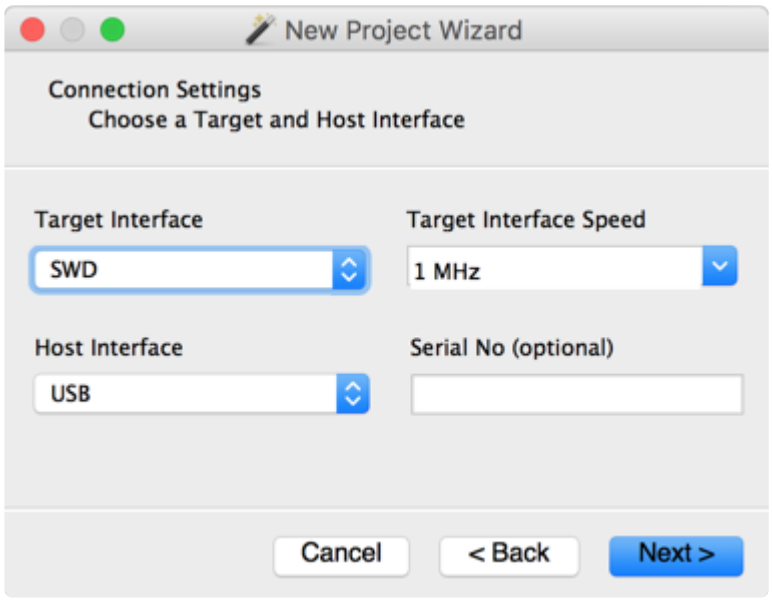
You can either solder on a standard 2x5 SWD header (http://adafru.it/752) on the pad available in the board, or you can solder wires to the SWD and SWCLK pads on the bottom of the PCB and use an SWD Cable Breakout Board (http://adafru.it/2743), or just connect cables directly to your J-Link via some other means.

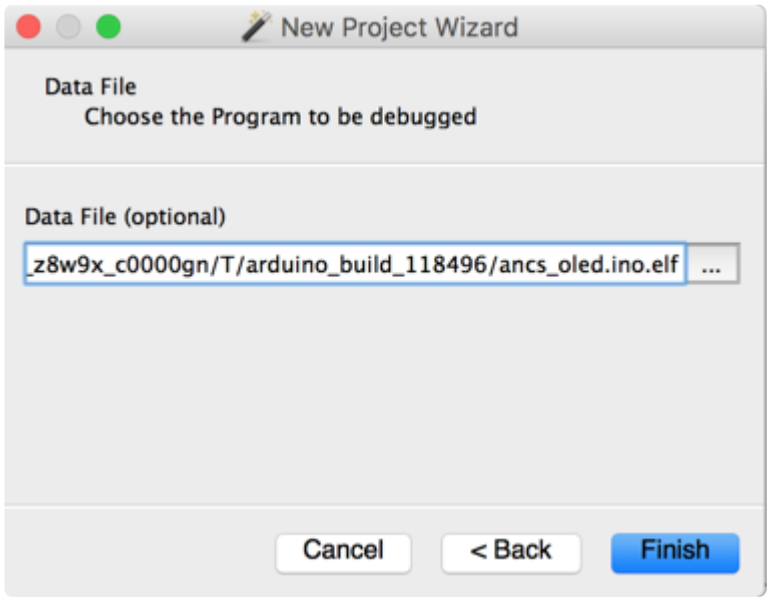
You will also need to connect the **VTRef** pin on the JLink to **3.3V** on the Feather to let the J-Link know what voltage level the target has, and share a common GND by connecting the GND pins on each device.

Before you can start to debug, you will need to get the .elf file that contains all the debug info for your sketch. You can find this file by enabling **Show Verbose Output During: compilation** in the **Arduino Preferences** dialogue box. When you build your sketch, you need to look at the log output, and find the .elf file, which will resemble something like this (it will vary depending on the OS used): `/var/folders/86/hb2vp14n5_5_yvdz_z8w9x_c0000gn/T/arduino_build_118496/ancs_oled.ino.elf`

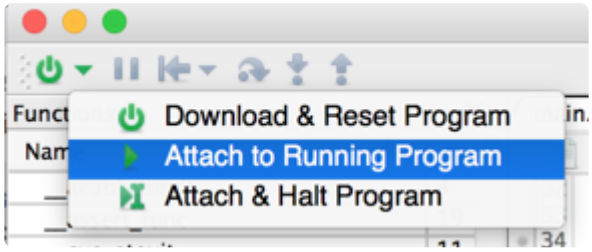
In the OZone New Project Wizard, when prompted to select a target device in OZone select **nRF52832_xxAA**, then make sure that you have set the Target Interface for the debugger to **SWD**, and finally point to the .elf file above:

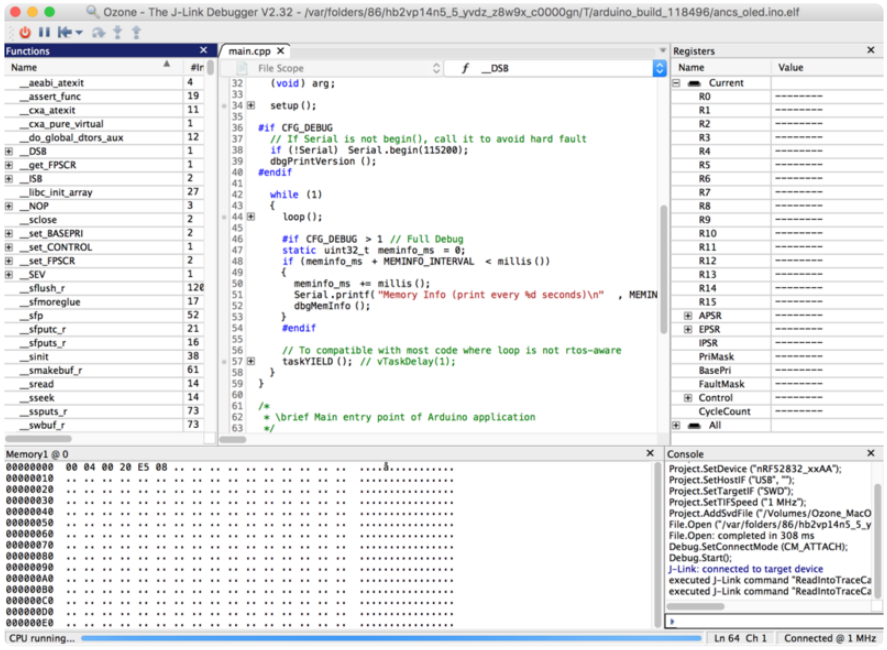

New Project Wizard
Target Device
Choose a Target Device
Device
nRF52832_xxAA
...
Peripherals (optional)
...
Cancel
< Back
Next >


New Project Wizard
Connection Settings
Choose a Target and Host Interface
Target Interface
SWD
Target Interface Speed
1 MHz
Host Interface
USB
Serial No (optional)
Cancel
< Back
Next >

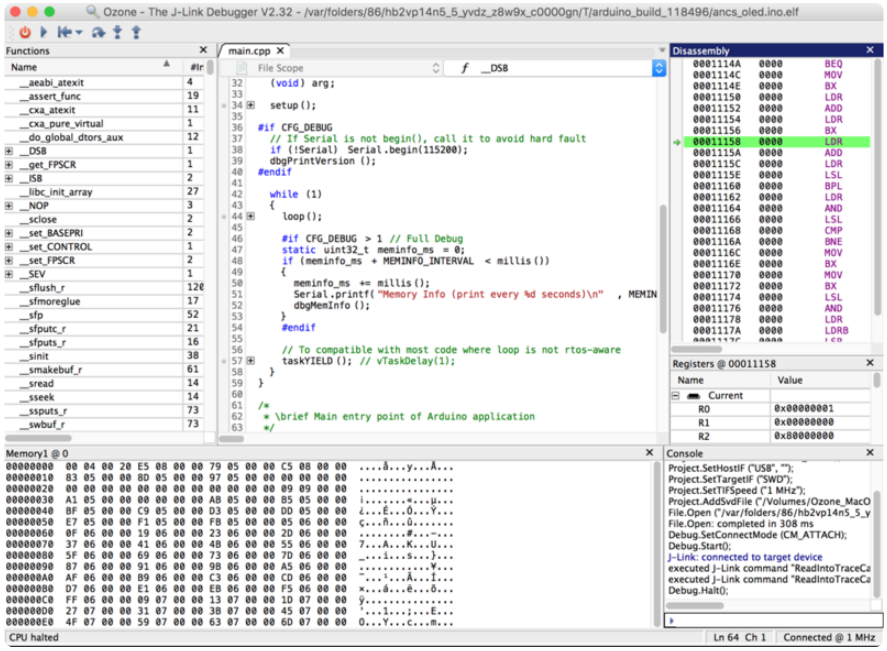

New Project Wizard
Data File
Choose the Program to be debugged
Data File (optional)
_z8w9x_c0000gn/T/arduino_build_118496/ancs_oled.ino.elf
...
Cancel
< Back
Finish

Next select the **Attach to running program** option in the top-left hand corner, or via the menu system, which will cause the debugger to connect to the nRF52 over SWD:

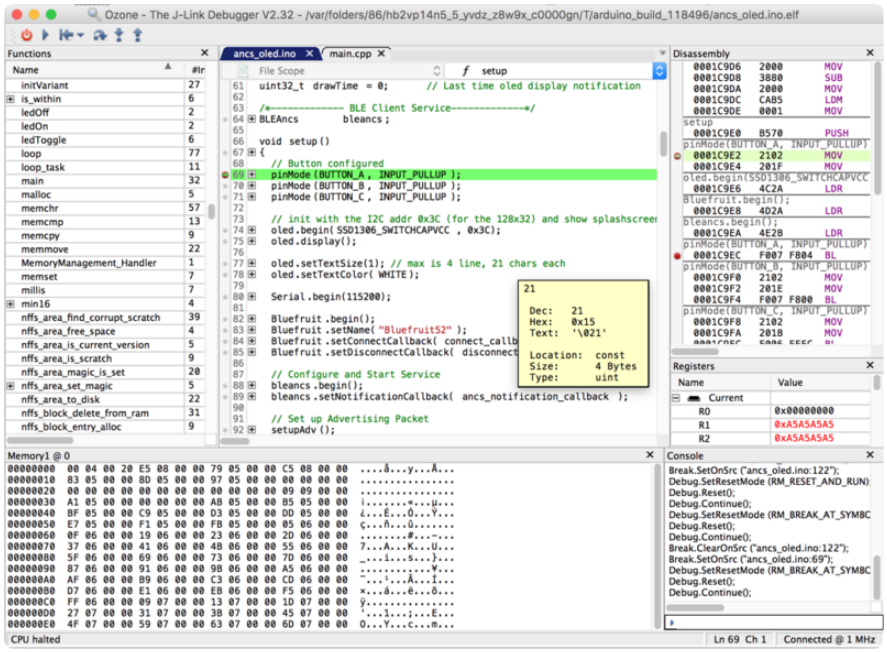




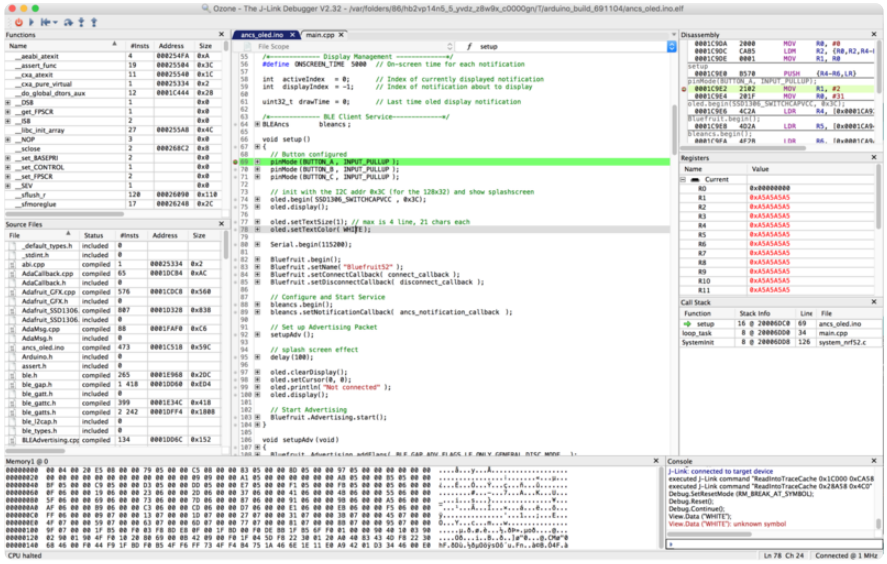
At this point, you can click the **PAUSE** icon to stop program execution, and then analyze variables, or set breakpoints at appropriate locations in your program execution, and debug as you would with most other embedded IDEs!



Clicking on the left-hand side of the text editor will set a breakpoint on line 69 in the image below, for example, and the selecting **Debug > Reset > Reset & Run** from the menu or icon will cause the board to reset, and you should stop at the breakpoint you set:



You can experiment with adding some of the other debug windows and options via the **View** menu item, such as the **Call Stack** which will show you all of the functions that were called before arriving at the current breakpoint:



Can I make two Bluefruit nRF52's talk to each other?

Yes, by running one board in peripheral mode and one board in central mode, where the central will establish a connection with the peripheral board and you can communicate using BLE UART or a custom service. See the following Central BLE

UART example to help you get started: https://github.com/adafruit/Adafruit_nRF52_Arduino/tree/master/libraries/Bluefruit52Lib/examples/Central

On Linux I'm getting 'arm-none-eabi-g++: no such file or directory', even though 'arm-none-eabi-g++' exists in the path specified. What should I do?

This is probably caused by a conflict between 32-bit and 64-bit versions of the compiler, libc and the IDE. The compiler uses 32-bit binaries, so you also need to have a 32-bit version of libc installed on your system (details (https://adafru.it/vnE)). Try running the following commands from the command line to resolve this:

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386
```

what should I do when Arduino failed to upload sketch to my Feather ?

If you get this error:

```
Timed out waiting for acknowledgement from device.

Failed to upgrade target. Error is: No data received on serial port. Not able to proceed.
Traceback (most recent call last):
  File "nordicsemi\__main__.py", line 294, in serial
  File "nordicsemi\dfu\dfu.py", line 235, in dfu_send_images
  File "nordicsemi\dfu\dfu.py", line 203, in _dfu_send_image
  File "nordicsemi\dfu\dfu_transport_serial.py", line 155, in send_init_packet
  File "nordicsemi\dfu\dfu_transport_serial.py", line 243, in send_packet
  File "nordicsemi\dfu\dfu_transport_serial.py", line 282, in get_ack_nr
nordicsemi.exceptions.NordicSemiException: No data received on serial port. Not able to proceed.
```

This is probably caused by the **bootloader** version mismatched on your feather and installed BSP. Due to the difference in flash layout (more details (https://adafru.it/Dsy)) and Softdevice API (which is bundled with bootloader), sketch built with selected bootloader can only upload to board having the same version. In short,

you need to **upgrade/burn bootloader to match** on your Feather, follow above Update The Bootloader (https://adafru.it/Dsx) guide

It only has to be done once to update your Feather

Do Feather/Metro nRF52832 and nRF52840 support BLE Mesh ?

They all support BLE Mesh, but we don't provide Arduino library for Mesh. You need to write code based on Nordic sdk mesh.

Unable to upload sketch/update bootloader with macOS

If you get error similar to this:

```
Arduino: 1.8.8 (Mac OS X), Board: "Adafruit Bluefruit nRF52832 Feather, 0.2.9 (s132 6.1.1), Level 0 (Release)"

[1716] Error loading Python lib '/var/folders/gw/b0cg4zm508qf_rf2m655gd3m0000gn/T/_MEIE6ec69/Python': dlopen: dlopen(/var/folders/gw/b0cg4zm508qf_rf2m655gd3m0000gn/T/_MEIE6ec69/Python, 10): Symbol not found: _futimens
  Referenced from: /var/folders/gw/b0cg4zm508qf_rf2m655gd3m0000gn/T/_MEIE6ec69/Python (which was built for Mac OS X 10.13)
  Expected in: /usr/lib/libSystem.B.dylib
 in /var/folders/gw/b0cg4zm508qf_rf2m655gd3m0000gn/T/_MEIE6ec69/Python
exit status 255
Error compiling for board Adafruit Bluefruit nRF52832 Feather.
```

It is probably due to the pre-built adafruit-nrfutil cannot run on your Mac. The binary is generated on MacOS 10.13, if your Mac is older than that. Please update your macOS, or you could follow this repo's readme here https://github.com/adafruit/Adafruit_nRF52_nrfutil (https://adafru.it/Cau)to manual install it (tried with pip3 first, or install from source if it doesn't work). Then use the installed binary to replace the one in the BSP.