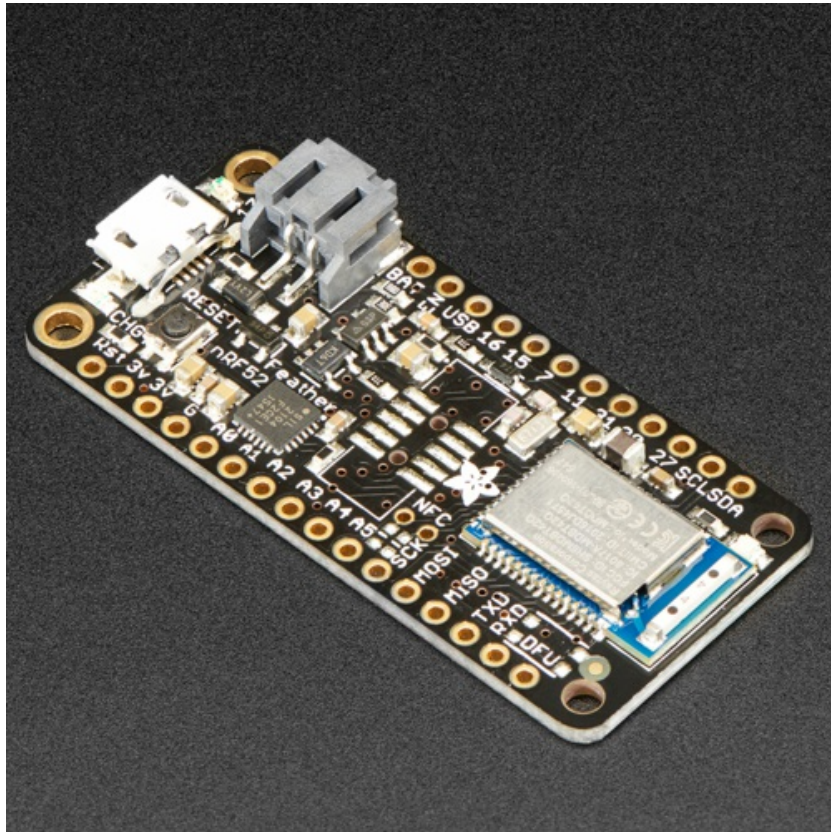


□

Bluefruit nRF52 Feather Learning Guide

Created by Kevin Townsend



Last updated on 2017-04-27 07:29:12 PM UTC

Guide Contents

Guide Contents	2
Introduction	6
nRF52832 Technical Details	6
nRF51 or nRF52 Bluefruit Devices?	6
Device Pinout	8
Special Notes	8
Power Pins	9
Analog Inputs	9
PWM Outputs	9
I2C Pins	9
Assembly	10
Header Options!	10
Soldering in Plain Headers	12
Prepare the header strip:	12
Add the breakout board:	12
And Solder!	13
Soldering on Female Header	14
Tape In Place	14
Flip & Tack Solder	15
And Solder!	15
Arduino BSP Setup	17
1. BSP Installation	17
Recommended: Installing the BSP via the Board Manager	17
2. Third Party Tool Installation	18
nrfutil (OS X and Linux Only)	18
3. Advanced Option: Manually Install the BSP via 'git'	18
Adafruit nRF52 BSP via git (for core development and PRs only)	18
BSP FAQs	19
Windows Related	19
OS X Related	19
Linux Related	19
Arduino Board Setup	20
1. Select the Board Target	20

2. Select the USB CDC Serial Port	20
3. Run a Test Sketch	20
Using the Bootloader	21
Forcing Serial Boot Mode	21
Factory Reset	21
Advanced: OTA DFU Bootloader	21
Flashing the Bootloader	23
Third Party Tool Requirements	23
JLink Drivers and Tools	23
Burning the Bootloader from the Arduino IDE	23
Manually Burning the Bootloader via nrfjprog	23
Examples	25
Example Source Code	25
Documented Examples	25
Advertising: Beacon	26
Complete Code	26
Output	26
BLE UART: Controller	28
Setup	28
Complete Code	28
Custom: HRM	33
HRM Service Definition	33
Implementing the HRM Service and Characteristics	33
Service + Characteristic Setup Code Analysis	34
Full Sample Code	35
Bluefruit nRF52 API	38
AdafruitBluefruit	39
API	39
Examples	39
BLEAdvertising	41
API	41
Related Information	41
Example	41

BLEService	43
Basic Usage	43
Order of Operations (Important!)	43
API	43
Example	43
BLECharacteristic	45
Basic Usage	45
Order of Operations (Important!)	45
API	45
Example	46
BLEDis	48
API	48
Example	48
Output	49
BLEUart	50
API	50
Example	50
BLEBeacon	52
API	52
Example	52
Testing	53
BLEMidi	54
API	54
Installing the Arduino MIDI Library	54
Example	54
Usage	56
BLEHidAdafruit	58
API	58
Example Sketches	58
Bonding HID Devices	58
Setting up your Bluefruit device for bonding	59
Bonding on iOS	59
Testing the HID Keyboard and Bonding	60
Memory Map	61

Flash Memory	61
SRAM Layout	61
Software Resources	62
Bluefruit LE Client Apps and Libraries	62
Bluefruit LE Connect (http://adafru.it/f4G) (Android/Java)	62
Bluefruit LE Connect (http://adafru.it/f4H) (iOS/Swift)	62
Bluefruit LE Connect for OS X (http://adafru.it/o9F) (Swift)	62
Bluefruit LE Command Line Updater for OS X (http://adafru.it/pLF) (Swift)	63
Deprecated: Bluefruit Buddy (http://adafru.it/mCn) (OS X)	63
ABLE (http://adafru.it/ijB) (Cross Platform/Node+Electron)	64
Bluefruit LE Python Wrapper (http://adafru.it/fQF)	64
Debug Tools	65
AdaLink (http://adafru.it/fPq) (Python)	65
Adafruit nRF51822 Flasher (http://adafru.it/fVL) (Python)	65
Downloads	66
Module Details	66
Schematic	66
FAQs	67

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 ATSAM21. Instead, this chip is programmed directly! It's got 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-programmed 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.

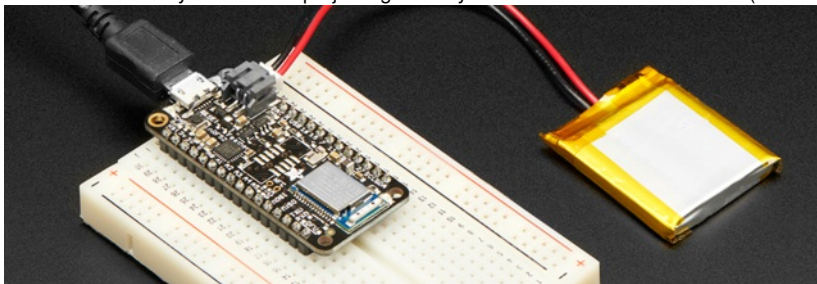
And to get you up and running quickly, we've done all the heavy lifting of getting the low level BLE stack into shape so that you can focus on your project from day one!

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 \(http://adafru.it/vaJ\)](http://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!](http://adafru.it/vby) (<http://adafru.it/vby>) (Wings that require the UART like the GPS FeatherWing won't work)

Further technical details available in the [nRF52832 \(http://adafru.it/vaJ\)](http://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 \(http://adafru.it/vaJ\)](http://adafru.it/vaJ) SoC) is quite different from the earlier nRF51822 based Bluefruit products ([Bluefruit M0 Feather \(http://adafru.it/t6a\)](http://adafru.it/t6a), 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:

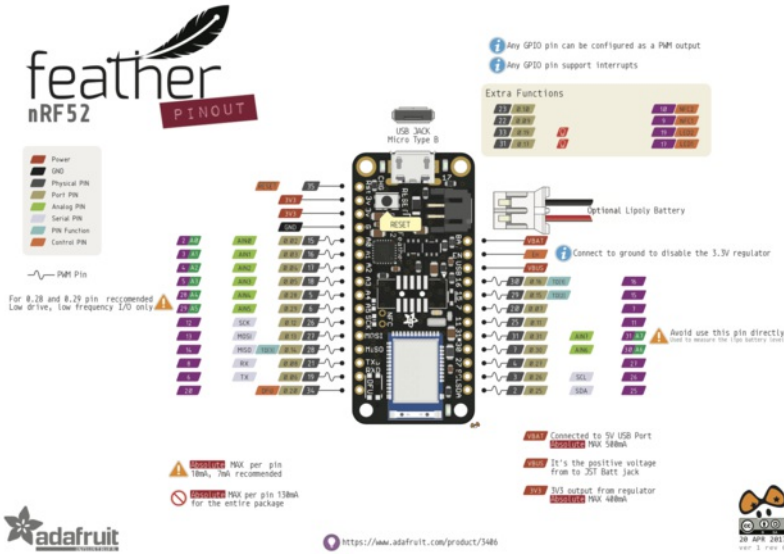
- nRF51 based Bluefruit boards run as modules that you connect to via an external MCU (typically an Atmel 32u4 or a SAMD21), sending AT style commands over SPI or UART.
- **With the nRF52, you run all of your code directly on the nRF52832 and no external MCU is used or required!**

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.

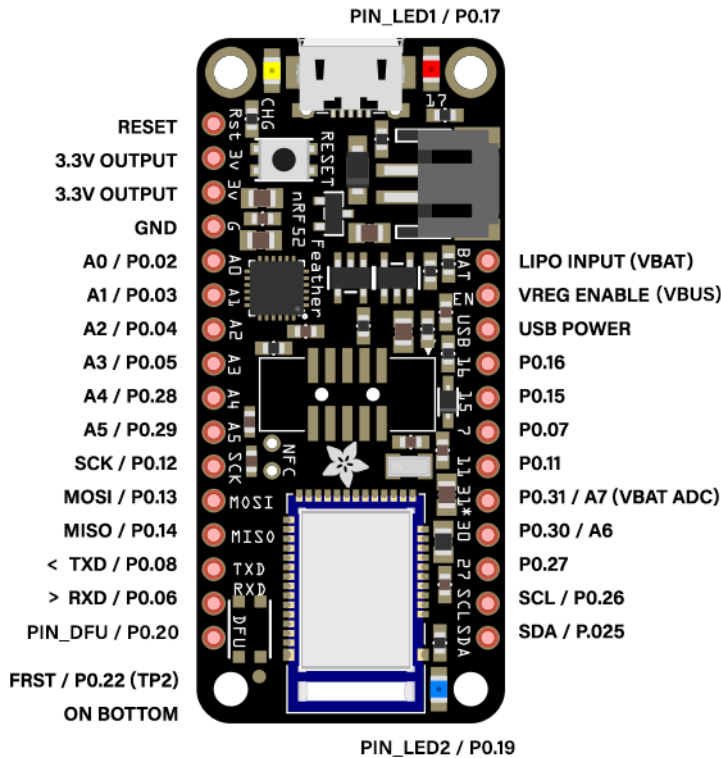
As a tradeoff, it also means a completely different API and development process, though!

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



BLUEFRUIT NRF52 FEATHER PINOUT



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 directly.**
- **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.

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..4095)

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 nRF52 BSP, and further improvements are planned here.

I2C Pins

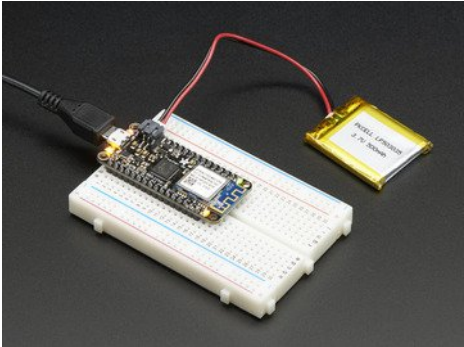
I2C pins on the nRF52832 require external pullup resistors to function, which are not present on the Adafruit nRF52 Feather by default. You will need to supply external pullups to use these. All Adafruit I2C breakouts have appropriate pullups on them already, so this normally won't be an issue for you.

Assembly

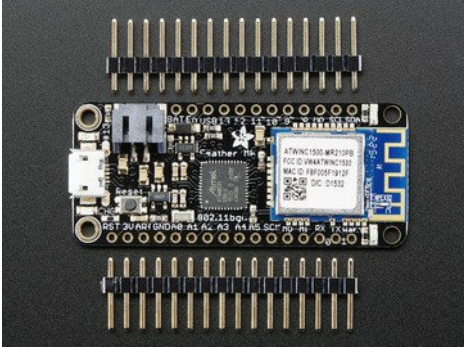
We ship Feathers fully tested but without headers attached - this gives you the most flexibility on choosing how to use and configure your Feather

Header Options!

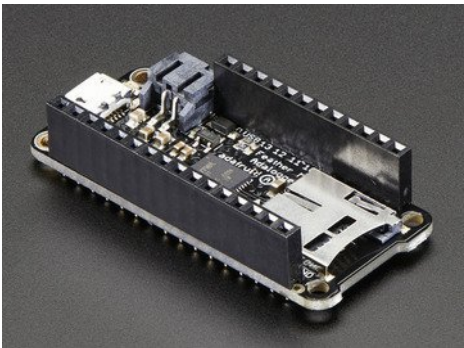
Before you go gung-ho on soldering, there's a few options to consider!



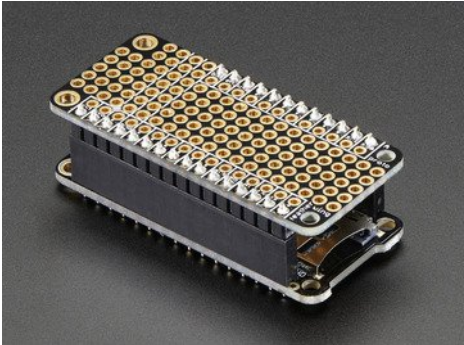
- The first option is soldering in plain male headers, this lets you plug in the Feather into a solderless breadboard



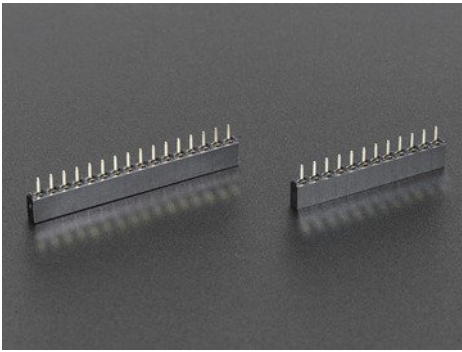
-



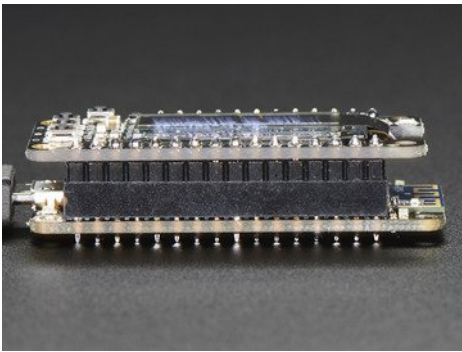
- Another option is to go with socket female headers. This won't let you plug the Feather into a breadboard but it will let you attach featherwings very easily



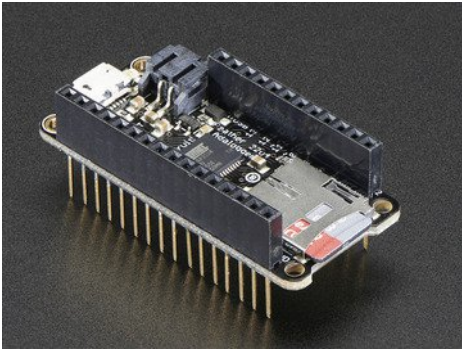
-



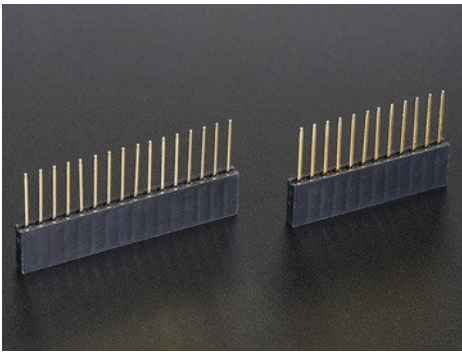
- We also have 'slim' versions of the female headers, that are a little shorter and give a more compact shape



-

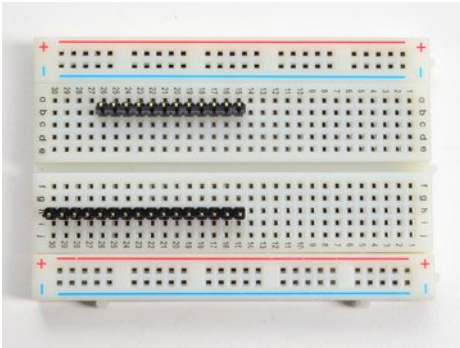


-



- 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

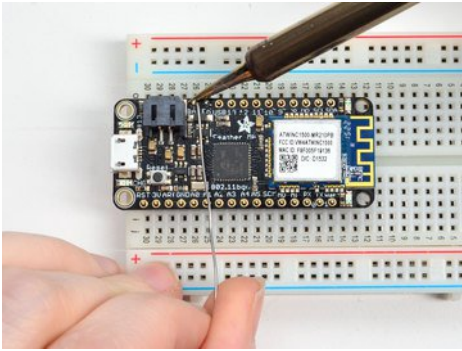


Prepare the header strip:

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

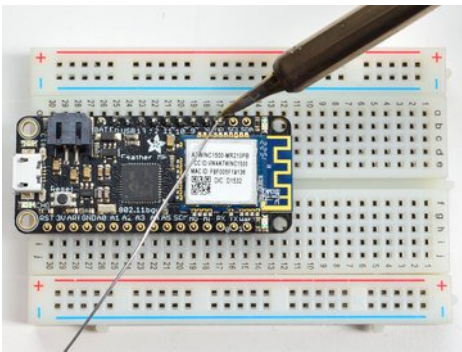
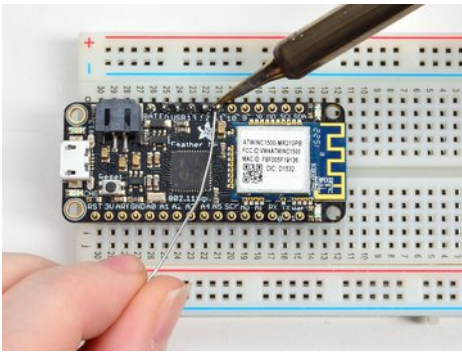


pads

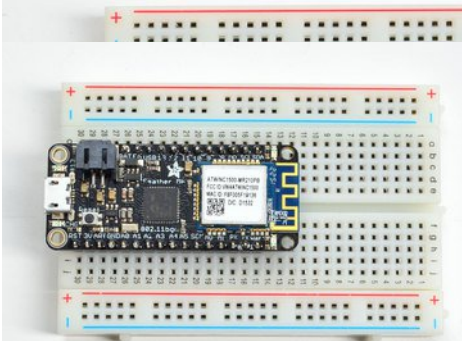
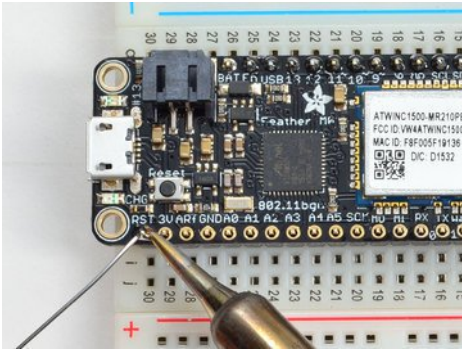
And Solder!

Be sure to solder all pins for reliable electrical contact.

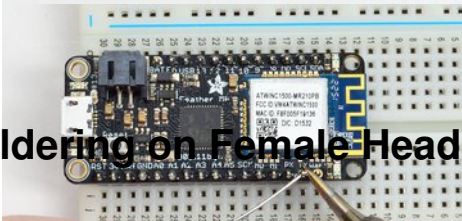
(For tips on soldering, be sure to check out our [Guide to Excellent Soldering](http://adafru.it/aTk) (<http://adafru.it/aTk>)).



Solder the other strip as well.



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

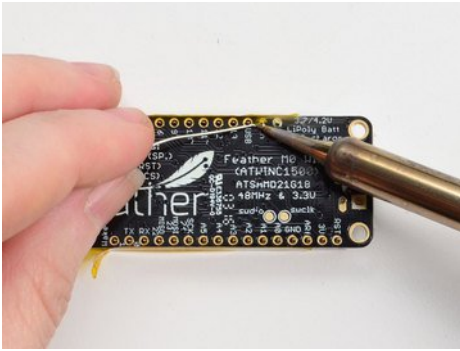


Soldering on Female Header



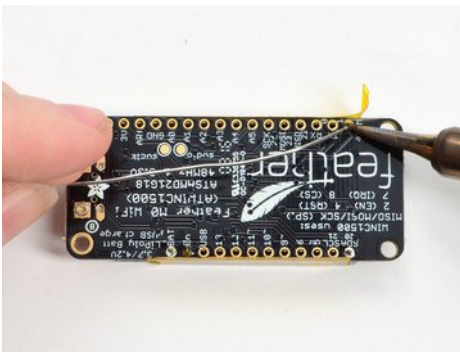
Tape In Place

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

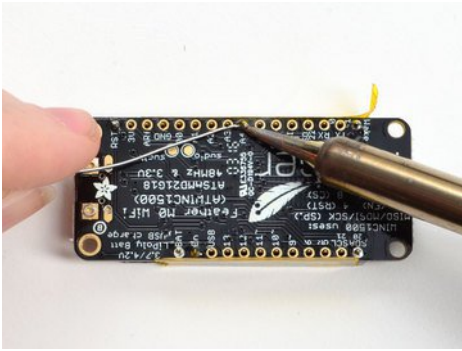
After flipping over, solder one or two points on each strip, to 'tack' the header in place



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](http://adafru.it/aTk) (<http://adafru.it/aTk>)).



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



Arduino BSP Setup

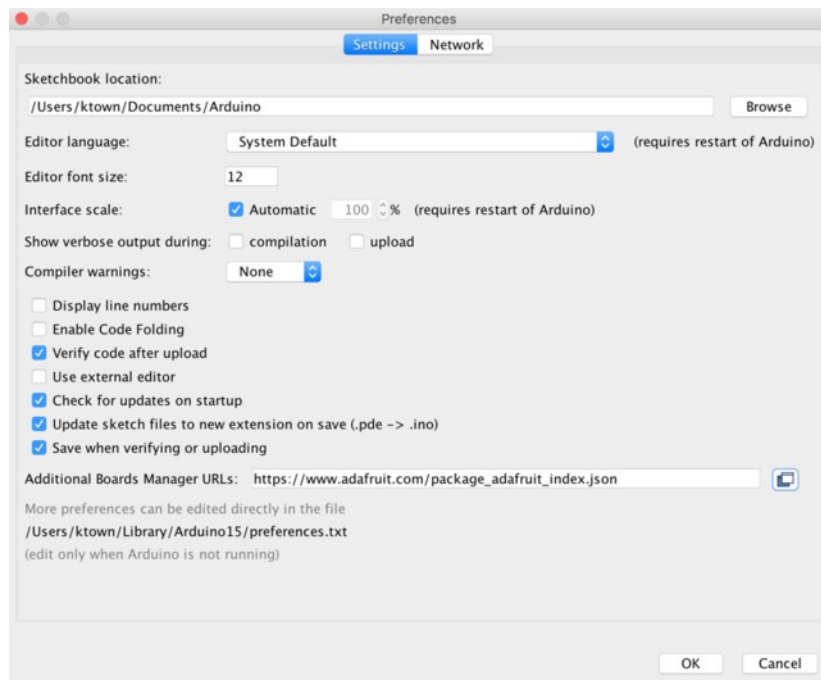
You can install the Adafruit Bluefruit nRF52 BSP in two steps:

nRF52 support requires at least Arduino IDE version 1.6.12! 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!

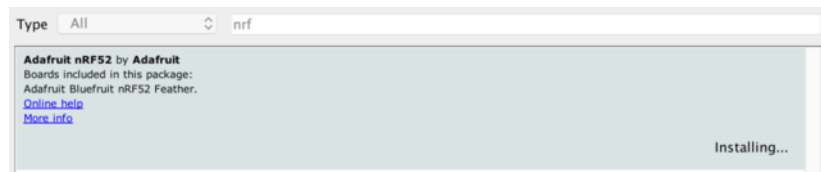
1. BSP Installation

Recommended: Installing the BSP via the Board Manager

- [Download and install the Arduino IDE \(http://adafru.it/fvm\)](http://adafru.it/fvm) (At least v1.6.12)
- Start the Arduino IDE
- Go into Preferences
- Add https://www.adafruit.com/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)

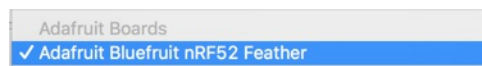


It will take up to a few minutes to finish installing the cross-compiling toolchain and tools associated with this BSP.

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



2. Third Party Tool Installation

The following third party tools must also be installed on your system to allow you to work with the Bluefruit nRF52 Feather:

This step is only required on OS X and Linux. If you are using Windows, a pre-built 32-bit binary of nrfutil is already included in the BSP that should work out of the box for most setups.

You will need to have both Python and pip available on your system to use the tools below!

nrfutil (OS X and Linux Only)

This is a [python wrapper](http://adafru.it/vaG) (<http://adafru.it/vaG>) for Nordic's **nrfutil**, which is used to flash boards using the built in serial bootloader.

To install this tool, open a terminal or command prompt window and go into the folder where the BSP was installed in step one above.

Depending on your operating system. The BSP should be located in one of the following paths:

- **Windows:** %APPDATA%\Local\Arduino15\packages\adafruit\hardware\nrf52
- **OS X:** ~/Library/Arduino15/packages/adafruit/hardware/nrf52
- **Linux:** ~/.arduino15/packages/adafruit/hardware/nrf52

The path above will also include a sub-folder with the version number (ex. `0.5.0`). Be sure to enter that sub-folder as well, for example:

```
$ cd ~/Library/Arduino15/packages/adafruit/hardware/nrf52
$ ls
0.5.0/
$ cd 0.5.0
```

Next go into the `tools/nrfutil-0.5.2` folder in the path above, and run the following commands to make nrfutil available to the Arduino IDE:

```
$ cd tools/nrfutil-0.5.2
$ sudo pip install -r requirements.txt
$ sudo python setup.py install
```

Don't install nrfutil from the pip package (ex. `sudo pip install nrfutil`). The latest nrfutil does not support DFU via Serial, and you should install the local copy of 0.5.2 included with the BSP via the `python setup.py install` command above.

3. 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 described below:

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

1. Go to the sketchbook folder on your command line, which should be one of the following:
OS X: ~/Documents/Arduino
Linux: ~/Arduino
Windows: ~/Documents/Arduino
2. Create a folder named `hardware/Adafruit`, if it does not exist, and change directories into it.
3. Clone the [Adafruit_nRF52_Arduino](http://adafru.it/vaF) (<http://adafru.it/vaF>) repo in the folder described in step 2:
`git clone git@github.com:adafruit/Adafruit_nRF52_Arduino.git`
4. This should result in a final folder name like `~/Documents/Arduino/hardware/Adafruit/Adafruit_nRF52_Arduino` (OS X).
5. Restart the Arduino IDE

You may also need to manually install the GCC ARM toolchain if it isn't already present:

1. Download the cross compiler: https://github.com/adafruit/Adafruit_nRF52_Arduino/releases/tag/gcc-5_2-2015q4 (<http://adafru.it/vuB>)
2. Extract the file to %APPDATA%\Local\Arduino15\packages\adafruit\tools\gcc-arm-none-eabi
3. Rename the extracted folder from `gcc-arm-none-eabi-5_2-2015q4` to `5_2-2015q4`

You should end up with the following folder structure for the cross-compiler:

Name	Date modified	Type	Size
arm-none-eabi	4/7/2017 10:18 PM	File folder	
bin	4/7/2017 10:19 PM	File folder	
lib	4/7/2017 10:19 PM	File folder	
share	4/7/2017 10:20 PM	File folder	

BSP FAQs

The following FAQs may be useful if you run into any problems:

Windows Related

If you are using BSP 0.6.0 or greater, there are no known issues with the BSP installation process on Windows. Please update to the latest version if you currently have an earlier release.

OS X Related

I can compile and link sketches on OS X, but nrfutil gives me the following error: 'AttributeError: 'int' object has no attribute 'value''?

Depending on your system setup and Python version, you may need to make a manual adjustment to a file in nrfutil, which is used when compiling and flashing files from the Arduino IDE.

Open the following file (please note that the BSP version number in the path may be different!):

~/Library/Arduino15/packages/adafruit/hardware/nrf52/0.5.1/tools/nrfutil-0.5.2/nordicsemi/dfu/init_packet.py ... and make the following changes:

```
@@ -79,7 +79,7 @@
-     for key in sorted(self.init_packet_fields.keys(), key=lambda x: x.value):
+     for key in sorted(self.init_packet_fields.keys(), key=lambda x: x):
```

```
@@ -94,7 +94,8 @@
-     for key in sorted(self.init_packet_fields.keys(), key=lambda x: x.value):
+     for key in sorted(self.init_packet_fields.keys(), key=lambda x: x):
```

Linux Related

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 \(http://adafru.it/vnE\)](http://adafru.it/vnE)). Try running the following commands from the command line to resolve this:

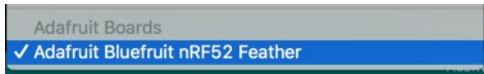
```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386
```

Arduino Board Setup

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**



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 SiLabs device



If you don't see the SiLabs device listed, you may need to install the [SiLabs CP2104 driver](http://adafru.it/vaH) (<http://adafru.it/vaH>) on your system.

3. 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:

```
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 pin #17 red LED on the Feather

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.

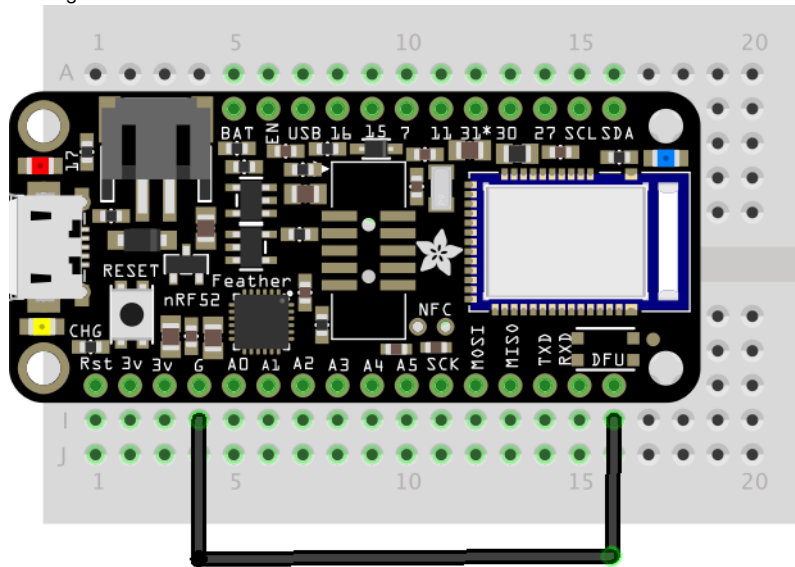
Forcing Serial Boot Mode

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.



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 supported by both Nordic mobile apps, and our 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 supported 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!

Flashing the Bootloader

All Adafruit nRF52 boards ship 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 rare cases where you may want or need to reflash the bootloader yourself, and have access to the HW required to do so.

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

Third Party Tool Requirements

To burn the bootloader from within the Arduino IDE, you will need the following tools installed on your system and available in the system path:

JLink Drivers and Tools

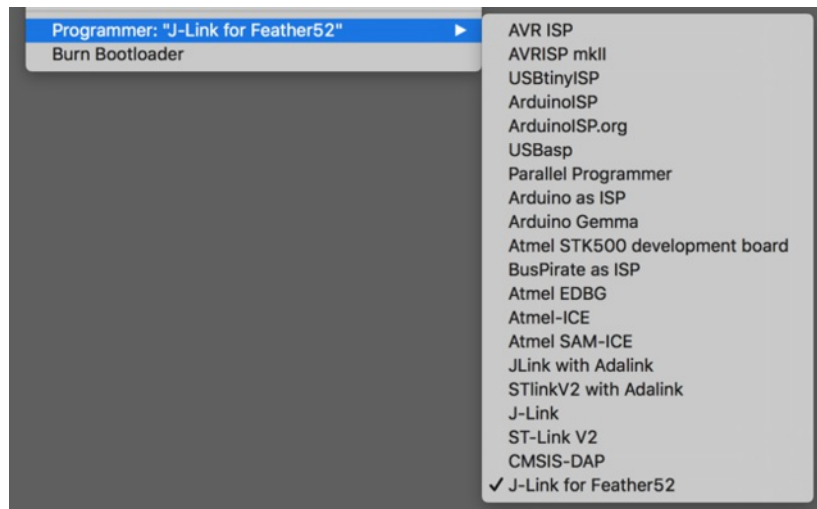
Download and install the [JLink Software and Documentation Pack \(http://adafru.it/val\)](http://adafru.it/val) from Segger, which will also install a set of command line tools.

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:



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 \(http://adafru.it/vaJ\)](http://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 following commands, updating the path to the .hex file as appropriate:

```
$ nrfjprog -e -f nrf52
$ nrfjprog --program bootloader_with_s132.hex -f nrf52
$ nrfjprog --reset -f nrf52
```

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.
Programming device.
```

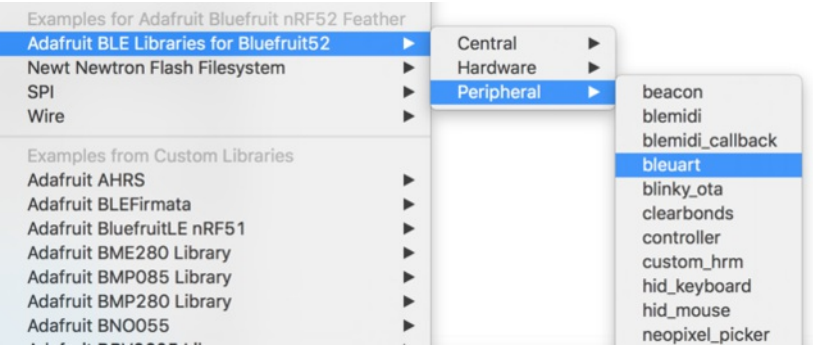
```
$ ./nrfjprog --reset -f nrf52
Applying system reset.
Run.
```

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
```


Examples

There are numerous examples available for the Bluefruit nRF52 Feather 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](http://adafru.it/vaK)
<http://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 Bluefruit 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.

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

The code below may be out of sync with the latest examples on Github. You should always consult Github for the latest version.

The latest version of this code is available on [Github \(http://adafru.it/vaM\)](http://adafru.it/vaM) and in the Examples menu.

```
#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 (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 optional secondary Scan Response packet for 'Name' instead
  Bluefruit.ScanResponse.addName();
}

void loop()
{
  // Toggle both LEDs every second
  digitalToggle(LED_BUILTIN);
  delay(1000);
}
```

Output

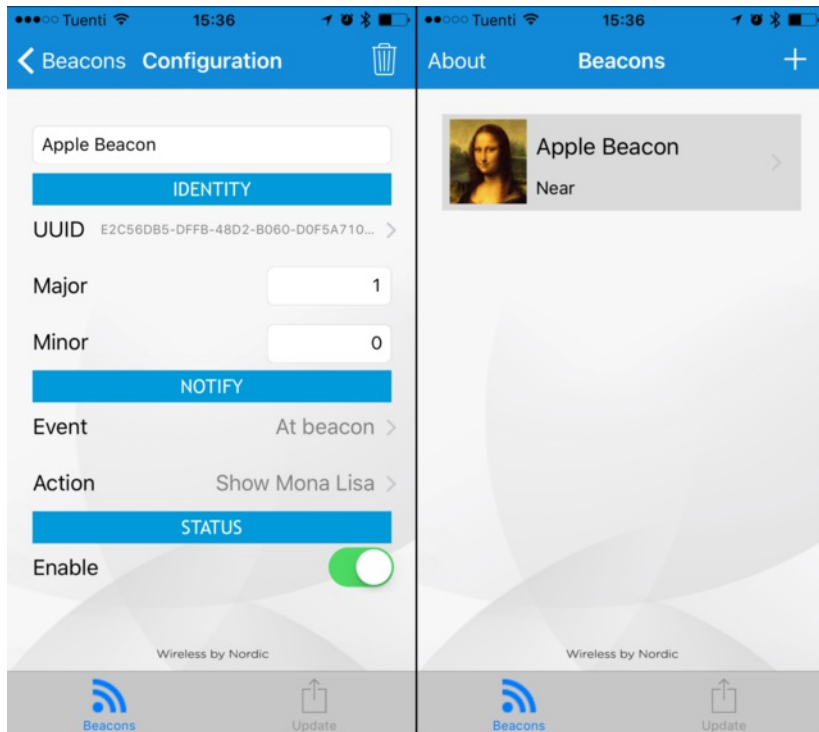
You can use the nRF Beacons application from Nordic Semiconductors to test this sketch:

- [nRF Beacons for iOS \(http://adafru.it/vaC\)](http://adafru.it/vaC)

- [nRF Beacons for Android](http://adafru.it/vaD) (<http://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 out free [OS \(http://adafru.it/f4H\)](http://adafru.it/f4H), [Android \(http://adafru.it/f4G\)](http://adafru.it/f4G) or [OS X \(http://adafru.it/o9F\)](http://adafru.it/o9F) applications.

- Load the [Controller example sketch \(http://adafru.it/vaN\)](http://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:

```

Adafruit Bluefruit52 Controller App Example
-----
RGB #FF504E

```

Complete Code

The latest version of this code is always available on [Github \(http://adafru.it/vaN\)](http://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.

```
#include <bluefruit.h>

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

  Bluefruit.begin();
  Bluefruit.setName("Bluefruit52");

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

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

  // There is no room for 'Name' in the Advertising packet
  // Use the optional secondary Scan Response packet for 'Name' instead
  Bluefruit.ScanResponse.addName();
}

/*****
 *!
 * @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-->0) {
        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](http://adafru.it/vaO) (<http://adafru.it/vaO>) service, as defined by the Bluetooth SIG.

HRM Service Definition

UUID: [0x180D](http://adafru.it/vaO) (<http://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    <-- 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.write(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(BLECharacteristic& chr, uint16_t cccd_value)
{
    // Display the raw request packet
    Serial.print("CCCD Updated: ");
    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");
  }
}
}
}

```

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, but this maybe be out of sync with the latest code available on Github. Please consult the [Github code \(http://adafruit.it/vaP\)](http://adafruit.it/vaP) if you have any problems with the code below.

```

#include <bluefruit.h>

#define STATUS_LED (17)
#define BLINKY_MS (2000)

/* 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 bscl = BLECharacteristic(UUID16_CHR_BODY_SENSOR_LOCATION);

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

uint32_t blinkyms;
uint8_t bps = 0;

// Advanced function prototypes
void setupAdv(void);
void setupHRM(void);
void connect_callback(void);
void disconnect_callback(uint8_t reason);
void cccd_callback(BLECharacteristic& chr, ble_gatts_evt_write_t* request);

void setup()
{
  Serial.begin(115200);
  Serial.println("Bluefruit52 HRM Example");
  Serial.println("-----");

  // Setup LED pins and reset blinky counter
  pinMode(STATUS_LED, OUTPUT);
  blinkyms = millis();

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

  // Set the advertised device name (keep it short!)
  Serial.println("Setting Device Name to 'Feather52 HRM'");
  Bluefruit.setName("Feather52 HRM");

  // Set the connect/disconnect callback handlers
  Bluefruit.setConnectCallback(connect_callback);
  Bluefruit.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.update(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)");
setupAdv();

// Start Advertising
Serial.println("Ready Player One!!!");
Serial.println("\nAdvertising");
Bluefruit.Advertising.start();
}

void setupAdv(void)
{
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);
  Bluefruit.Advertising.addTxPower();

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

  // 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 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
  // 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.write(2); // Set the characteristic to 'Wrist' (2)
}

void connect_callback(void)
{
  Serial.println("Connected");
}

void disconnect_callback(uint8_t reason)
{
  (void) reason;

  Serial.println("Disconnected");
  Serial.println("Advertising!");
}

void cccd_callback(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()) {
      Serial.println("Heart Rate Measurement 'Notify' enabled");
    } else {
      Serial.println("Heart Rate Measurement 'Notify' disabled");
    }
  }
}

void loop()
{
  // Blinky!
  if (blinkyms+BLINKY_MS < millis()) {
    blinkyms = millis();
    digitalToggle(STATUS_LED);

    if (Bluefruit.connected()) {
      uint8_t hrmdata[2] = { 0b00000110, bps++ }; // Sensor connected, increment BPS value
      err_t resp = hrmc.notify(hrmdata, sizeof(hrmdata)); // Note: We use .notify instead of .write!

      // This isn't strictly necessary, but you can check the result
      // of the .notify() attempt to see if it was successful or not
      switch (resp) {
        case ERROR_NONE:
          // Value was written correctly!
          Serial.print("Heart Rate Measurement updated to: "); Serial.println(bps);
          break;
        case NRF_ERROR_INVALID_PARAM:
          // Characteristic property not set to 'Notify'
          Serial.println("ERROR: Characteristic 'Property' not set to Notify!");
          break;
        case NRF_ERROR_INVALID_STATE:
          // Notify bit not set in the CCCD or not connected
          Serial.println("ERROR: Notify not set in the CCCD or not connected!");
          break;
        default:
          // Unhandled error code
          Serial.print("ERROR: 0x"); Serial.println(resp, HEX);
          break;
      }
    }
  }
}

```

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

The Bluefruit nRF52 Feather codebase is in an early BETA stage and 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 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);

err_t begin(bool prph_enable = true, bool central_enable = false);

/*-----*/
/* Lower Level Classes (Bluefruit.Advertising.*, etc.)
 *-----*/
BLEAdvertising Advertising;
BLEAdvertising ScanResponse;
BLECentral Central;

/*-----*/
/* General Purpose Functions
 *-----*/
void autoConnLed (bool enabled);
void setConnLedInterval (uint32_t ms);
void startConnLed (void);
void stopConnLed (void);
void setName (const char* str);
char* getName (void);
bool setTxPower (int8_t power);
int8_t getTxPower (void);

/*-----*/
/* GAP, Connections and Bonding
 *-----*/
bool connected (void);
void disconnect (void);
err_t setConnInterval (uint16_t min, uint16_t max);
err_t setConnIntervalMS (uint16_t min_ms, uint16_t max_ms);
uint16_t connHandle (void);
bool connBonded (void);
uint16_t connInterval (void);
void clearBonds (void);

ble_gap_addr_t peerAddr(void);

bool txbuf_get(uint32_t ms);

/*-----*/
/* Callbacks
 *-----*/
typedef void (*connect_callback_t) (void);
typedef void (*disconnect_callback_t) (uint8_t reason);

void setConnectCallback (connect_callback_t fp);
void setDisconnectCallback( 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](#)
<http://adafru.it/vaK>

BLEAdvertising

The Bluefruit nRF52 Feather codebase is in an early BETA stage and 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 formatted 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 BLEAdvertising class 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 BLEAdvertising class has the following public API:

```
// Constructor
BLEAdvertising(void);

bool start (uint8_t mode = 0);
bool stop (void);

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 addUuid (uint16_t uuid16);
bool addUuid (uint8_t const uuid128[]);
bool addService (BLEService& service);

bool setBeacon (BLEBeacon& beacon);

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

Related Information

- [Generic Access Profile \(http://adafruit.it/val\)](http://adafruit.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 setupAdv(void)  
{  
  // Advertise as BLE only and general discoverable  
  Bluefruit.Advertising.addFlags(BLE_GAP_ADV_FLAGS_LE_ONLY_GENERAL_DISC_MODE);  
  
  // Add the TX Power to the advertising packet  
  Bluefruit.Advertising.addTxPower();  
  
  // Include bleuart 128-bit uuid in the advertising packet  
  Bluefruit.Advertising.addService(bleuart);  
  
  // There is no room for Name in Advertising packet  
  // Use Scan response to store it instead  
  Bluefruit.ScanResponse.addName();  
}
```

BLEService

The Bluefruit nRF52 Feather codebase is in an early BETA stage and 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 bscl = 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 <-- 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] = { 0b0000110, 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
  bscl.setProperties(CHR_PROPS_READ);
  bscl.setPermission(SECMODE_OPEN, SECMODE_NO_ACCESS);
  bscl.setFixedLen(1);
  bscl.begin();
  bscl.write(2); // Set the characteristic to 'Wrist' (2)
}

void cccd_callback(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()) {
      Serial.println("Heart Rate Measurement 'Notify' enabled");
    } else {
      Serial.println("Heart Rate Measurement 'Notify' disabled");
    }
  }
}

```

BLECharacteristic

The Bluefruit nRF52 Feather codebase is in an early BETA stage and 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 throughout 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& chr, ble_gatts_evt_read_t * request);
```

```

typedef void (*write_authorize_cb_t) (BLECharacteristic& chr, ble_gatts_evt_write_t* request);
typedef void (*write_cb_t) (BLECharacteristic& chr, uint8_t* data, uint16_t len, uint16_t offset);
typedef void (*write_cccd_cb_t) (BLECharacteristic& chr, uint16_t value);
typedef void (*chars_cb_t) (void);

BLEUuid uuid;

// Constructors
BLECharacteristic(void);
BLECharacteristic(uint16_t uuid16);
BLECharacteristic(uint8_t const uuid128[]);

BLEService& parentService()
{
    return *_service;
}

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

// Configure
void setTempMemory(void);

void setProperties(uint8_t prop);
void setPermission(BleSecurityMode read_perm, BleSecurityMode write_perm);

void setStringDescriptor(const char* descriptor); // aka user descriptor
void setReportRefDescriptor(uint8_t id, uint8_t type);

void setMaxLen(uint16_t max_len);
void setFixedLen(uint16_t fixed_len);

// 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);

err_t begin(void);

ble_gatts_char_handles_t handles(void);

// Write
err_t write(const void* data, int len, uint16_t offset);
err_t write(const void* data, int len);
err_t write(const char * str);

err_t write(int num);
err_t write(uint32_t num);
err_t write(uint16_t num);
err_t write(uint8_t num);

// Notify
bool notifyEnabled(void);

err_t notify(const void* data, int len, uint16_t offset);
err_t notify(const void* data, int len);
err_t notify(const char * str);

err_t notify(int num);
err_t notify(uint32_t num);
err_t notify(uint16_t num);
err_t notify(uint8_t 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  <-- 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.write(2); // Set the characteristic to 'Wrist' (2)
}

void cccd_callback(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()) {
      Serial.println("Heart Rate Measurement 'Notify' enabled");
    } else {
      Serial.println("Heart Rate Measurement 'Notify' disabled");
    }
  }
}
}

```

BLEDis

The Bluefruit nRF52 Feather codebase is in an early BETA stage and 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](http://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](http://adafru.it/vav) (0x2A24), exposed via `.setModel(const char*)`
- [Serial Number String](http://adafru.it/vaw) (0x2A25), private
- [Firmware Revision String](http://adafru.it/vax) (0x2A26), private
- [Hardware Revision String](http://adafru.it/vay) (0x2A27), exposed via `.setHardwareRev(const char*)`
- [Software Revision String](http://adafru.it/vaz) (0x2A28), exposed via `.setSoftwareRev(const char*)`
- [Manufacturer Name String](http://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:

- **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 <bluefruit.h>  
  
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 Feather codebase is in an early BETA stage and 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 <bluefruit.h>

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 < 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");
}

```

BLEBeacon

The Bluefruit nRF52 Feather codebase is in an early BETA stage and 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 \(http://adafru.it/vaB\)](http://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 manufacturer);
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& 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& beacon);
```

See the example below for a concrete usage example.

Example

The following example will configure the nRF52 to advertise a 'Beacon' payload:

```
#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 (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 optional secondary Scan Response packet for 'Name' instead
Bluefruit.ScanResponse.addName();
}

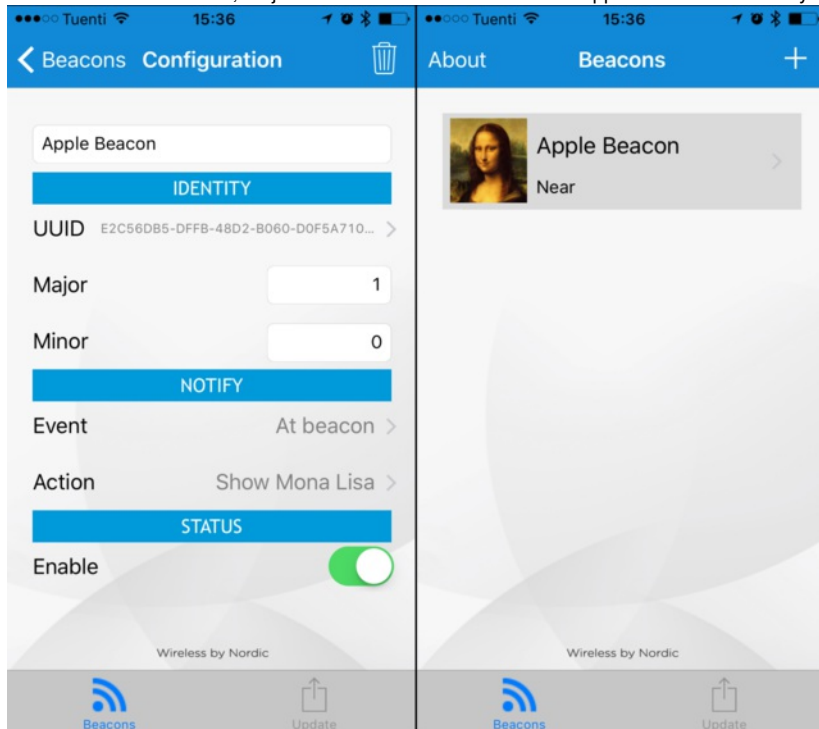
void loop()
{
// Toggle both LEDs every second
digitalToggle(LED_BUILTIN);
delay(1000);
}

```

Testing

If you test with the nRF Beacons application ([iOS \(http://adafru.it/vaC\)](http://adafru.it/vaC) or [Android \(http://adafru.it/vaD\)](http://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 Feather codebase is in an early BETA stage and 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);

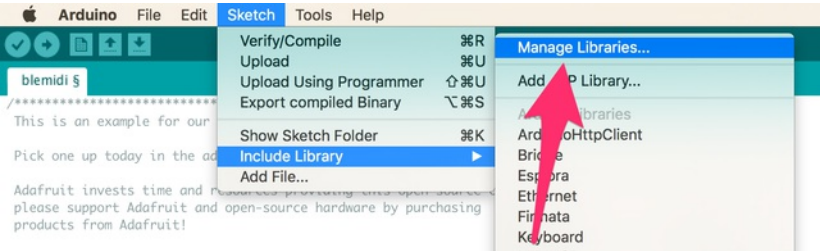
err_t begin (void);
bool notifyEnabled (void);

// 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);

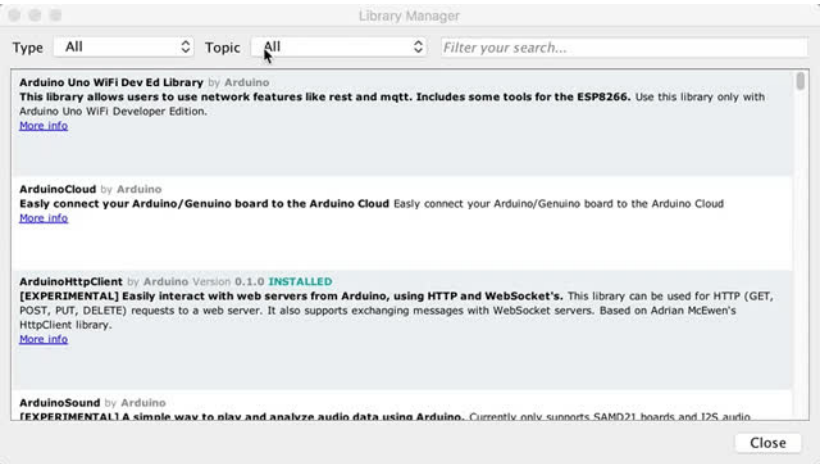
size_t write (const char *str);
size_t write (const uint8_t *buffer, size_t size);
```

Installing the Arduino MIDI Library

BLEMidi is easiest to use when combined with the [Arduino MIDI Library](#). You will need **version 4.3.0 or higher** installed before continuing with the example code.

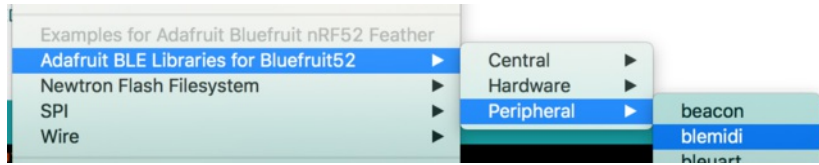


Next, select **Communication** from the topic dropdown, and enter **MIDI Library** into the search box. Click the **Install** button to install version 4.3.0 or higher of the **MIDI Library**.



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 <bluefruit.h>
#include <MIDI.h>

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 ble midi 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 < 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 >= 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 Feather codebase is in an early BETA stage and 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 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 interval=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 \(http://adafru.it/vb8\)](http://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 \(http://adafru.it/vb9\)](http://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 **asbonding**, and allows bonded devices to securely communicate without user interaction in the future.

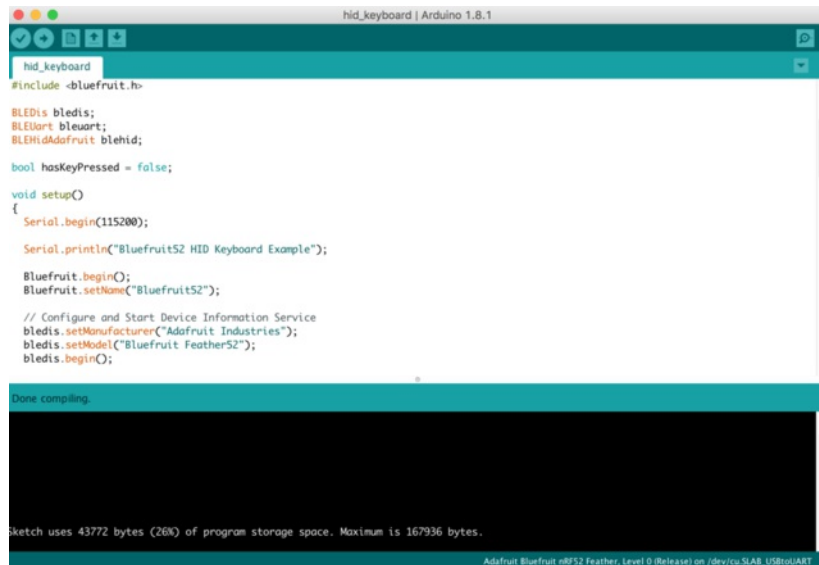
- To cancel the bonding agreement, you can simply delete the keys on the nRF52 via the [clearbonds](http://adafru.it/vba) sketch, or delete the bonding data on your mobile device or computer.

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:



```
hid_keyboard | Arduino 1.8.1
hid_keyboard
#include <Bluefruit.h>

BLEDS bledis;
BLEUART bleuart;
BLEHIDAdafruit blehid;

bool hasKeyPressed = false;

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

  Serial.println("Bluefruit52 HID Keyboard Example");

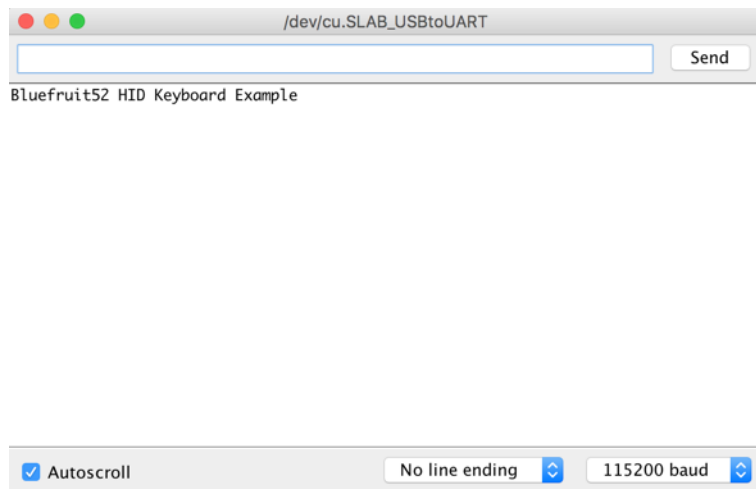
  Bluefruit.begin();
  Bluefruit.setName("Bluefruit52");

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

Done compiling.

Sketch uses 43772 bytes (26%) of program storage space. Maximum is 167936 bytes.
Adafruit Bluefruit nRF52 Feather, Level 0 (Release) on /dev/cu.SLAB_USBtoUART
```

Opening the Serial Monitor will give you the following output (though it may differ depending on the debug level you have selected):



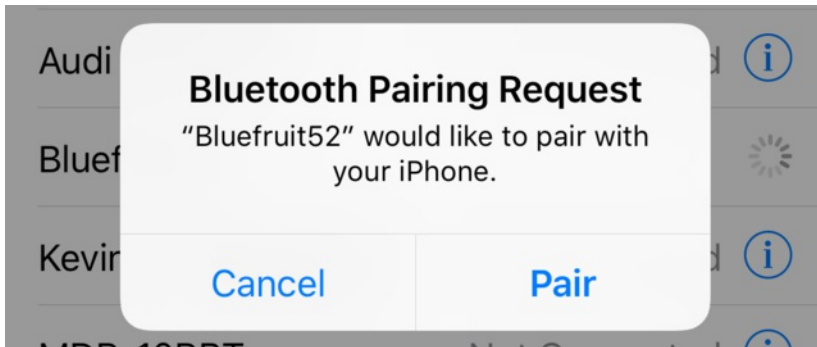
Bonding on iOS

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:



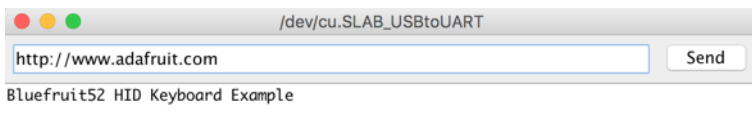
Click the **Pair** button, and the devices will be paired and bonded, and will automatically connect to each other in the future.

If everything went well, you will see the device in your **MY DEVICES** list, as follows:

Bluefruit52 Connected ⓘ

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.



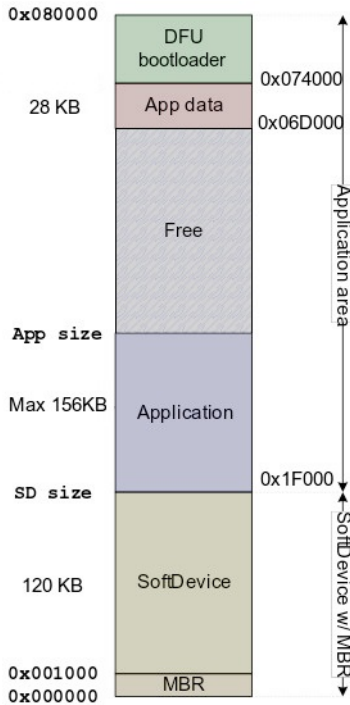
The text will then appear in your mobile app or bonded device.

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!

Memory Map

Flash Memory

The nRF52832 has 512KB flash memory with the following layout:



- **MBR:** Contains the Master Boot Record (MBR). This startup code checks the address for the bootloader, and then executes the appropriate bootloader code.
- **SoftDevice:** This section of flash memory contains the **S132** 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. The sketches are limited to 156KB of flash maximum.
- **Free:** This section of flash memory is kept empty to enable over the air (OTA) 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 OTA complete successfully and that the entire image is safely store on the device to avoid losing the application code.
- **App Data:** This 28KB section of flash memory is reserved for config settings. It uses an open source file system called the [Newtron Flash File System](http://adafru.it/vaQ) (<http://adafru.it/vaQ>), which is part of the OpenSource Mynewt operating system. Bonding data is stored here, 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:

- **Stack Memory:** 3KB, 0x2001000 ... 0x2000F400
- **Heap Memory:** 0x2000F400, counting downward

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.

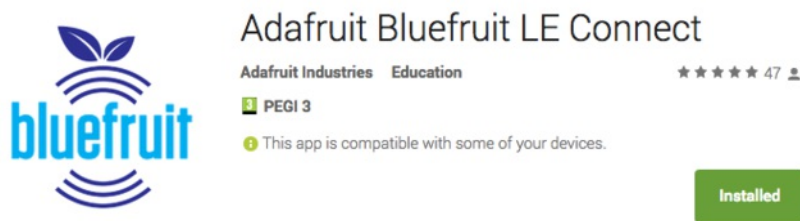
Bluefruit LE Client Apps and Libraries

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 \(http://adafru.it/f4G\)](http://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 \(http://adafru.it/f4G\)](http://adafru.it/f4G).

The full [source code \(http://adafru.it/fY9\)](http://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 \(http://adafru.it/fYa\)](http://adafru.it/fYa) to use this project.



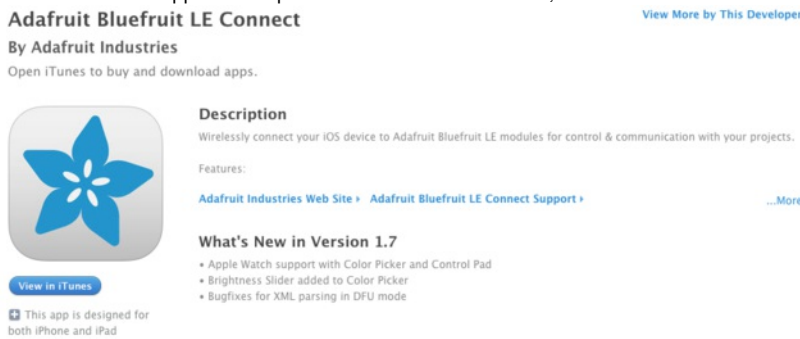
[Bluefruit LE Connect \(http://adafru.it/f4H\)](http://adafru.it/f4H) (iOS/Swift)

Apple was very early to adopt Bluetooth Low Energy, and we also have an iOS version of the [Bluefruit LE Connect \(http://adafru.it/f4H\)](http://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 developer program to use this project:

- Version 1.x source code: [https://github.com/adafruit/Bluefruit_LE_Connect \(http://adafru.it/ddv\)](https://github.com/adafruit/Bluefruit_LE_Connect)
- Version 2.x source code: [https://github.com/adafruit/Bluefruit_LE_Connect_v2 \(http://adafru.it/o9E\)](https://github.com/adafruit/Bluefruit_LE_Connect_v2)

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.

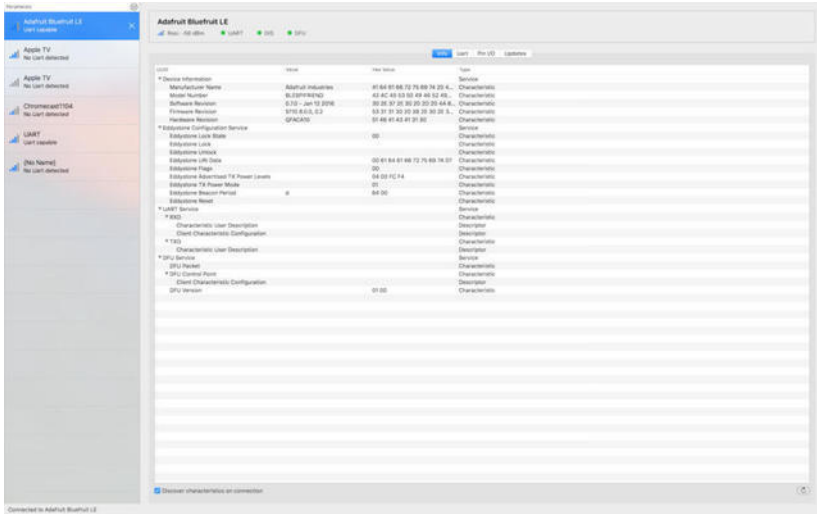


[Bluefruit LE Connect for OS X \(http://adafru.it/o9F\)](http://adafru.it/o9F) (Swift)

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 \(http://adafru.it/o9E\)](http://adafru.it/o9E).



[Bluefruit LE Command Line Updater for OS X \(http://adafru.it/pLF\)](http://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 \(http://adafru.it/pLF\)](http://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 <command> [options...]
```

```
Commands:
Scan peripherals: scan
Automatic update: update [--enable-beta] [--uuid <uuid>]
Custom firmware: dfu --hex <filename> [--init <filename>] [--uuid <uuid>]
Show this screen: --help
Show version: --version
```

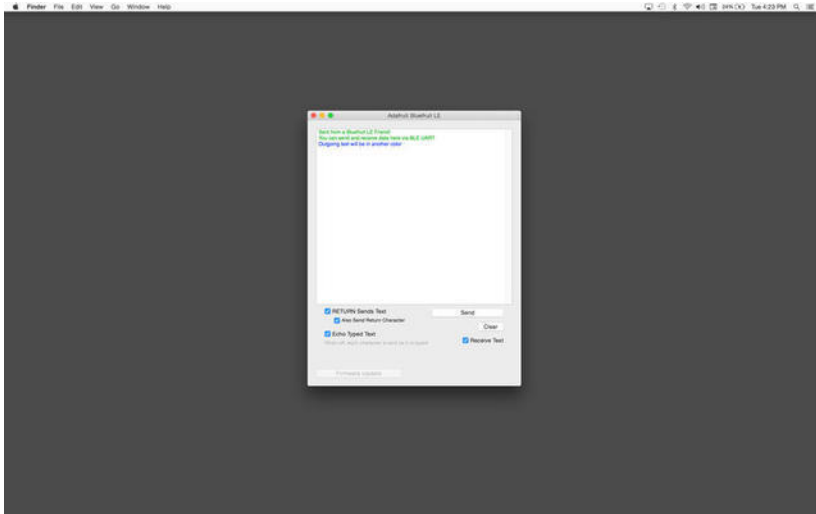
```
Options:
--uuid <uuid> If present the peripheral with that uuid is used. If not present a list of peripherals is displayed
--enable-beta If not present only stable versions are used
```

```
Short syntax:
-u = --uuid, -b = --enable-beta, -h = --hex, -i = --init, -v = --version, -? = --help
```

Deprecated: [Bluefruit Buddy \(http://adafru.it/mCn\)](http://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 \(http://adafru.it/mCo\)](http://adafru.it/mCo).



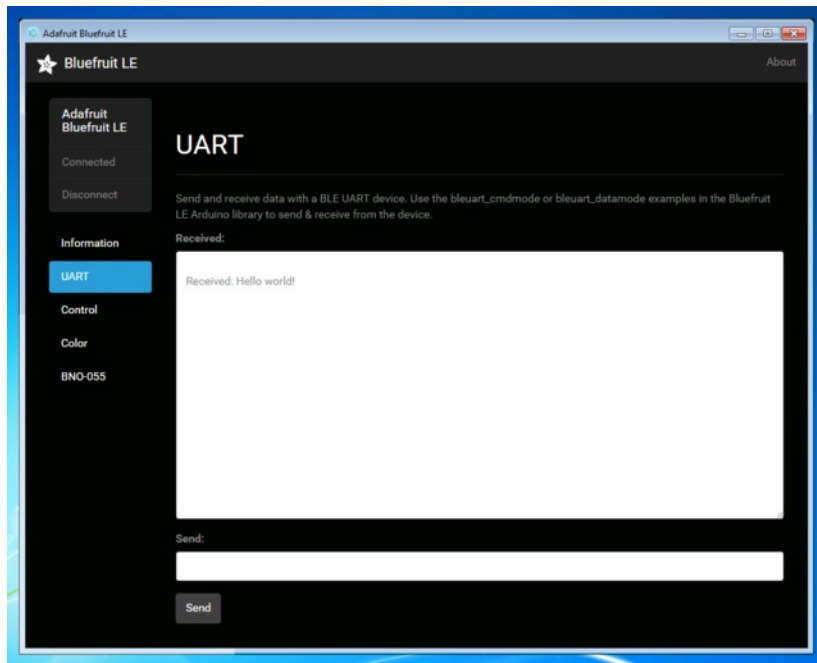
[ABLE \(http://adafru.it/ijB\)](http://adafru.it/ijB) (Cross Platform/Node+Electron)

[ABLE \(http://adafru.it/ijB\)](http://adafru.it/ijB) (Adafruit Bluefruit LE Desktop) is a cross-platform desktop application based on Sandeep Misty's [noble library \(http://adafru.it/ijC\)](http://adafru.it/ijC) and the [Electron \(http://adafru.it/ijD\)](http://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 \(http://adafru.it/1327\)](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 \(http://adafru.it/ijE\)](http://adafru.it/ijE) is available in addition to the easy to use [pre-compiled binaries \(http://adafru.it/ijB\)](http://adafru.it/ijB).



[Bluefruit LE Python Wrapper \(http://adafru.it/fQF\)](http://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 retrieve 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](http://adafru.it/fQF) (<http://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 (<http://adafru.it/fPq>) (Python)

This command line tool is a python-based wrapper for programming ARM MCUs using either a [Segger J-Link](http://adafru.it/fYU) (<http://adafru.it/fYU>) or an [STLink/V2](http://adafru.it/ijf) (<http://adafru.it/ijf>). You can use it to reflash your Bluefruit LE module using the latest firmware from the [Bluefruit LE firmware repo](http://adafru.it/edX) (<http://adafru.it/edX>).

Details on how to use the tool are available in the `readme.md` file on the main [Adafruit_Adalink](http://adafru.it/fPq) (<http://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 (<http://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](http://adafru.it/fVL) (<http://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_Firmv
...
```

Downloads

The following resources may be useful working with the Bluefruit nRF52 Feather:

- [Adafruit_nRF52_Arduino](http://adafru.it/vaF) (<http://adafru.it/vaF>): The core code for this device (hosted on Github)
- [nRF52 Example Sketches](http://adafru.it/vaK) (<http://adafru.it/vaK>): Browse the example code from the core repo on Github
- [nRF52832 Product Specification](http://adafru.it/vaR) (<http://adafru.it/vaR>): Key technical documentation for the nRF52832 SoC
- [EagleCAD PCB files on GitHub](http://adafru.it/vbH) (<http://adafru.it/vbH>)

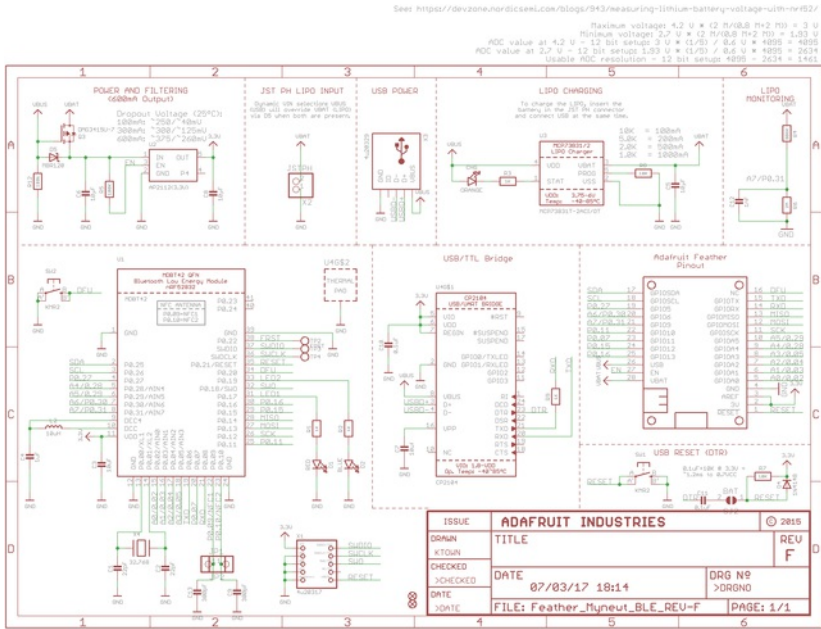
[Feather Bluefruit NRF52 Pinout Diagram](http://adafru.it/vWa)
<http://adafru.it/vWa>

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](http://adafru.it/vbb)
<http://adafru.it/vbb>

Schematic



FAQs

NOTE: For FAQs relating to the **BSP**, see the dedicated [BSP FAQ list \(http://adafru.it/vnF\)](http://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 nRF52832? Can I do hard real time processing (bit-banging NeoPixels, 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 nRF52832 to meet hard timing requirements when the radio is enabled and being actively used for Bluetooth Low Energy. This isn't possible on the nRF52832 even without FreeRTOS, though, since the SoftDevice (Nordic's proprietary binary blob stack) has higher priority than any user code, including control over interrupt handlers.

Can I use GDB to debug my nRF52832?

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