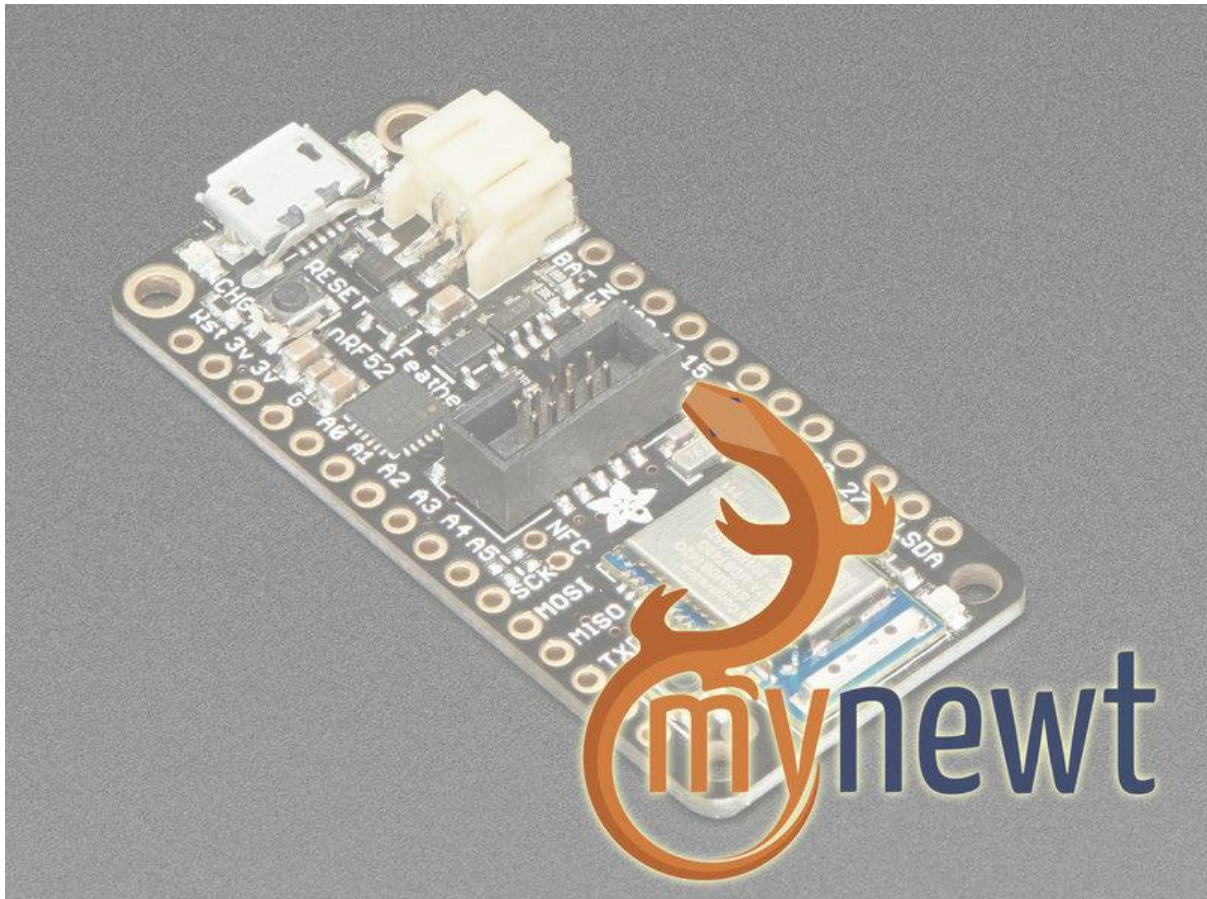




Adafruit nRF52 Pro Feather with Mynewt

Created by lady ada



<https://learn.adafruit.com/adafruit-nrf52-pro-feather>

Last updated on 2023-01-04 01:14:45 PM EST

Table of Contents

Overview	5
<ul style="list-style-type: none">• Bluetooth Certification Warning	
Device Pinout	10
<ul style="list-style-type: none">• Special Notes• Power Pins• Analog Inputs• PWM Outputs• I2C Pins	
Assembly	12
<ul style="list-style-type: none">• Header Options!• Soldering in Plain Headers• Prepare the header strip:• Add the breakout board:• And Solder!• Soldering on Female Header• Tape In Place• Flip & Tack Solder• And Solder!	
Native Installation (OS X)	22
<ul style="list-style-type: none">• Install newt and newtmgr Binaries• Install newt• Install newtmgr• Install an ARM Cross-Compiling Toolchain• Optional: Install the OpenOCD Debugger• Optional: Install Segger J-Link Drivers• Optional: Install minicom	
Native Installation (Linux)	26
<ul style="list-style-type: none">• Install newt and newtmgr Binaries• Install an ARM Cross-Compiling Toolchain• Optional: Install the OpenOCD Debugger (Segger J-Link)• Optional: Install Segger J-Link Drivers• Optional: Install minicom	
Native Installation (Win)	29
<ul style="list-style-type: none">• Cut auto-reset trace• Install newt & newtmgr Binaries• ARM Cross-Compiler	
Apache Mynewt Tools	32
<ul style="list-style-type: none">• Standard Tools• Adafruit Tools	
newt	32
<ul style="list-style-type: none">• Installing newt• Common newt Commands• newt build• newt create-image	

- newt load <target_name>
- newt size
- newt target show

newtmgr 35

- Installing newtmgr
- Connection Profiles
- Adding a Serial Connection Profile
- Listing Existing Profiles
- Test the Serial Connection Profile
- Common newtmgr Commands
- Display Scheduler Task Activity via 'taskstat'
- Uploading Application Images with newtmgr
- Display Internal Statistics

Apache Mynewt Applications 42

Create a New Project 42

- Create a Project Skeleton
- Download Project Dependencies
- Create a New Application
- Create a New Target
- 1. Set the Target's app Field
- 2. Set the Target's bsp Field
- 3. Set the build_profile Field
- 4. Test the Target Settings
- Final Project Structure
- Build and Flash the Target
- Sign the Build with a Version Number
- Flash the Image via a Segger J-Link
- Flash the Image via the Serial Port and newtmgr

Adding Tasks 48

- Declaring a task, priority and stack size
- Initializing a task
- Implementing the task handler
- Task Delays
- os_time_delay
- Example: Updating apps/first/src/main.c
- Checking Task Status

Adding Shell Commands 53

- Adding shell support
- Adding a custom command handler
- Example: Updating apps/first/src/main.c
- Testing the Shell Command

Adding Statistics 56

- Configuring Your App for Statistics
- Adding Stats to your main.c File:
- Accessing the Stats in Your Code
- Initializing the Stats
- Updating the Stats Values
- Accessing Stats with the Console or newtmgr
- Console Access
- newtmgr Access

- Example: Adding Stats to apps/first/src/main.c

Adding BLE UART Support 63

- Mynewt Nimble (BLE Stack) Documentation

Advanced Debugging 64

GDB Debugging 64

- Starting the Debugger
- Displaying Values
- Memory Manipulation
- Stack Manipulation
- Working with Source Code
- Basic Example
- Formatting Display Values
- Displaying an Array of Values
- Useful Mynewt/Nimble Structs and Fields
- Display the Stack Trace
- Display Stack Frame Details
- Displaying ARM Registers
- Displaying a function
- Displaying code at a specific address
- Running an arbitrary function when halted at a breakpoint

Command Line Debugging 73

- Grep'ing Source Code
- Grep recursively for a partial string
- Grep recursively for an exact string
- Grep recursively for a string ignoring the case
- Grep recursively with specific file types

Field Debugging Tips 74

- Debugging Crash Dumps
- Option 1: Debugging Crash Dumps with GDB
- Option 2: Debugging Crash Dumps with objdump
- Debugging Repeatable Crashes

Adafruit_Mynewt 77

- Installing Adafruit_Mynewt
- Provided Apps
- Helper Modules

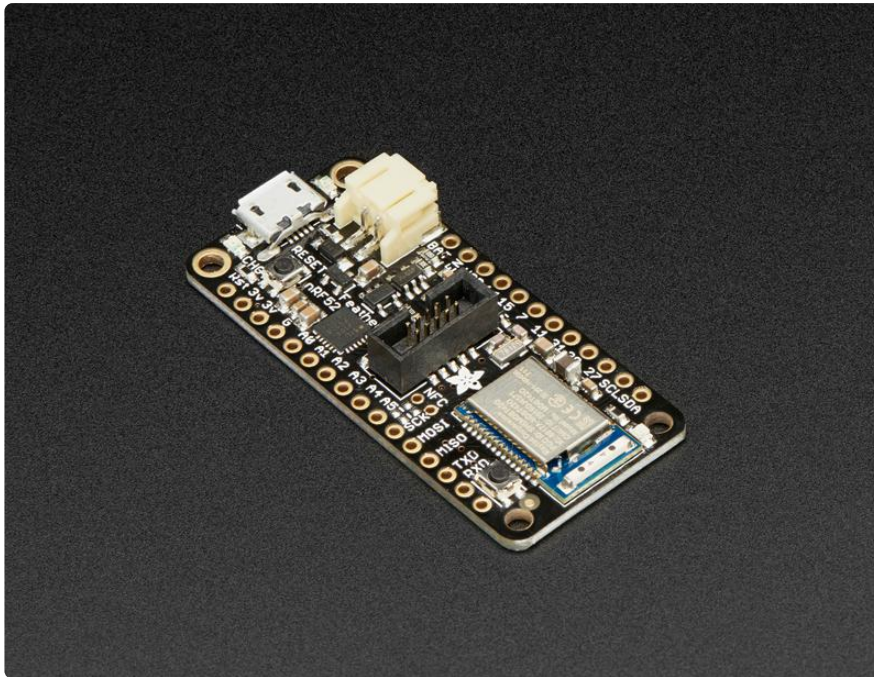
External Content 79

- Mynewt Bootloader
- Build and Package Management

Downloads 79

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

Overview

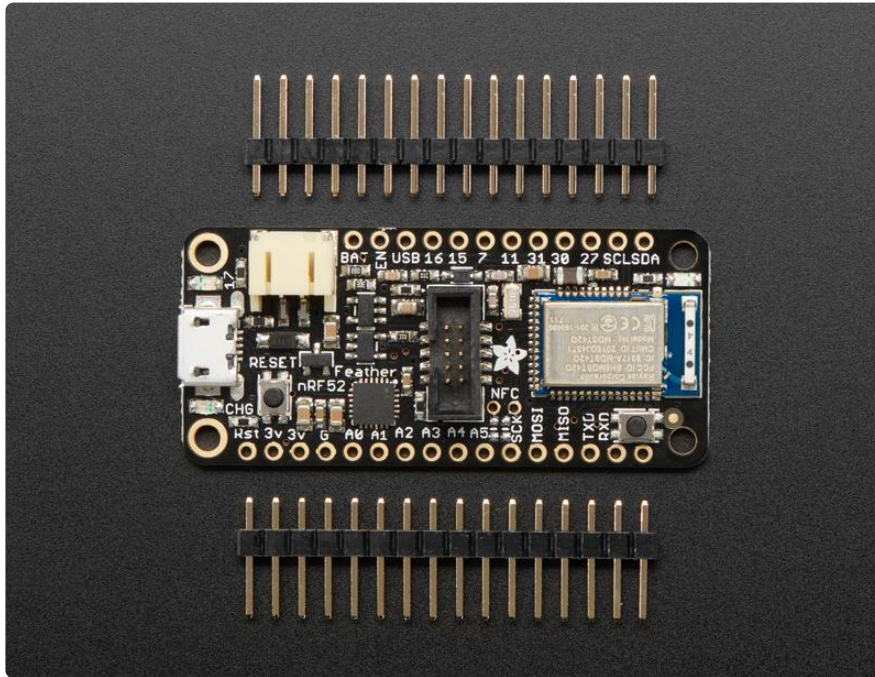


The Adafruit Feather nRF52 Pro is our latest Bluetooth Low Energy board for advanced projects and users who want to use a fully open source Bluetooth Low Energy 5.0 stack. It has a native-bluetooth chip, the nRF52832, as well as built in USB Serial and battery charging! [We have other boards in the Feather family, check'em out here.](#) ()

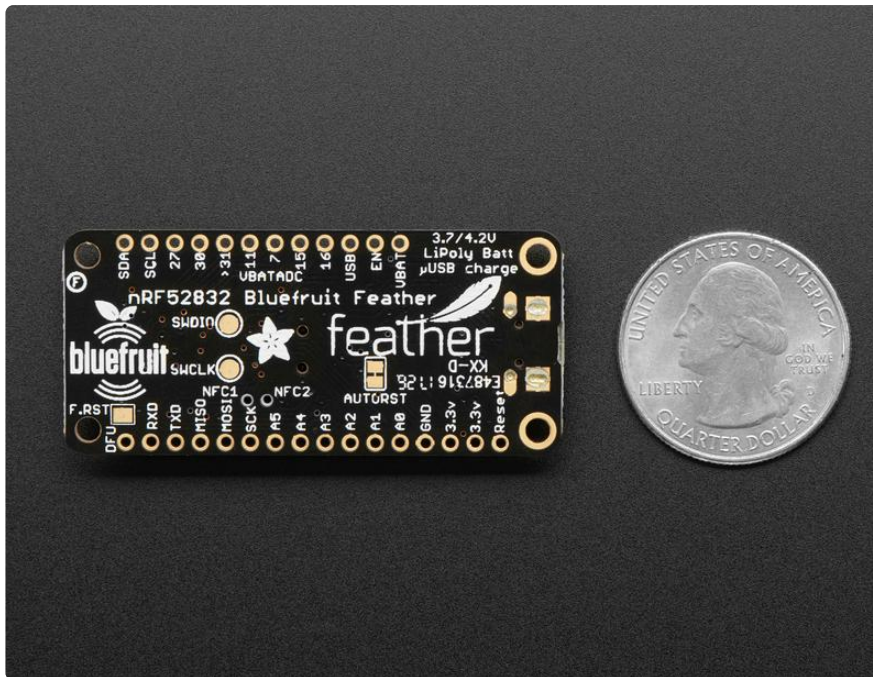


This is a special Feather - unlike the rest of the Feather family, this board is not for use with Arduino IDE. Instead, it is for use with Mynewt only! We have programmed it

with the Mynewt bootloader and updated the hardware to add an SWD connector and an additional DFU button. [If you want to use the nRF52 with Arduino IDE please check out the Bluefruit nRF52 Feather which works out-of-the-box with our Arduino board support package.](#) () This Feather is for advanced users only, you will be interacting with the Mynewt RTOS rather than Arduino and you cannot easily go back-and-forth without an SWD programmer.



This chip has twice the flash, SRAM and performance of the earlier nRF51-based modules, and can do a lot of heavy lifting. That extra processing power and memory means you can now run an RTOS for fast development of complex projects. This board was designed to work with the [Apache Mynewt](#) () operating system, which is released by the Apache Foundation under the permissive and commercial-friendly Apache license. Apache Mynewt includes a number of professionally written networking stacks (BLE 5.0, [OIC/OCF](#) (), etc.), development tools, and professional project management features like a secure bootloader that allows you to cryptographically sign firmware images and verify them during device updates.



The Adafruit Feather nRF52 Pro ships pre-programmed with the Mynewt serial bootloader that allows you to flash firmware to the device directly from the command-line using the on-board USB Serial converter and the 'newtmgr' tool, so you can get started right away with no additional hardware required. For more advanced debugging, however, you can use a Segger J-Link and the on-board SWD debug connector, which gives you access to a more powerful set of development tools and options.

Features:

- 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 \(\)](#) 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

- SWD connector for debugging
- 100% open source firmware when used with [Apache Mynewt \(\)](#)

Bluetooth Low Energy is the hottest new low-power, 2.4GHz spectrum wireless protocol. In particular, its the only wireless protocol that you can use with iOS without needing special certification and it's supported by all modern smart phones. This makes it excellent for use in portable projects that will make use of an iOS or Android phone or tablet. It also is supported in Mac OS X and Windows 8+.

To make it easy to use for portable projects, we added a connector for any of our 3.7V Lithium polymer batteries and built in battery charging. You don't need a battery, it will run just fine straight from the micro USB connector. But, if you do have a battery, you can take it on the go, then plug in the USB to recharge. The Feather will automatically switch over to USB power when it's available. We also tied the battery thru a divider to an analog pin, so you can measure and monitor the battery voltage to detect when you need a recharge.

The Power of Bluefruit LE

The nRF52 Pro uses an nRF52832 chipset from Nordic, which can be used as both a main microcontroller and a bluetooth low energy interface. For most people, they'll be very happy to use the standard Nordic UART RX/TX connection profile - example code is provided! In this profile, the Bluefruit acts as a data pipe, that can 'transparently' transmit back and forth from your iOS or Android device. You can use our [iOS App \(\)](#) or [Android App \(\)](#), or [write your own to communicate with the UART service \(\)](#).

The board is capable of much more than just sending strings over the air! Thanks to the Apache Mynewt operating system, you have full control over how the device behaves, including the ability to define and manipulate your own [GATT Services and Characteristics \(\)](#), or change the way that the device advertises itself for other Bluetooth Low Energy devices to see.

Use the Bluefruit App to get your project started

Using our Bluefruit [iOS App \(\)](#) or [Android App \(\)](#), you can quickly get your project prototyped by using your iOS or Android phone/tablet as a controller. This data can be read over BLE and processed directly by the nRF52 microcontroller

Comes fully assembled and tested, with a USB Serial bootloader that lets you quickly use it from the command line. We also toss in some header so you can solder it in and

plug into a solderless breadboard. [Lipoly battery \(\)](#) and [MicroUSB cable \(\)](#) not included (but we do have lots of options in the shop if you'd like!)

Bluetooth Certification Warning

Because of complex and expensive Bluetooth SIG certification requirements, the nRF52 Pro doesn't ship with any default Bluetooth functionality on the post-production firmware, since at present the Mynewt BLE stack has not gone through the certification process with the Bluetooth SIG. We aren't able to ship BLE functionality out of the box without this certification. In order to use Mynewt in a production product, you will need to go through the certification process for your product, although you are free to develop locally with Mynewt in a test environment.

The only Bluetooth Low Energy stack certified to run on the nRF52 modules with no additional work on your part is the Nordic SoftDevice, which is used in the Arduino variant of these boards and can be flashed onto the nRF52 Pro using your choice of ARM SWD debugger, such as the [Segger J-Link. \(\)](#)

When the Mynewt stack is certified by the Bluetooth SIG (no ETA on if or when that will happen since it's a long and expensive process), Bluetooth functionality will be enabled on shipping devices, but at present only USB Serial is used in the default firmware on production devices, and the Nordic SD should be used in commercial products if you are not able to go through the certification process yourself. There are no HW restrictions on the nRF52 Pro boards, we simply have our hands tied on what we can ship and claim coming off the production line.

UPDATE (Sept 17 2017): Runtime, one of the main contributors to the Mynewt core codebase, have contracted an external company to certify nimble (the Mynewt BLE stack). Once the certification process is completed, the results and certificate details will be posted here.

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

Special Notes

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

- PIN_DFU / P0.20: If this pin is detected to be at GND level at startup, the board will enter a special serial bootloader mode and will not execute any user code, going straight into bootloader mode. If you wish to use this pin as a standard GPIO, make sure that it is pulled high with a pullup resistor so that your code will execute normally when the MCU starts up.
- P0.31 / A7: This pin is hard wired to a voltage-divider on the LIPO battery input, allow you to safely measure the LIPO battery level on your device. If possible, you should avoid using this pin as an input because you will lose the ability to read the battery voltage. You can use it as an output just make sure to switch the pin to analog input when you want to do the battery read, then back to output when toggling pins
- FRST/P0.22: Setting this pin to GND at startup will cause the device to perform a factory reset at startup, erasing and config data as well as the user sketch. At the next reset, you should enter serial bootloader mode by default, since no user sketch will be present. You can use this to recover 'bricked' boards, but if you don't wish to do this be careful not to have FRST low at startup. By default, a weak internal pull-up resistor is enabled on this pin during the bootloader phase. FRST does not work with Mynewt bootloader !!

Power Pins

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

Analog Inputs

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

The following default values are used:

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

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

PWM Outputs

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

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

Please note that DMA based PWM output is still a work in progress in the initial release of the nRF52 BSP, and further improvements are planned here.

I2C Pins

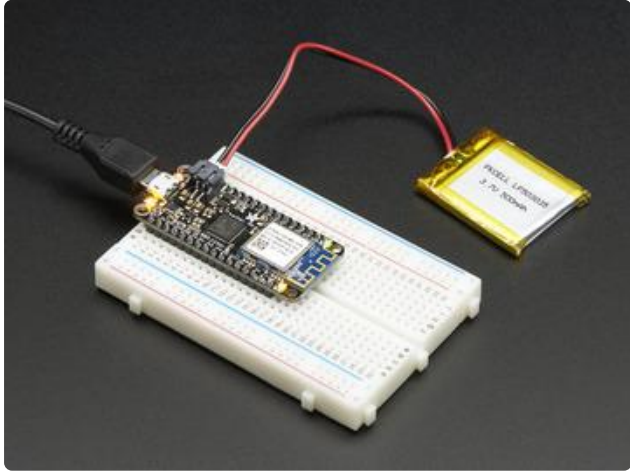
I2C pins on the nRF52832 require external pullup resistors to function, which are not present on the Adafruit nRF52 Feather by default. You will need to supply external pullups to use these. All Adafruit I2C breakouts have appropriate pullups on them already, so this normally won't be an issue for you.

Assembly

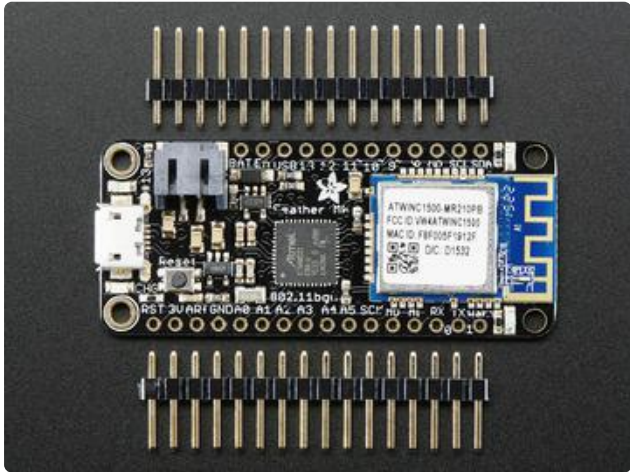
We ship Feathers fully tested but without headers attached - this gives you the most flexibility on choosing how to use and configure your Feather

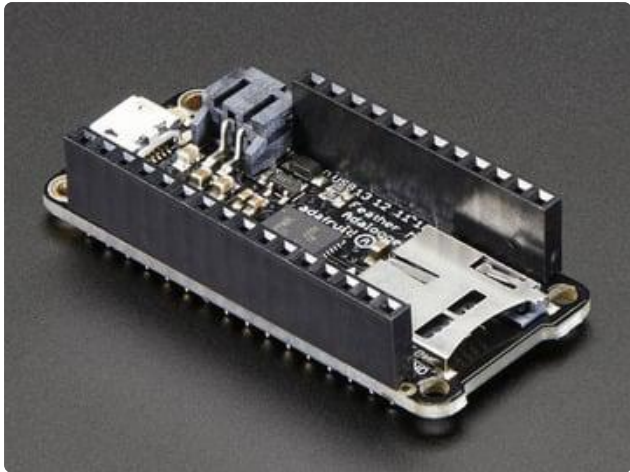
Header Options!

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

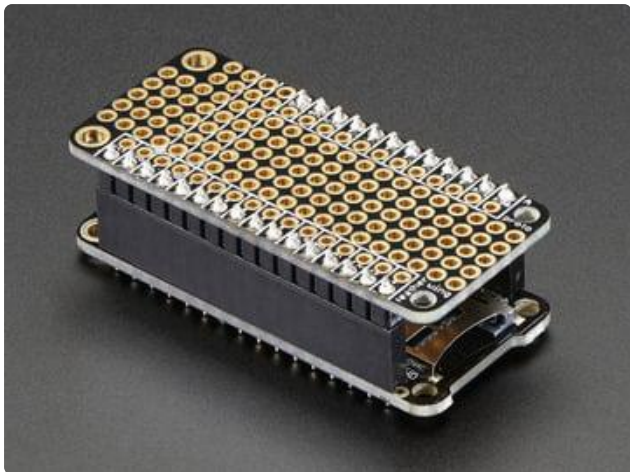


The first option is soldering in plain male headers, this lets you plug in the Feather into a solderless breadboard

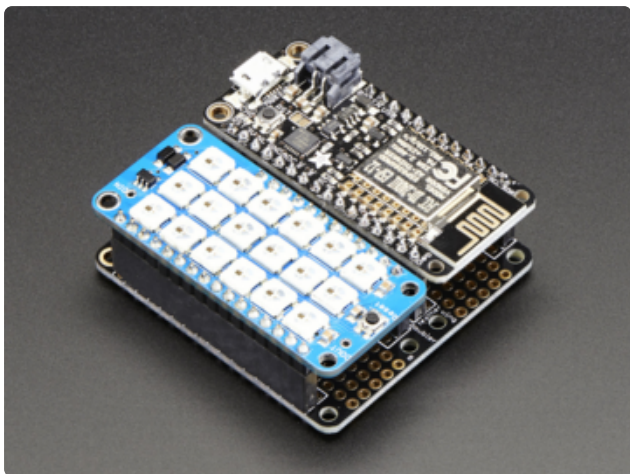


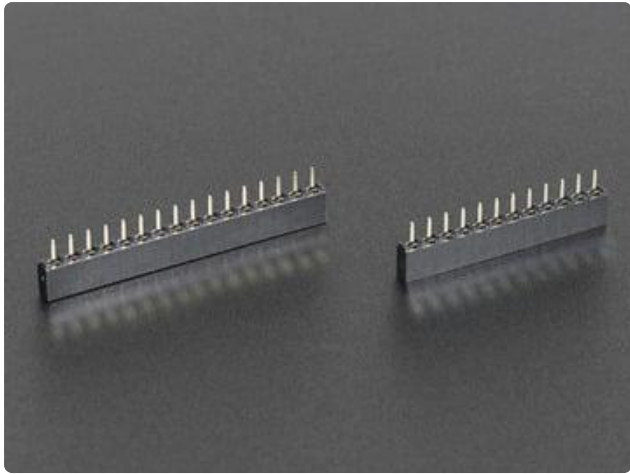


Another option is to go with socket female headers. This won't let you plug the Feather into a breadboard but it will let you attach featherwings very easily

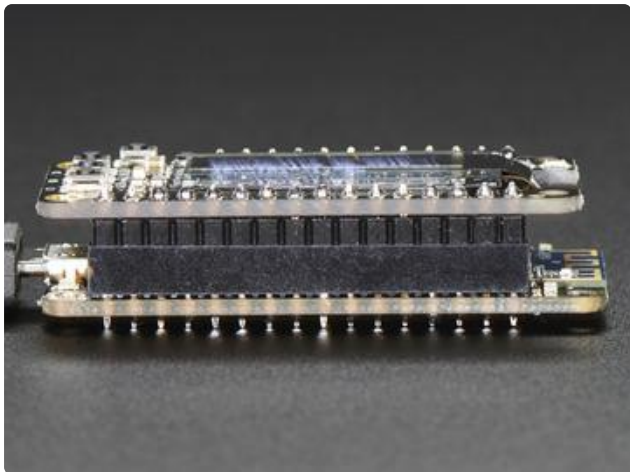


A few Feather boards require access to top-side components like buttons or connectors, making stacking impractical. Sometimes you can stack in the opposite order—FeatherWing underneath—or, if both Feather and Wing require top-side access, place the boards side-by-side with a [FeatherWing Doubler](#)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



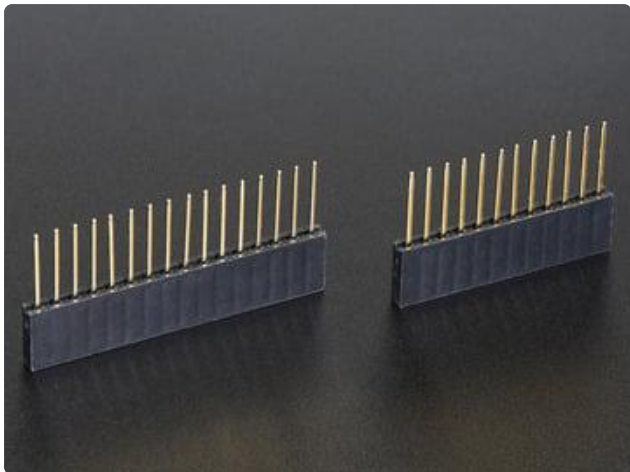


We also have 'slim' versions of the female headers, that are a little shorter and give a more compact shape

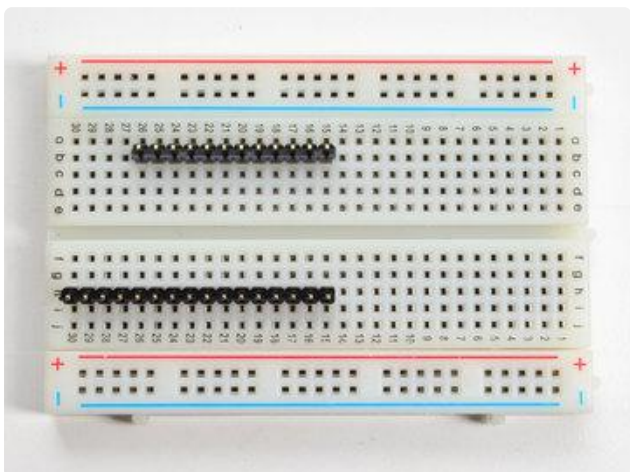




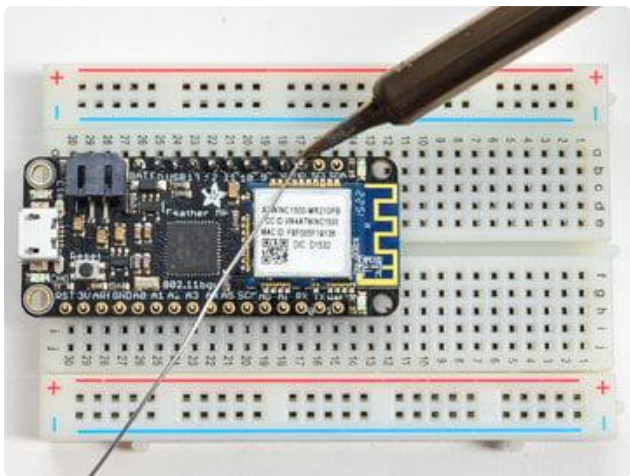
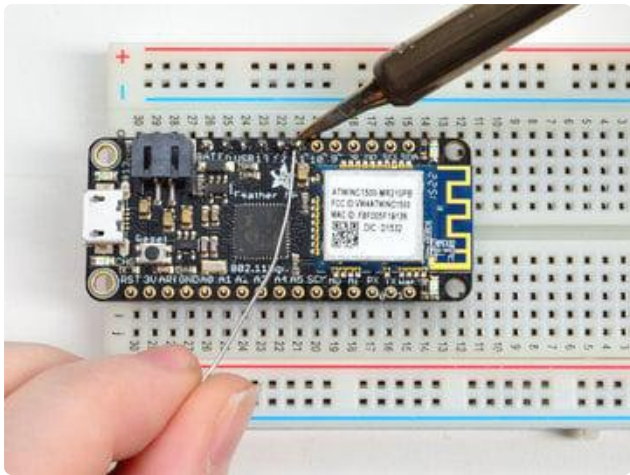
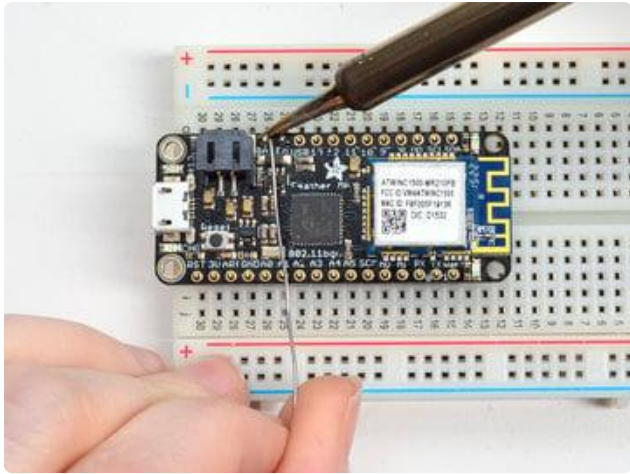
Finally, there's the "Stacking Header" option. This one is sort of the best-of-both-worlds. You get the ability to plug into a solderless breadboard and plug a featherwing on top. But its a little bulky



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



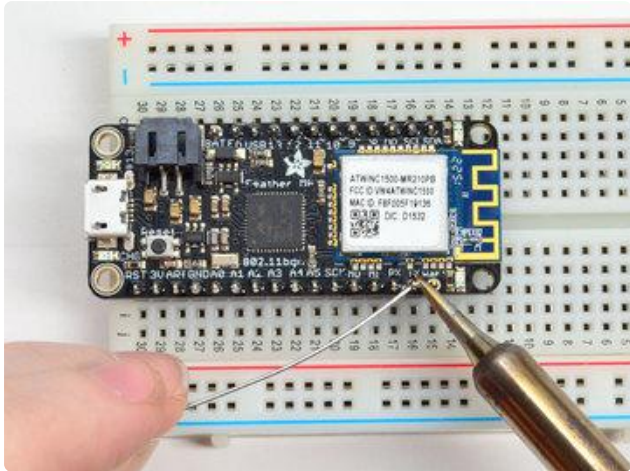
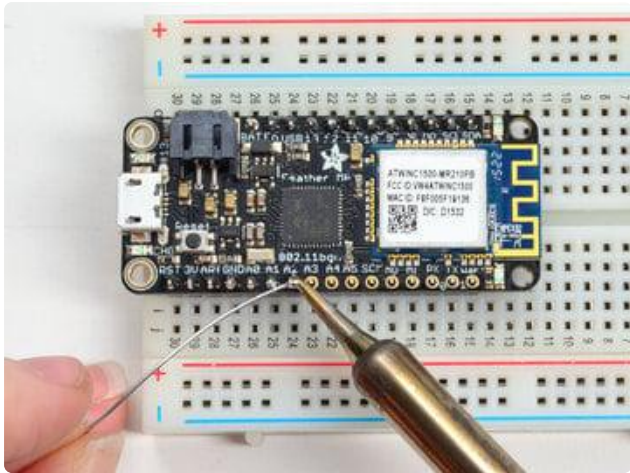
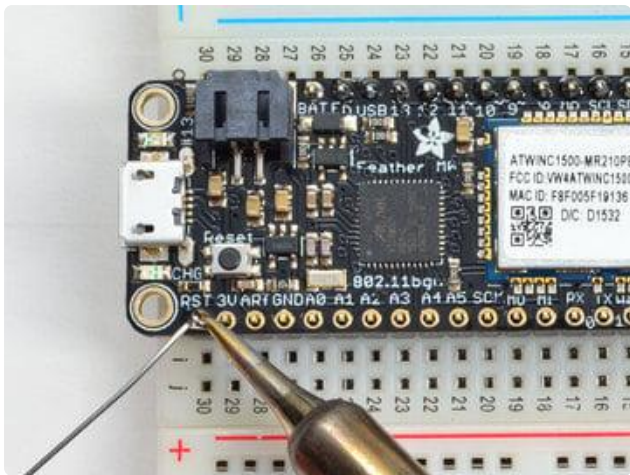
Add the breakout board:

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

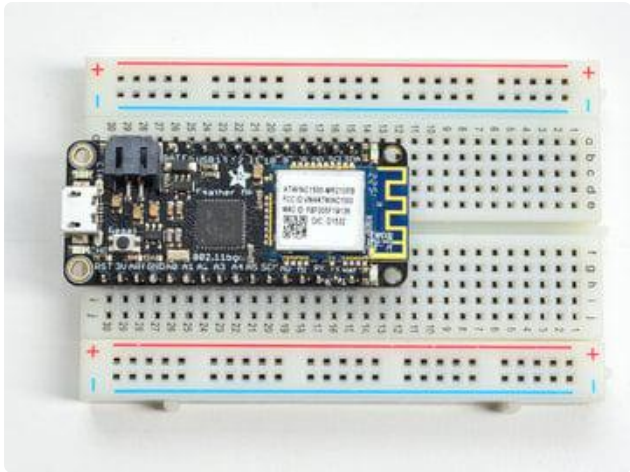
And Solder!

Be sure to solder all pins for reliable electrical contact.

(For tips on soldering, be sure to check out our [Guide to Excellent Soldering](#) ()).



Solder the other strip as well.



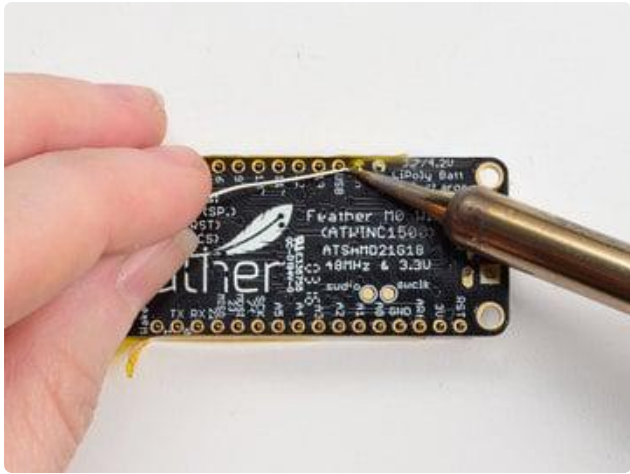
You're done! Check your solder joints visually and continue onto the next steps

Soldering on Female Header



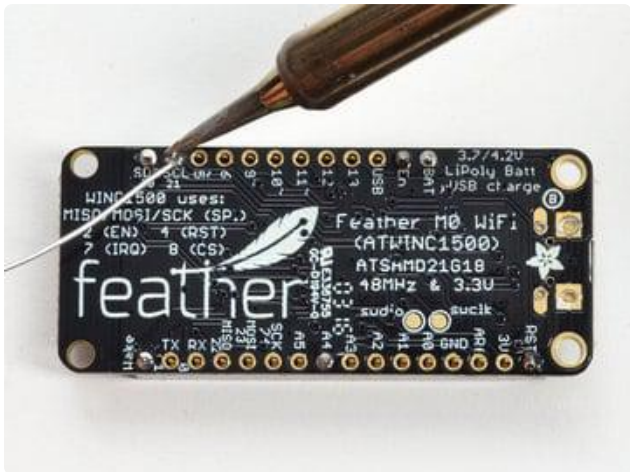
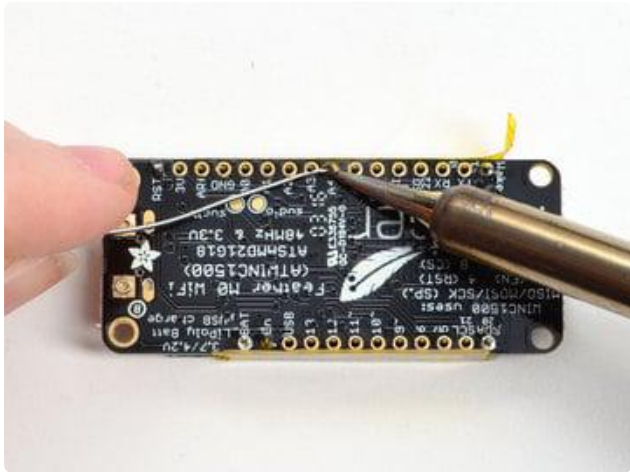
Tape In Place

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 \(\)](#)).



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



Native Installation (OS X)

Mynewt relies on two command line utilities (`newt` and `newtmgr`), both of which are described elsewhere in this learning guide, as well as the presence of an ARM cross-compiling toolchain (`arm-none-eabi-gcc` , etc.).

Both of these need to be installed and available on your system to work with Mynewt projects.

Install `newt` and `newtmgr` Binaries

Binary versions of the command-line tools are available from Apache.

Before you start, you will need to add the `JuulLabs-OSS/homebrew-mynewt` 'tap' to brew, which can be done as follows:

```
$ brew tap JuulLabs-OSS/mynewt
$ brew update
```

Install newt

The full setup guide is available here: https://mynewt.apache.org/latest/newt/install/newt_mac.html ()

```
$ brew install mynewt-newt
==&gt; Installing mynewt-newt from juullabs-oss/mynewt
==&gt; Downloading https://github.com/juullabs-oss/binary-releases/raw/master/mynewt-newt-tools_1.5.0/mynewt-newt-1.5.0.sierra.bottle
==&gt; Downloading from https://raw.githubusercontent.com/JuulLabs-OSS/binary-releases/master/mynewt-newt-tools_1.5.0/mynewt-newt-
##### 100.0%
==&gt; Pouring mynewt-newt-1.5.0.sierra.bottle.tar.gz
/usr/local/Cellar/mynewt-newt/1.5.0: 3 files, 8.1MB
```

You can test the install with:

```
$newt version
Apache Newt version: 1.5.0
```

Install newtmgr

The full setup guide is available here: https://mynewt.apache.org/latest/newtmgr/install/install_mac.html ()

```
$ brew install mynewt-newtmgr
==&gt; Installing mynewt-newtmgr from runtimeco/mynewt
==&gt; Downloading https://github.com/runtimeco/binary-releases/raw/master/mynewt-newt-tools_1.5.0/mynewt-newtmgr-1.1.0.sierra.bottle.tar.gz
==&gt; Downloading from https://raw.githubusercontent.com/runtimeco/binary-releases/master/mynewt-newt-tools_1.5.0/mynewt-newtmgr-1.5.0.sierra.bottle.tar.gz
##### 100.0%
==&gt; Pouring mynewt-newtmgr-1.5.0.sierra.bottle.tar.gz
/usr/local/Cellar/mynewt-newtmgr/1.5.0: 3 files, 17.8MB
```

You can test the install with:

```
$ newtmgr help
Newtmgr helps you manage remote devices running the Mynewt OS

Usage:
  newtmgr [flags]
  newtmgr [command]

Available Commands:
  config    Read or write a config value on a device
  conn     Manage newtmgr connection profiles
  crash    Send a crash command to a device
  datetime Manage datetime on a device
  echo     Send data to a device and display the echoed back data
  fs       Access files on a device
  help     Help about any command
```

```
image      Manage images on a device
log        Manage logs on a device
mpstat     Read mempool statistics from a device
reset      Perform a soft reset of a device
run        Run test procedures on a device
stat       Read statistics from a device
taskstat   Read task statistics from a device
```

Flags:

```
-c, --conn string      connection profile to use
-h, --help             help for newtmgr
-l, --loglevel string  log level to use (default "info")
  --name string        name of target BLE device; overrides profile setting
-t, --timeout float    timeout in seconds (partial seconds allowed) (default 10)
-r, --tries int        total number of tries in case of timeout (default 1)
```

Use "newtmgr [command] --help" for more information about a command.

Install an ARM Cross-Compiling Toolchain

In order to build and debug ARM binaries, you will also need to install a cross-compiling toolchain targeting the ARM architecture.

Thankfully, pre-built binaries are also available in brew (used to install newt and newtmgr), as described in the setup guide below:

```
$ brew tap PX4/homebrew-px4
$ brew update
$ brew install gcc-arm-none-eabi-49
```

You can test the installation with the following command (making sure that the version listed is 4.9!):

```
$ arm-none-eabi-gcc --version
arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors) 4.9.3 20150529 (release)
[ARM/embedded-4_9-branch revision 224288]
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Optional: Install the OpenOCD Debugger

If you wish to debug or flash your projects with a Segger J-Link, you will also need to install the OpenOCD debug tool via the following steps:

```
$ brew install openocd
```


If you see one of these errors:

- Library not loaded: /usr/local/lib/libusb-0.1.4.dylib
Run: `brew install libusb-compat`
- Library not loaded: /usr/local/opt/libftdi/lib/libftdi1.2.dylib
Run: `brew install libftdi`
- Library not loaded: /usr/local/lib/libhidapi.0.dylib
Run: `brew install hidapi`

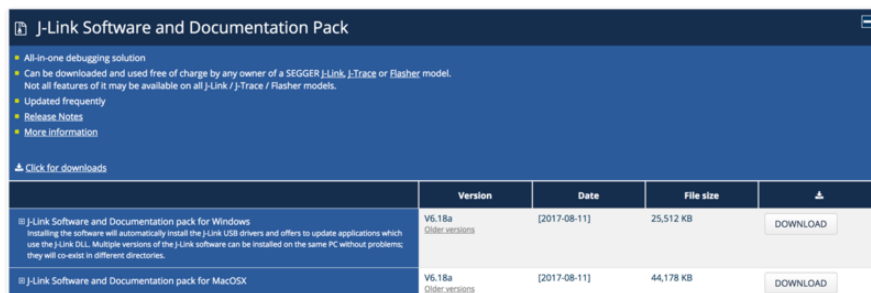
Make sure you have at least version 0.10.0, which can be verified with the following command:

```
$ openocd --version
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.org/doc/doxygen/bugs.html
```

Optional: Install Segger J-Link Drivers

If you wish to use the `newt load` or `newt debug` commands and a Segger J-Link, you will need to install the Segger J-Link drivers as well.

You will need to go to the [Segger J-Link Downloads page \(\)](#) and download the 'J-Link Software and Documentation pack for MacOSX' installer:



	Version	Date	File size	
J-Link Software and Documentation pack for Windows <small>Installing the software will automatically install the J-Link USB drivers and offers to update applications which use the J-Link DLL. Multiple versions of the J-Link software can be installed on the same PC without problems; they will co-exist in different directories.</small>	V6.18a <small>Older versions</small>	[2017-08-11]	25,512 KB	DOWNLOAD
J-Link Software and Documentation pack for MacOSX	V6.18a <small>Older versions</small>	[2017-08-11]	44,178 KB	DOWNLOAD

Once downloaded, run the installer and the JLinkExe and related J-Link commands that Mynewt relies on will be available from the command-line.

Optional: Install minicom

Much of the interaction with your Mynewt device will happen over the serial port, including working with Shell-based commands on your device.

A relatively easy to use command line application to work with serial port connections is minicom, which is used throughout this learning guide. It can be installed from the command-line with the following command:

```
$ brew install minicom
```

You can establish a serial connection via minicom with the following command (adjusting the /dev/tty device name as appropriate):

```
$ minicom -D /dev/tty.SLAB_USBtoUART
```

Native Installation (Linux)

Mynewt relies on two command line utilities (`newt` and `newtmgr`), both of which are described elsewhere in this learning guide, as well as the presence of an ARM cross-compiling toolchain (`arm-none-eabi-gcc` , etc.).

Both of these need to be installed and available on your system to work with Mynewt projects.

Install `newt` and `newtmgr` Binaries

Binary versions of the command-line tools are available from Apache.

The full setup guide is available here: https://mynewt.apache.org/latest/newt/install/newt_linux.html ()

In summary, copy following commands and paste it (one line each time) into your terminal to add newt's public key and deb package to your apt repository for installation.

```
$ wget -qO - https://raw.githubusercontent.com/JuulLabs-OSS/debian-mynewt/master/mynewt.gpg.key | sudo apt-key add -
$ sudo tee /etc/apt/sources.list.d/mynewt.list && EOF
deb https://raw.githubusercontent.com/JuulLabs-OSS/debian-mynewt/master latest main
EOF
$ sudo apt-get update
$ sudo apt-get install newt
$ sudo apt-get install newtmgr
```

Check your installation with version command

```
$ newt version
Apache Newt version: 1.5.0
$ newtmgr version
Apache Newtmgr 1.5.0
```

Install an ARM Cross-Compiling Toolchain

In order to build and debug ARM binaries, you will also need to install a cross-compiling toolchain targeting the ARM architecture.

Thankfully, pre-built binaries are available, as described in the setup guide below:

```
$ sudo apt-get remove binutils-arm-none-eabi gcc-arm-none-eabi
$ sudo add-apt-repository ppa:team-gcc-arm-embedded/ppa
$ sudo apt-get update
$ sudo apt-get install gcc-arm-none-eabi
```

Optional: Install the OpenOCD Debugger (Segger J-Link)

If you wish to debug or flash your projects with a Segger J-Link, you will also need to install the OpenOCD debug tool via the following steps:

```
$ sudo apt-get install openocd
```

Once it is installed, you can check and make sure you have at least v 0.10.0 which support nRF52 chip

```
$openocd -v
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
```

If you see any of these error messages:

- openocd: error while loading shared libraries: libhidapi-hidraw.so.0: cannot open shared object file: No such file or directory
- openocd: error while loading shared libraries: libusb-1.0.so.0: cannot open shared object file: No such file or directory

run the following command to install the libraries:

```
$ sudo apt-get install libhidapi-dev:i386
```

Optional: Install Segger J-Link Drivers

If you wish to use the `newt load` or `newt debug` commands and a Segger J-Link, you will need to install the Segger J-Link drivers as well.

On Ubuntu systems, you will need to go to the [Segger J-Link Downloads page \(\)](#) and download an appropriate binary package:

- For 32-bit systems download:
J-Link Software and Documentation pack for Linux, DEB Installer, 32-bit
- For 64-bit systems download:
J-Link Software and Documentation pack for Linux, DEB Installer, 64-bit

Once you have the file on your system, install the package via dpkg as follows (varying the file for the version of architecture you are using):

```
$ sudo dpkg -i JLink_Linux_V618a_x86_64.deb
Selecting previously unselected package jlink.
(Reading database ... 175450 files and directories currently installed.)
Preparing to unpack JLink_Linux_V618a_x86_64.deb ...
Removing /opt/SEGGER/JLink ...
/opt/SEGGER/JLink not found (OK)
Unpacking jlink (6.18.1) ...
Setting up jlink (6.18.1) ...
```

Optional: Install minicom

Much of the interaction with your Mynewt device will happen over the serial port, including working with Shell-based commands on your device.

A relatively easy to use command line application to work with serial port connections is minicom, which is used throughout this learning guide. It can be installed from the command-line with the following command:

```
$ sudo apt-get install minicom
```

You can establish a serial connection via minicom with the following command (adjusting the /dev/tty device name as appropriate):

```
$ sudo minicom -D /dev/ttyUSB0
```

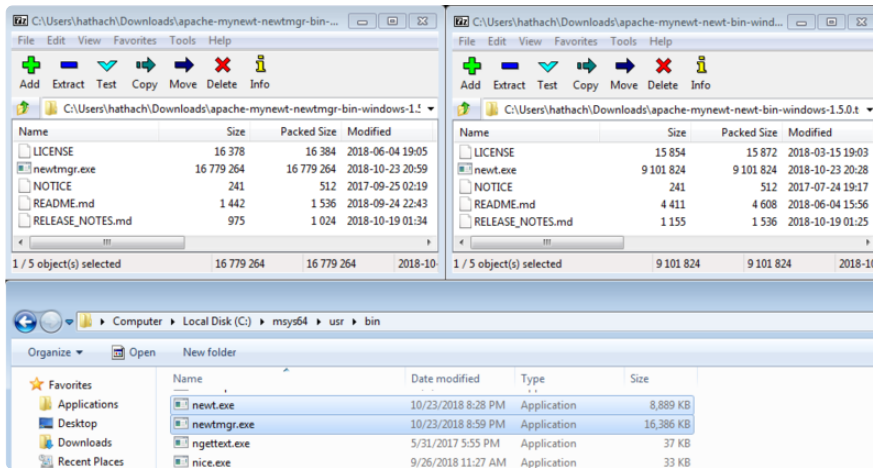
Native Installation (Win)

Mynewt relies on two command line utilities (`newt` and `newtmgr`), both of which are described elsewhere in this learning guide, as well as the presence of an ARM cross-compiling toolchain (`arm-none-eabi-gcc` , etc.).

Both of these need to be installed and available on your system to work with Mynewt projects.

Cut auto-reset trace

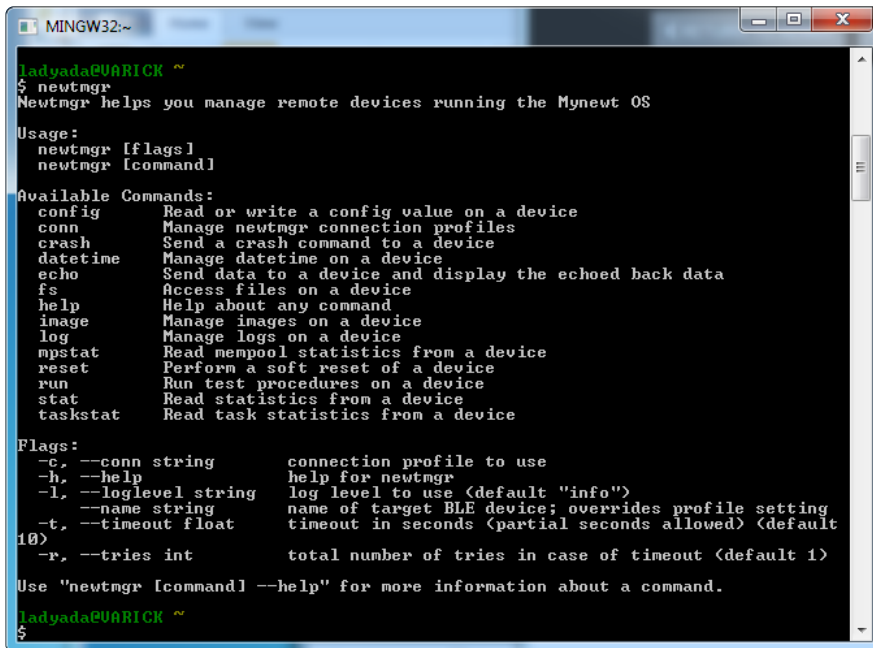
Windows serial has a somewhat annoying thing it does where it toggles the DTR/RTS lines when you open a serial port. This will make debugging impossible since you keep resetting the board. Before you continue you must cut the trace-jumper on the bottom of the board:



You can test it by running newt and newtmg from the shell

```
$ newt version
Apache Newt version: 1.5.0

$ newtmg version
Apache Newtmg 1.5.0
```



ARM Cross-Compiler

You will also need an cross-compiling toolchain for the ARM architecture, specifically [GCC 4.9 \(\)](#), since newt will call arm-none-eabi-gcc from the command line.

For details on installing this see the official Mynewt documentation here: [https://mynewt.apache.org/latest/get_started/native_install/cross_tools.html \(\)](https://mynewt.apache.org/latest/get_started/native_install/cross_tools.html)

NOTE: arm-none-eabi-gcc also needs to be available in the system PATH variable to work with newt!

Download GCC for ARM 4.9 2015 Q2 for Windows 32-bit

Apache Mynewt Tools

Standard Tools

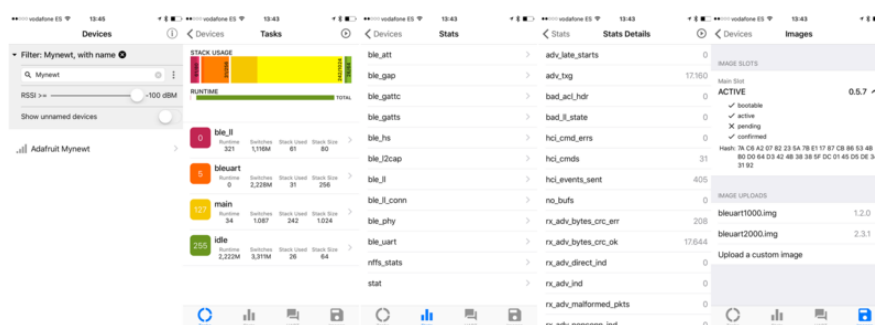
There are two main tools used to interact with Mynewt devices:

1. [newt \(\)](#): Build system and project management tool
2. [newtmgr \(\)](#): Field management tool for deployed devices

Consult the dedicated tool pages for details on how to setup and use these two important tools.

Adafruit Tools

In addition to the default system tools, Adafruit maintains an iOS app called [Adafruit Mynewt Manager \(\)](#) that can be useful to monitor and work with deployed devices via your mobile phone.



newt

newt is a build and project management tool that hides most of the messy details of creating, building and debugging mynewt projects. With newt, you don't need to worry about makefiles, compiler flags, debug config settings, and library versions or updates. The details are elegantly hidden behind the easy to use 'newt' wrapper.

For full details on newt and the entire command set see the [official newt documentation](#) ().

Installing newt

For details on installing the newt tool on your operating system of choice see:

- OS X Installation: https://mynewt.apache.org/latest/newt/install/newt_mac.html ()
- Linux Installation: https://mynewt.apache.org/latest/newt/install/newt_linux.html ()
- Windows Installation: https://mynewt.apache.org/latest/newt/install/newt_windows.html ()

Common **newt** Commands

Please consult the [official newt documentation](#) () for details on every command, but the most commonly used commands are detailed below:

newt build

This command will build a firmware image for the specified target name, and will handle things like the makefile generation and any compiler parameters and file inclusions required by the target.

Full documentation: https://mynewt.apache.org/latest/newt/command_list/newt_build.html ()

```
$ newt build first
Building target targets/first
...
Target successfully built: targets/first
```

newt create-image

This command assigns a version number to the image, and can also be used to optionally cryptographically sign images with an appropriate .pem key file.

Full documentation: https://mynewt.apache.org/latest/newt/command_list/newt_create_image.html ()

```
$ newt create-image first 1.2.3
App image succesfully generated: [PATH]/bin/targets/first/app/apps/first/first.img
```

newt load <target_name>

This command will use a Segger J-Link (if you have one connected) to flash the version controlled .img file to the device.

Make sure you run 'newt create-image' on the firmware image before loading it to the device, or the firmware will be rejected by the bootloader which verifies all firmware at startup!

Full documentation: https://mynewt.apache.org/latest/newt/command_list/newt_load.html ()

```
$ newt load first
Loading app image into slot 1
```

newt size

This command will give you detailed information on the flash and SRAM usage of your firmware, and can be helpful when debugging or when tight on space.

Full documentation: https://mynewt.apache.org/latest/newt/command_list/newt_size.html ()

```
$ newt size first
Size of Application Image: app
FLASH      RAM
   59      246 *fill*
   32         0 apps_first.a
 1404      104 boot_bootutil.a
   426       26 boot_split.a
 1236         0 crypto_mbedtls.a
   492         0 encoding_base64.a
 1228         0 encoding_cborattr.a
 2912         0 encoding_tinycbor.a
   540      496 hw_bsp_ada_feather_nrf52.a
    64         0 hw_cmsis-core.a
   256         0 hw_drivers_uart_uart_hal.a
   392         0 hw_hal.a
 3744       72 hw_mcu_nordic_nrf52xxx.a
 7316     8233 kernel_os.a
 2856       44 libc_baselibc.a
 2416       24 mgmt_imgmgr.a
   220       20 mgmt_mgmt.a
   884      100 mgmt_newtmgr.a
 1401       44 mgmt_newtmgr_nmgr_os.a
   136       32 mgmt_newtmgr_transport_nmgr_shell.a
```

```

1791      37 sys_config.a
2203     100 sys_console_full.a
 544     128 sys_flash_map.a
3171      90 sys_log_full.a
 408      12 sys_mfg.a
 752      76 sys_reboot.a
5324     774 sys_shell.a
1385      73 sys_stats_full.a
  32       5 sys_sysinit.a
1676      0 time_datetime.a
 556      0 util_crc.a
 124      0 util_mem.a
 128      0 first-sysinit-app.a
1492      0 libgcc.a

```

```

objsize
text    data    bss     dec     hex filename
47568   316    10020   57904   e230 [PATH]/bin/targets/first/app/apps/first/
first.elf

```

newt target show

This will give you a list of all targets defined in your project, and some basic details about them.

Full documentation: https://mynewt.apache.org/latest/newt/command_list/newt_target.html ()

```

$ newt target show
targets/first
  app=apps/first
  bsp=@apache-mynewt-core/hw/bsp/ada_feather_nrf52
  build_profile=debug
targets/my_blinky_sim
  app=apps/blinky
  bsp=@apache-mynewt-core/hw/bsp/native
  build_profile=debug

```

newtmgr

newtmgr is an image management tool that can be used to interact with the bootloader and images on the device. For full details on the newtmgr tool and the entire command set see the [official newtmgr documentation](#) ().

Installing newtmgr

For details on installing the newtmgr tool on your operating system of choice see:

- OS X Installation: https://mynewt.apache.org/latest/newt/install/newt_mac.html ()

- Linux Installation: https://mynewt.apache.org/latest/newt/install/newt_linux.html ()
- Windows Installation: https://mynewt.apache.org/latest/newt/install/newt_windows.html ()

Connection Profiles

The `newtmgr` tool works with connection profiles, such as serial or ble, depending on how you wish to communicate with the device under test.

Before you can use `newtmgr` you will need to setup at least one connection profile, as described below:

Adding a Serial Connection Profile

```
$ newtmgr conn add serial1 type=serial connstring=/dev/tty.SLAB_USBtoUART
```

Or, on windows:

```
$ newtmgr conn add serial1 type=serial connstring=COM14
```

Listing Existing Profiles

You can get a list of all defined connection profiles via:

```
$ newtmgr conn show
Connection profiles:
  serial1: type=serial, connstring='/dev/tty.SLAB_USBtoUART'
```

Test the Serial Connection Profile

You can test the connection profile with the `'image list'` command, which will list any application images present in flash memory:

```
$ newtmgr -c serial1 image list
Images:
  slot=0
    version: 0.1.0
    bootable: true
    flags: active confirmed
```

```
hash: be52a255c25546dacc497d62faea910459903a1c1916ce831697d40fc2c20689
Split status: N/A (0)
```

The image shown above is the default image that the nRF52 Pro ships with

Common newtmgr Commands

The following commands are commonly used when doing day to day development and debugging with Mynewt based devices:

Display Scheduler Task Activity via 'taskstat'

You can also test the connection with `newtmgr` using the `taskstat` command, which will list some basic statistics about the tasks running in the task manager.

Enter the following command:

```
$ newtmgr -c serial1 taskstat
```

Which should give you the following task statistics (output will vary depending on the Mynewt application that is running at the time):

task	pri	tid	runtime	csw	stksz	stkuse	last_checkin	next_checkin
ble_ll	0	2	21	4381	80	58	0	0
bleuart	5	3	0	12030	256	31	0	0
idle	255	0	11980	16292	64	26	0	0
main	127	1	0	27	1024	210	0	0

Uploading Application Images with newtmgr

If you don't have a Segger J-Link or similar debugger, the `newtmgr` tool can be used to upload an application image to the secondary bank of flash memory, and then switch application images during the next reset. This allows you to perform field updates over an inexpensive serial connection with no external HW required.

1. Build the Target Application (`newt build`)

The first step is of course to build the target application that you wish to deploy. The default `bleuart ()` project is used in this case:

```
$ newt build bleuart
Building target targets/bleuart
...
Target successfully built: targets/bleuart
```

2. Sign and Version the Image (`newt create-image`)

You then need to 'sign' the image you just built with a version number and meta data used to validate the image when flashing, which is done via the `create-image` command:

```
$ newt create-image bleuart 0.2.0
App image successfully generated: [PATH]/bin/targets/bleuart/app/apps/bleuart/bleuart.img
```

Pay attention to the `.img` output filename since you will need this full path in the next step!

3. Upload the Image (`newtmgr image upload`)

Now that we have a signed and versioned firmware image, we can copy the image to bank 1 of flash memory using the serial bootloader and the `newtmgr` tool:

```
$ newtmgr -c serial1 image upload [PATH]/bin/targets/bleuart/app/apps/bleuart/bleuart.img
353
704
...
157426
157612
Done
```

At this point, you can see that there are two images stored on the device in two different flash banks, the original 0.1.0 image that the board ships with, and the new 0.2.0 image we just built, signed and uploaded:

```
$ newtmgr -c serial1 image list
Images:
slot=0
  version: 0.1.0
  bootable: true
  flags: active confirmed
  hash: be52a255c25546dacc497d62faea910459903a1c1916ce831697d40fc2c20689
slot=1
  version: 0.2.0
  bootable: true
  flags:
  hash: 87276847693699896f68b3c26d378648cace2900db4145cd5ade6049ac5ec15a
Split status: N/A (0)
```

Pay attention to the 'flags' field, which indicates the state of images on the system. This value will change as we run through the update process.

4. Test the Image Upload (`newtmgr image test [hash]`)

Now that the image has been uploaded successfully to the secondary bank of flash memory, we need to tell the system to 'test' the image during the next reset.

What the test command will do is check the signature of the image, and then try to switch the two flash banks so that the old application (0.1.0) is in the secondary flash bank, and the new application (0.2.0) is in the primary bank.

This happens at reset, and will cause a 15-20 second delay as the flash writes are happening, so be sure to wait for the testing process to complete before interacting or interfering with the device!

You can issue a test (and flash bank switch) request as follows, using the hash value from the `image list` command above:

```
$ newtmgr -c serial1 image test
87276847693699896f68b3c26d378648cace2900db4145cd5ade6049ac5ec15a
Images:
slot=0
  version: 0.1.0
  bootable: true
  flags: active confirmed
  hash: be52a255c25546dacc497d62faea910459903a1c1916ce831697d40fc2c20689
slot=1
  version: 0.2.0
  bootable: true
  flags: pending
  hash: 87276847693699896f68b3c26d378648cace2900db4145cd5ade6049ac5ec15a
Split status: N/A (0)
```

Notice that the 0.2.0 image flag has been changed to PENDING, which means a test and flash bank switch is pending on the next reset!

5. Reset to Perform the Test and Image Bank Switch (`newtmgr reset`)

For the flash bank switch and test procedure to complete, you must reset the device, which can be done from the command line as follows:

```
$ newtmgr -c serial1 reset
Done
```

Be sure to wait at least 20 seconds after issuing the reset command, since the flash write takes 15-20 seconds to complete and the device will appear unresponsive during the update process!

If you run the image list command again you will see that the banks and flags values have changed, with version 0.1.0 in the secondary bank and it's status set to 'confirmed':

```
$ newtmgr -c serial1 image list
Images:
slot=0
  version: 0.2.0
  bootable: true
  flags: active
  hash: 87276847693699896f68b3c26d378648cace2900db4145cd5ade6049ac5ec15a
slot=1
  version: 0.1.0
  bootable: true
  flags: confirmed
  hash: be52a255c25546dacc497d62faea910459903a1c1916ce831697d40fc2c20689
Split status: N/A (0)
```

6. Confirm the Image Switch (`newtmgr image confirm`)

The final step is to 'confirm' the image update process to make it permanent, which is done via the `image confirm` sub-command:

```
$ newtmgr -c serial1 image confirm
```

As of newtmgr 1.1.0 the 'confirm' command will always return "Error: 1", but the confirm process does work, as verified with the image list command below!

If you check the flags values again, you will see that version 0.2.0 is now 'confirmed' and will continue to execute every time you reset the device:

```
$ newtmgr -c serial1 image list
Images:
slot=0
  version: 0.2.0
  bootable: true
  flags: active confirmed
  hash: 87276847693699896f68b3c26d378648cace2900db4145cd5ade6049ac5ec15a
slot=1
  version: 0.1.0
  bootable: true
  flags:
  hash: be52a255c25546dacc497d62faea910459903a1c1916ce831697d40fc2c20689
Split status: N/A (0)
```


If you reset without confirming, the device will revert to the original (0.1.0) image!

Note: You can optionally skip the 'test + reset' phase, and go directly to confirming an image after the upload is finished by using 'newtmgr -c serial1 confirm [hash]' with the hash of the image to finalize. This is a more dangerous choice, but the option is there if you wish to save some time.

Display Internal Statistics

Mynewt has an internal statistics system that is very useful when debugging issues in the field or during development. You can defined individual statistics fields, and increment the values at appropriate moments, and access these from the command-line with the `newtmgr stat` command set.

List all Statistic Groups

```
$ newtmgr -c serial1 stat list
stat groups:
  ble_att
  ble_gap
  ble_gattc
  ble_gatts
  ble_hs
  ble_l2cap
  ble_ll
  ble_ll_conn
  ble_phy
  ble_uart
  nffs_stats
  stat
```

List all Values in a Statistic Group

```
$ newtmgr -c serial1 stat ble_phy
stat group: ble_phy
  8310 phy_isrs
    0 radio_state_errs
    15 rx_aborts
    0 rx_crc_err
    0 rx_hw_err
    0 rx_late
    15 rx_starts
    0 rx_valid
  265440 tx_bytes
    0 tx_fail
    8295 tx_good
    0 tx_hw_err
    0 tx_late
```

Apache Mynewt Applications

This section contains a guide on creating a progressively more complex application in Mynewt, starting with the absolute basics and adding useful features one by one.

The individual pages can be references in isolation, but the example code builds up from one example to the next in chronological order.

- [Create a New Project \(\)](#): This page shows how to generate a new project, and add a custom application and build target to the project.
- [Adding Tasks \(\)](#): Add a custom task to the task scheduler
- [Adding Shell Commands \(\)](#): Add a custom shell command to the serial-based console
- [Adding Statistics \(\)](#): Add custom statistics to track the internal state of the system in an easy to monitor fashion

Create a New Project

This page makes the following assumptions:

1. You have already installed the `newt` and `newtmgr` tools on your system (see the Native Installation pages in this learning guide for details)
2. The various build tools (`arm-none-eabi-gcc` , etc.) are available from the command-line
3. You are using Mynewt version 1.1 for the tools and core codebase

This project will have the following features:

- `newtmgr` support will be enabled over the serial port
- Shell support will be included, allowing you to enter commands over the serial connection

Create a Project Skeleton

The first thing to do is create a project skeleton via the following command:

We'll use 'myproject' as a project name here, but you are free to change it to something else

```
$ newt new myproject
```

This should give you the following output:

```
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in myproject...
Project myproject successfully created.
```

Download Project Dependencies

Next download the project dependencies from the associated repos.

Since this is a new project, only `apache-mynewt-core` will be downloaded:

```
$ cd myproject
$ newt install -v
```

Depending on your system state, this should give you the following output:

```
$ newt install -v
[apache-mynewt-core]:
Downloading repository description
Download successful!
Downloading repository mynewt-core (branch: master; commit: mynewt_1_1_0_tag) at
https://github.com/apache/mynewt-core.git
Cloning into '/var/folders/86/hb2vp14n5_5_yvdz_z8w9x_c0000gn/T/newt-
repo021712424'...
remote: Counting objects: 65349, done.
remote: Compressing objects: 100% (151/151), done.
remote: Total 65349 (delta 130), reused 162 (delta 95), pack-reused 65099
Receiving objects: 100% (65349/65349), 80.03 MiB | 1.94 MiB/s, done.
Resolving deltas: 100% (39656/39656), done.
Will create new branch mynewt_1_1_0_tag from tag tags/mynewt_1_1_0_tag
apache-mynewt-core successfully installed version 1.1.0-none
```

This should give you the following project structure:

```
.
├── LICENSE
├── NOTICE
├── README.md
├── apps
│   └── blinky
│       ├── pkg.yml
│       └── src
├── project.state
├── project.yml
├── repos
│   └── apache-mynewt-core
├── targets
│   └── my_blinky_sim
│       └── pkg.yml
```

```
└─ target.yml
└─ unittest
└─ pkg.yml
└─ target.yml
```

Create a New Application

Now that you have a project skeleton, you can start adding applications to it.

A Mynewt app requires at least a `main()` function, a `pkg.yml` file, and a most of the time a `syscfg.yml` file, as well as a new 'app' folder where the files should be stored.

To get started, create a new app folder as follows:

```
$ mkdir apps/first
```

We'll call this application 'first', but you can use any name you find appropriate

Next create the core `apps/first/pkg.yml` file with the following text (for example via `$ nano apps/first/pkg.yml`, or using your favorite text editor):

This file lists the external dependencies for your project

```
pkg.name: apps/first
pkg.type: app

pkg.deps:
- "@apache-mynewt-core/libc/baselibc"
- "@apache-mynewt-core/kernel/os"
- "@apache-mynewt-core/sys/sysinit"
- "@apache-mynewt-core/sys/shell"
- "@apache-mynewt-core/sys/console/full"
- "@apache-mynewt-core/sys/log/full"
- "@apache-mynewt-core/sys/stats/full"
- "@apache-mynewt-core/hw/hal"
- "@apache-mynewt-core/mgmt/imgmgr"
- "@apache-mynewt-core/mgmt/newtmgr"
- "@apache-mynewt-core/mgmt/newtmgr/transport/nmgr_shell"
- "@apache-mynewt-core/boot/split"
- "@apache-mynewt-core/boot/bootutil"
```

Next create a `apps/first/syscfg.yml` file with the following content (for example `$ nano apps/first/syscfg.yml`):

This file contains config settings for your application to control what features and used or enabled during the build process

```

    syscfg.vals:
# Use INFO log level to reduce code size.  DEBUG is too large for nRF51.
LOG_LEVEL: 1
REBOOT_LOG_CONSOLE: 1

# Enable the shell task.
SHELL_TASK: 1

# Include names for statistics.
STATS_NAMES: 1

# Enable shell commands.
STATS_CL_I: 1
LOG_CL_I: 1
CONFIG_CL_I: 1

# Enable newtmgr commands.
STATS_NEWTMGR: 1
LOG_NEWTMGR: 1
CONFIG_NEWTMGR: 1

```

Finally create a `apps/first/src/main.c` file where the main source code will be stored:

```

$ mkdir -p apps/first/src
$ nano apps/first/src/main.c

```

... and enter the following code:

```

#include <assert.h>;
#include <string.h>;

#include "os/os.h"
#include "bsp/bsp.h"
#include "hal/hal_gpio.h"
#include "sysinit/sysinit.h"

int
main(int argc, char **argv)
{
    int rc;

    /* Initialize the OS */
    sysinit();

    /* Configure the LED GPIO as an output and HIGH (0n) */
    hal_gpio_init_out(LED_BLINK_PIN, 1);

    while (1) {
        /* Run the event queue to process background events */
        os_eventq_run(os_eventq_dflt_get());
    }

    return rc;
}

```

Create a New Target

Next, you need to create a new target that points to your app via the following command:

```
$ newt target create first
Target targets/first successfully created
```

... and then configure the new target with the following mandatory fields:

1. Set the Target's app Field

Point the new target to the app we created above via:

```
$ newt target set first app=apps/first
Target targets/first successfully set target.app to apps/first
```

2. Set the Target's bsp Field

Next set the bsp (board support package) for the new target, which indicates the HW that the app will be running on.

If you are using the Adafruit nRF52 Pro Feather this should be:

```
$ newt target set first bsp=@apache-mynewt-core/hw/bsp/ada_feather_nrf52
Target targets/first successfully set target.bsp to @apache-mynewt-core/hw/bsp/ada_feather_nrf52
```

3. Set the build_profile Field

Finally set the build profile for the new target (`debug` or `optimized`):

```
$ newt target set first build_profile=debug
Target targets/first successfully set target.build_profile to debug
```

4. Test the Target Settings

You can review the target settings via the following command:

'my_blinky_sim' is an artifact of the default project creation process and can be ignored

```
$ newt target show first
targets/first
  app=apps/first
  bsp=@apache-mynewt-core/hw/bsp/ada_feather_nrf52
  build_profile=debug
targets/my_blinky_sim
  app=apps/blinky
  bsp=@apache-mynewt-core/hw/bsp/native
  build_profile=debug
```

Final Project Structure

The final project structure should look like this:

```
apps/first
├── LICENSE
├── NOTICE
├── README.md
├── apps
│   └── blinky
│       ├── pkg.yml
│       └── src
│           └── main.c
├── pkg.yml
├── project.yml
├── src
│   └── main.c
└── syscfg.yml
```

Build and Flash the Target

Finally, you can build your target, pointing to the new application, and using the specified target BSP and build profile:

```
$ newt build first
Building target targets/first
Compiling repos/apache-mynewt-core/boot/bootutil/src/loader.c
...
Compiling repos/apache-mynewt-core/util/mem/src/mem.c
...
Archiving util_mem.a
Linking [PATH]/bin/targets/first/app/apps/first/first.elf
Target successfully built: targets/first
```

Sign the Build with a Version Number

You can prepare the image to be flashed to the device with a mandatory version number and signing details via the `newt create-image` command:

```
$ newt create-image first 0.1.5
App image succesfully generated: [PATH]/bin/targets/first/app/apps/first/first.img
```

And finally you can flash the image via one of two methods:

Flash the Image via a Segger J-Link

With the J-Link connected to the nRF52, run the following command to flash the signed image:

This command assumes that you have the Segger J-Link drivers installed on your system, as described in the Native Installation pages.

```
$ newt load first
Loading app image into slot 1
```

Flash the Image via the Serial Port and newtmgr

For details on how to flash the image over newtmgr using only the serial port (no J-Link required), see [Uploading Application Images with newtmgr \(\)](#).

Adding Tasks

The official Mynewt 'Task' documentation is available at: https://mynewt.apache.org/latest/os/core_os/task/task.html

A task (`os_task`) in Mynewt is made up of a task handler function, a task 'stack' which provide the block of memory that will be used when executing the task, and a priority level.

Since Mynewt is a multitasking environment, tasks are also assigned a priority level, and at any given time the highest priority task will run. When the highest priority task stops (waiting for an event, or when delayed in code) the next highest priority task will

fire, and so on until the scheduler gets down to the lowest priority task, usually called 'idle' (which will be set by the kernel when the OS starts up).

Declaring a task, priority and stack size

In order to declare a task, you need to set the task's:

- priority
- stack size
- and the name of the task handler that will be run when the task is active.

The task's priority can be from 1..255, where the higher the number the lower the priority.

The stack size is in units of `os_stack_t`, which is usually a 32-bit value, meaning a stack size of 64 (as shown in the example below) is 256 bytes wide.

The task handler has the following signature: `void my_task_func(void *arg)`

```
/* Define task stack and task object */
#define MY_TASK_PRIO      (OS_TASK_PRI_HIGHEST)
#define MY_STACK_SIZE    OS_STACK_ALIGN(64)
struct os_task my_task;
os_stack_t my_task_stack[MY_STACK_SIZE];
```

Initializing a task

To initialize the task, you need to call the `sysinit()` function then add your task to the os via: `os_task_init`. This normally takes place in the main loop, or in a dedicated function called inside main like `init_tasks()`.

`os_task_init` has the following signature and parameters:

```
os_task_init(struct os_task *t, const char *name, os_task_func_t func,
             void *arg, uint8_t prio, os_time_t sanity_itvl,
             os_stack_t *stack_bottom, uint16_t stack_size)
```

- `struct os_task *t`: A pointer to the `os_task` to initialize
- `const char *name`: The public name to associate with this task, which will be visible in the shell, newtmgr, and other reporting systems.

- `os_task_func_t func` : The function to execute when this task is active, which will have the following signature: `void my_task_handler(void *arg)`
- `void *arg` : Optional arguments to pass into the task handler
- `uint8_t prio` : The priority level for the task, lower = higher priority
- `os_time_t sanity_itvl` : The time at which this task should check in with the sanity task. `OS_WAIT_FOREVER` means never check in.
- `os_stack_t *stack_bottom` : A pointer to the bottom of a task's stack.
- `uint16_t stack_size` : The size of the task's stack (in `os_stack_t` units), which are usually 32-bits.

The following examples initialises a task matching the values declared earlier in this document:

```

/* This is the main function for the project */
int main(void) {
    int rc;

    /* Initialize OS */
    os_init();

    /* Initialize the task */
    os_task_init(&my_task, "my_task", my_task_func, NULL, MY_TASK_PRIO,
                OS_WAIT_FOREVER, my_task_stack, MY_STACK_SIZE);

    /* Start the OS */
    os_start();

    /* os start should never return. If it does, this should be an error */
    assert(0);

    return rc;
}

```

Implementing the task handler

The last part of the system is the task handler, which will be called every time that the task is active (as determined by the scheduler).

Task handlers are infinite loops that have an initial setup face, and then normally a `while(1)` loop that runs forever as long as the task is active.

The following example initialises a GPIO pin as an output, setting the pin high. It then starts an infinite loop and toggles the LED every second, sleeping between 1s intervals so that other tasks can run:

Note that there are two parts to the task handler. An initial chunk of code that will be run once when the task is initialized, then a 'while (1)' loop that will be run in repetition whenever the task has focus in the scheduler.

```
static void
my_task_func(void *arg)
{
    hal_gpio_init_out(LED_BLINK_PIN, 1);

    while (1) {
        /* Wait one second */
        os_time_delay(OS_TICKS_PER_SEC * 1);

        /* Toggle the LED */
        hal_gpio_toggle(LED_BLINK_PIN);
    }
}
```

Task Delays

There are various ways that a task can be interrupted, such as delaying execution for a specific amount of time, waiting for an event on an event queue, waiting for a semaphore, etc.

Delaying your tasks is important, because as long as your task is active, no tasks of lower priority will execute. As such, it's important to manage your tasks as efficiently as possible to ensure that clock cycles are available for other tasks in the system.

`os_time_delay`

The `os_time_delay` function is the easiest way to cause a delay in execution in your task. Simply specify a specific number of ticks, and the scheduler will mark this task as inactive for the indicated delay.

Please note that `os_time_delay` uses system ticks, which may vary from one system to the next, so any delays should be based on the `OS_TICKS_PER_SECOND` macro to remain portable.

Example: Updating `apps/first/src/main.c`

A simple example of adding a blinky task handler to `apps/first` can be implemented by updating `apps/first/src/main.c` with the following code, and reflashing the firmware image:

```

#include <assert.h>;
#include <string.h>;

#include "os/os.h"
#include "bsp/bsp.h"
#include "hal/hal_gpio.h"
#include "sysinit/sysinit.h"

/* Define task stack and task object */
#define LED_TASK_PRI0      (100) /* 1 = highest, 255 = lowest */
#define LED_STACK_SIZE    OS_STACK_ALIGN(64)
struct os_task led_task;
os_stack_t led_task_stack[LED_STACK_SIZE];

static void led_task_func(void *arg);

int
main(int argc, char **argv)
{
    int rc;

    /* Initialize the task */
    os_task_init(&led_task, "blinky", led_task_func, NULL,
                LED_TASK_PRI0, OS_WAIT_FOREVER, led_task_stack,
                LED_STACK_SIZE);

    /* Initialize the OS */
    sysinit();

    while (1) {
        /* Run the event queue to process background events */
        os_eventq_run(os_eventq_dflt_get());
    }

    return rc;
}

static void
led_task_func(void *arg)
{
    /* Configure the LED GPIO as an output and HIGH (0n) */
    hal_gpio_init_out(LED_BLINK_PIN, 1);

    while (1) {
        /* Wait one second */
        os_time_delay(OS_TICKS_PER_SEC * 1);

        /* Toggle the LED */
        hal_gpio_toggle(LED_BLINK_PIN);
    }
}

```

You can vary the blinky rate by adjusting the delay in 'os_time_delay' in the task handler

Checking Task Status

You can verify the task status from the command line with the `newtmgr taskstat` command:

```
$ newtmgr -c serial1 taskstat
  task pri tid  runtime      csw   stksz   stkuse last_checkin next_checkin
  blinky 100  2      0         4     64     26      0         0
  idle 255  0     424        5     64     32      0         0
  main 127  1      0         3    1024     93      0         0
```

Adding Shell Commands

The official Shell documentation can be seen here: <https://mynewt.apache.org/latest/os/modules/shell/shell.html>

Mynewt includes a 'Shell' library that allows you to add an extensible command-line style interface to your applications. You can implement custom commands that users can execute over a simple serial connection, and display formatted text back to the end users.

Adding shell support

To add shell support to your app, make sure the following `pkg.deps` are defined in your `pkg.yml` file:

```
pkg.deps:
- "@apache-mynewt-core/sys/console/full"
- "@apache-mynewt-core/sys/shell"
- "@apache-mynewt-core/sys/sysinit"
```

And in the `syscfg.vals` section of `syscfg.yml` add the following:

```
syscfg.vals:
  # Enable the shell task.
  SHELL_TASK: 1
```

Adding a custom command handler

To add a new command handler use the following code snippets, placing the different sections at an appropriate place in your code:

```
#include "console/console.h"
#include "shell/shell.h"

...

// Command handler prototype declaration
static int shell_test_cmd(int argc, char **argv);
```

```

// Shell command struct
static struct shell_cmd shell_test_cmd_struct = {
    .sc_cmd = "test",
    .sc_cmd_func = shell_test_cmd
};

...

// Implement your command handler
static int
shell_test_cmd(int argc, char **argv)
{
    console_printf("Test!\n");
    return 0;
}

...

// Call this before sysinit to register the command
#if MYNEWT_VAL(SHELL_TASK)
    shell_cmd_register(&shell_test_cmd_struct);
#endif

```

Example: Updating apps/first/src/main.c

You can update `apps/first` with a custom shell command using the following code in main.c:

```

#include <assert.h>
#include <string.h>

#include "os/os.h"
#include "bsp/bsp.h"
#include "hal/hal_gpio.h"
#include "sysinit/sysinit.h"
#include "console/console.h"
#include "shell/shell.h"

/* Define task stack and task object */
#define LED_TASK_PRI0      (100) /* 1 = highest, 255 = lowest */
#define LED_STACK_SIZE    OS_STACK_ALIGN(64)
struct os_task led_task;
os_stack_t led_task_stack[LED_STACK_SIZE];

/* LED task handler prototype declaration */
static void led_task_func(void *arg);

/* Command handler prototype declaration */
static int shell_test_cmd(int argc, char **argv);

/* Shell command struct */
static struct shell_cmd shell_test_cmd_struct = {
    .sc_cmd = "test",
    .sc_cmd_func = shell_test_cmd
};

int
main(int argc, char **argv)
{
    int rc;

    /* Initialize the task */

```

```

os_task_init(&led_task, "blinky", led_task_func, NULL,
            LED_TASK_PRIO, OS_WAIT_FOREVER, led_task_stack,
            LED_STACK_SIZE);

/* Call this before sysinit to register the command */
#ifdef MYNEWT_VAL(SHELL_TASK)
    shell_cmd_register(&shell_test_cmd_struct);
#endif

/* Initialize the OS */
sysinit();

while (1) {
    /* Run the event queue to process background events */
    os_eventq_run(os_eventq_dflt_get());
}

return rc;
}

/* Implement the 'test' command handler */
static int
shell_test_cmd(int argc, char **argv)
{
    console_printf("Test!\n");
    return 0;
}

static void
led_task_func(void *arg)
{
    /* Configure the LED GPIO as an output and HIGH (0n) */
    hal_gpio_init_out(LED_BLINK_PIN, 1);

    while (1) {
        /* Wait one second */
        os_time_delay(OS_TICKS_PER_SEC * 1);

        /* Toggle the LED */
        hal_gpio_toggle(LED_BLINK_PIN);
    }
}

```

Testing the Shell Command

You can test the new 'test' shell command by connecting over serial:

```
$ minicom -D /dev/tty.SLAB_USBtoUART
```

Once connected you can send the 'help' command to get a list of valid shell commands, and then then 'test' command which should display 'Test!' in the shell:

```
Welcome to minicom 2.7

OPTIONS:
Compiled on Mar 17 2016, 21:54:43.
Port /dev/tty.SLAB_USBtoUART, 01:42:23

Press Meta-Z for help on special keys

[help
026488 help
026488 test
026489 stat
026490 config
026491 log
[026492 compat> test
026717 Test!
026717 compat> █
```

Adding Statistics

Full documentation on statistics is available here: <https://mynewt.apache.org/latest/os/modules/stats/stats.html>

Mynewt includes an internal statistics system where numeric values can be declared and incremented over time, then accessed as a debug and analysis tool.

For example, every time a specific sensor is accessed, you can increment the `sensor_reads` statistic value by one (assigning whatever variable name you want), and when a specific bug or condition occurs you can check the value to see whether or how many times the sensor was accessed before the specific condition occurs.

You can also use statistics to verify the number of bytes sent and received over a specific peripheral to make sure sensitive interrupt driven code is working well without having to stop the entire system.

Configuring Your App for Statistics

To enable statistics in your application, the following dependency needs to be present in `pkg.yml`:


```
pkg.deps:
  - "@apache-mynewt-core/sys/stats"
```

To save flash space internally, statistics are referenced by number by default. If you wish to preserve the full name of the individual statistics for easier debugging, then the following flag also needs to be added to the `syscfg.yml` file:

```
syscfg.vals:
  # Include names for statistics.
  STATS_NAMES: 1
```

Optionally, if you wish to access statistics from the shell interface, you can also enable the '`stat`' command with the following flag in `syscfg.yml`:

```
syscfg.vals:
  STATS_CLI: 1
```

Adding Stats to your main.c File:

Stats requires the following header be present:

```
#include <stats/stats.h>;
```

This will enable the use of the `STATS*` macros to define the stats layout, following the example below:

```
STATS_SECT_START(my_stat_section)
  STATS_SECT_ENTRY(attempt_stat)
  STATS_SECT_ENTRY(error_stat)
STATS_SECT_END
```

At compile time, this will resolve to the following structure:

```
struct stats_my_stat_section {
  struct stats_hdr_s_hdr;
  uint32_t sattempt_stat;
  uint32_t serror_stat;
};
```

You will also need to provide names for each field, regardless of WHETHER you have enabled naming support via `STATS_NAMES` or not:

Note that the field names need to match between the STATS_SECT_ENTRY name above and the STATS_NAME entry below!

```
/* Define a few stats for querying */
STATS_NAME_START(my_stat_section)
    STATS_NAME(my_stat_section, attempt_stat)
    STATS_NAME(my_stat_section, error_stat)
STATS_NAME_END(my_stat_section)
```

At compile time, this will resolve to the following structure:

```
struct stats_name_map g_stats_map_my_stat_section[] = {
    { __builtin_offsetof (struct stats_my_stat_section, sattempt_stat),
      "attempt_stat" },
    { __builtin_offsetof (struct stats_my_stat_section, serror_stat),
      "error_stat" },
};
```

Accessing the Stats in Your Code

You will need to declare a global variable somewhere to hold the stats data, using the model below:

```
STATS_SECT_DECL(my_stat_section) g_mystat;
```

If the global definition is in another file and you are referencing it elsewhere, you would declare this in the file where you will modify locally:

```
extern STATS_SECT_DECL(my_stat_section) g_mystat;
```

Initializing the Stats

Before your stats can be used or accessed, they need to be initialised and registered.

You can initialise your stats entry as follows:

The stats_init() and stats_register() function calls should occur AFTER the sysinit(); function!

```
rc = stats_init(
    STATS_HDR(g_mystat),
    STATS_SIZE_INIT_PARMS(g_mystat, STATS_SIZE_32),
```

```
    STATS_NAME_INIT_PARMS(my_stat_section));
assert(rc == 0);
```

For the stat size, you can use one of the following values:

- `STATS_SIZE_16` -- stats are 16 bits (wraps at 65536)
- `STATS_SIZE_32` -- stats are 32 bits (wraps at 4294967296)
- `STATS_SIZE_64` -- stats are 64-bits

You then need to register the stats entry so that you can access it, which is done via the following function call:

NOTE: The name below is the name that you will use when accessing the stats via the console or via the `newtmgr stat` command

```
rc = stats_register("my_stats", STATS_HDR(g_mystat));
assert(rc == 0);
```

Updating the Stats Values

Incrementing

To increment the stats values, you can use the `STATS_INC` or `STATS_INCN` macros, as shown below:

```
STATS_INC(g_mystat, attempt_stat);
rc = do_task();
if(rc == ERR) {
    STATS_INC(g_mystat, error_stat);
}
```

To increment the value a specific number use `STATIS_INCN`:

```
STATS_INCN(g_mystat, attempt_stat, 5);
```

Accessing Stats with the Console or `newtmgr`

Console Access

Assuming that you have enabled named access to stats via `STATS_NAME_ENABLE` you can access your stats from the console via:

```
stat my_stats
```

This will give you something resembling the following output:

```
12274:attempt_stat: 3  
12275:error_stat: 0
```

If you don't have names enabled via `STATS_NAME_ENABLE` you would see something like this:

```
stat my_stats  
29149:s0: 3  
29150:s1: 0
```

`newtmgr` Access

You can also access stats through `newtmgr` as follows:

```
$ newtmgr -c serial1 stat my_stats  
Return Code = 0  
Stats Name: my_stats  
  attempt_stat: 0  
  error_stat: 0
```

Example: Adding Stats to `apps/first/src/main.c`

You can extend `apps/first` with custom stats by updating `apps/first/src/main.c` with the following code:

```
#include <assert.h>  
#include <string.h>
```

```

#include "os/os.h"
#include "bsp/bsp.h"
#include "hal/hal_gpio.h"
#include "sysinit/sysinit.h"
#include "console/console.h"
#include "shell/shell.h"
#include "stats/stats.h"

/* Define task stack and task object */
#define LED_TASK_PRI0      (100) /* 1 = highest, 255 = lowest */
#define LED_STACK_SIZE    OS_STACK_ALIGN(64)
struct os_task led_task;
os_stack_t led_task_stack[LED_STACK_SIZE];

/* LED task handler prototype declaration */
static void led_task_func(void *arg);

/* Command handler prototype declaration */
static int shell_test_cmd(int argc, char **argv);

/* Shell command struct */
static struct shell_cmd shell_test_cmd_struct = {
    .sc_cmd = "test",
    .sc_cmd_func = shell_test_cmd
};

/* Define a custom stats group for querying */
STATS_SECT_START(led_stat_section)
    STATS_SECT_ENTRY(led_toggles)
STATS_SECT_END

/* Define a few stat name fields for querying */
STATS_NAME_START(led_stat_section)
    STATS_NAME(led_stat_section, led_toggles)
STATS_NAME_END(led_stat_section)

/* Add the global variable to access and increment stats */
STATS_SECT_DECL(led_stat_section) g_ledstats;

int
main(int argc, char **argv)
{
    int rc;

    /* Initialize the task */
    os_task_init(&led_task, "blinky", led_task_func, NULL,
                LED_TASK_PRI0, OS_WAIT_FOREVER, led_task_stack,
                LED_STACK_SIZE);

    /* Call this before sysinit to register the command */
    #if MYNEWT_VAL(SHELL_TASK)
        shell_cmd_register(&shell_test_cmd_struct);
    #endif

    /* Initialize the OS */
    sysinit();

    /* Initialise the custom stats section */
    rc = stats_init(
        STATS_HDR(g_ledstats),
        STATS_SIZE_INIT_PARMS(g_ledstats, STATS_SIZE_32),
        STATS_NAME_INIT_PARMS(led_stat_section));
    assert(rc == 0);

    /* Then register the custom section with the stats system */
    rc = stats_register("led_stats", STATS_HDR(g_ledstats));
    assert(rc == 0);

    while (1) {

```

```

        /* Run the event queue to process background events */
        os_eventq_run(os_eventq_dflt_get());
    }

    return rc;
}

/* Implement the 'test' command handler */
static int
shell_test_cmd(int argc, char **argv)
{
    console_printf("Test!\n");
    return 0;
}

static void
led_task_func(void *arg)
{
    /* Configure the LED GPIO as an output and HIGH (0n) */
    hal_gpio_init_out(LED_BLINK_PIN, 1);

    while (1) {
        /* Wait one second */
        os_time_delay(OS_TICKS_PER_SEC * 1);

        /* Toggle the LED */
        hal_gpio_toggle(LED_BLINK_PIN);

        /* Increment the LED stat */
        STATS_INC(g_ledstats, led_toggles);
    }
}

```

Monitoring via `netwmgr`

You can monitor the `led_toggles` value via `newtmgr` with the following command (where we can see that the LED had been toggled 11 times to far in this case):

```

$ newtmgr -c serial1 stat led_stats
stat group: led_stats
          11 led_toggles

```

Monitoring via shell/console

Alternatively, you can connect to the shell (for example, via: `$ minicom -D /dev/tty.USBtoUART`) and run the `stat led_stats` command to get the same value(s):

```
myproject — minicom -D /de
Welcome to minicom 2.7

OPTIONS:
Compiled on Mar 17 2016, 21:54:43.
Port /dev/tty.SLAB_USBtoUART, 10:46:49

Press Meta-Z for help on special keys

[stat stat
030422 num_registered: 2
[030422 compat> stat
030775 Must specify a statistic name to dump, possible names are:
030777 stat
030777 led_stats
[030778 compat> stat led_stats
031420 led_toggles: 245
031420 compat> █
```

Adding BLE UART Support

The easiest way to add BLE UART support is using the [bleuart helper module \(\)](#) that is part of [Adafruit_Mynewt \(\)](#), an in house Mynewt project that we created during product development.

If you wish to test a complete application that shows how to use BLE UART, you should start with the [bleuart \(\)](#) example application, which exposes shell commands to work with BLE UART, as well as tracks the TX and RX byte count via statistics, and works well with the Adafruit Mynewt Manager mobile application.

[Browse the bleuart example application code on Github](#)

This section will be expanded in the near future with details on working with BLE and BLE UART in Mynewt, but at the moment we only have example code that we've written to get you started.

Mynewt Nimble (BLE Stack) Documentation

Mynewt's open source Bluetooth Low Energy stack is called nimble. Full documentation on number can be found via the link below:

[View Mynewt nimble documentation](#)

Advanced Debugging

The pages in this section are for advanced users who need to debug remote or complex systems, and are aimed at giving you a head start solving specific problems:

- [GDB Debugging \(\)](#): General overview of common GDB debugging commands and suggestions on how to debug common problems via GDB
- [Command Line Debugging \(\)](#): Some common tricks to help debug issues from the command line
- [Field Debugging Tips \(\)](#): Advice on debugging crash dumps from devices in the field

GDB Debugging

GDB debugging will require a Segger JLink for the GDB Server and debug port access

The definitive guide to GDB is available here:

ftp://ftp.gnu.org/old-gnu/Manuals/gdb/html_chapter/gdb_toc.html

Starting the Debugger

You can start GDB (GNU Debugger) with newt with the following command, with a JLink connected to the target device:

```
# Optionally build and flash the image
$ newt build target_name
$ newt create-image target_name 0.0.0
$ newt load target_name

# Start GDB
$ newt debug target_name
```

You can then start and stop code execution on the target MCU via:

```
(gdb) monitor halt
```


and

```
(gdb) monitor go
```

You can also start a fresh run of the code via:

```
(gdb) monitor reset  
(gdb) c
```

You can check if the OS is running by executing the following code, which will display the OS time counter:

```
(gdb) p/d g_os_time
```

Displaying Values

To display the current state of a struct or global variable, use the `print [name]` (or `p [name]`) command.

Basic Example

The example below halts the processor and then prints a struct instance named `ble_phy_stats`.

> Tip: You can optionally enable pretty printing via `set print pretty on`

```
(gdb) monitor halt  
(gdb) set print pretty on  
(gdb) print ble_phy_stats  
$5 = {  
  s_hdr = {  
    s_name = 0x0,  
    s_size = 0 '\000',  
    s_cnt = 0 '\000',  
    s_pad1 = 0,  
    s_next = {  
      stqe_next = 0x0  
    }  
  },  
  sphy_isrs = 0,  
  stx_good = 1,  
}
```

```

stx_fail = 0,
stx_late = 0,
stx_bytes = 27,
srx_starts = 0,
srx_aborts = 0,
srx_valid = 0,
srx_crc_err = 0,
srx_late = 0,
sno_bufs = 0,
sradio_state_errs = 0,
srx_hw_err = 0,
stx_hw_err = 0
}

```

Formatting Display Values

You can also format the printed values with the following formatters:

- **x** Regard as integer and display as hexadecimal
- **d** Regard as integer and display as signed decimal
- **u** Regard as integer and display as unsigned decimal
- **c** Regard as integer and print as a char constant
- **f** Regard as float and print in floating point syntax
- **t** Print integer as binary
- **a** Print as an address (hex plus offset). Useful to discover where an address is located (ex. `p/a 0x12345` yields `$3 = 0x12345 <_initialize_vx+396>`)

To print the BLE link layer stack (`g_ble_ll_stack`) in hex enter:

```

(gdb) p/x g_ble_ll_stack
$17 = {0xdeadbeef <repeats 22=" times=">, 0x20002568, 0x304, 0xe000e100, 0x100,
0x20001be4, 0x0, 0xffffffff, 0x0,
0xffffffff, 0x20002204, 0x19f14, 0x20002218, 0x0, 0x20001e90, 0x10000000,
0x20002180, 0x354, 0xa0a3, 0x92b2,
0x61000000, 0x20001e8c, 0x200021d8, 0x0, 0x9657, 0x4, 0xffffffff, 0xffffffff,
0x1fff8000, 0x0, 0xa897, 0x0, 0xa85d,
0x1fff8000, 0xffffffff, 0xffffffff, 0x1fff8000, 0x0, 0x0, 0x8, 0xde, 0x93c9, 0x0}

```

Displaying an Array of Values

You can display the contents of an array as follows:

```
(gdb) monitor halt
(gdb) set print pretty on
(gdb) print *array@len
```

Useful Mynewt/Nimble Structs and Fields

Some useful Mynewt or nimble fields to display can be seen below:

- `ble_phy_stats` - PHY stats for traffic tracking
- `ble_ll_stats` - Link layer stats
- `ble_ll_conn_stats` - Connection stats
- `g_ble_ll_adv_sm` - Advertising state machine
- `g_ble_ll_stack` - Link layer stack

For example:

```
(gdb) monitor halt
(gdb) set print pretty on
(gdb) p ble_phy_stats
$16 = {
  s_hdr = {
    s_name = 0x0,
    s_size = 0 '\000',
    s_cnt = 0 '\000',
    s_pad1 = 0,
    s_next = {
      stqe_next = 0x0
    }
  },
  sphy_isrs = 0,
  stx_good = 1,
  stx_fail = 0,
  stx_late = 0,
  stx_bytes = 27,
  srx_starts = 0,
  srx_aborts = 0,
  srx_valid = 0,
  srx_crc_err = 0,
  srx_late = 0,
  sno_bufs = 0,
  sradio_state_errs = 0,
  srx_hw_err = 0,
  stx_hw_err = 0
}
```

Memory Manipulation

You can display the memory contents of a specific address via the `x` command.

To see the main stack pointer location on an ARM chip, for example, run:

```
(gdb) x 0
0x0: 0x20008000
```

You can also adjust the output size with the optional `x/nfu` flags:

- `n` Indicates how much memory to display (in `u` units), default = 1
- `f` Indicates the display format, where:
 - `s` means null-terminated string
 - `i` means machine instruction
 - `x` Display as hexadecimal (default)
 - `d` Display as signed decimal
 - `u` Display as unsigned decimal
 - `c` Print as a char constant
 - `f` Print in floating point syntax
 - `t` Print integer as binary
- `u` The unit size, which can be:
 - `b` Bytes
 - `h` Halfwords (two bytes)
 - `w` Words (four bytes)
 - `g` Giant worlds (eight bytes)

> Note: Whenever you change the unit size (`u`), the updated value becomes
> the system default and will be retained on future requests until it is
> changed again.

For example, to display the same 32-bit stack pointer at address 0 on an ARM processor as four individual bytes displayed as unsigned decimal values you would enter:

```
(gdb) x/4ub 0
0x0: 0 128 0 32
```

To display the 32-bit value in binary notation you would enter:

```
(gdb) x/4tb 0
0x0: 00000000 10000000 00000000 00100000
```

To display 4 'words' as 32-bit values in hex notation:

```
(gdb) x/4xw 0
0x0: 0x20008000 0x00003049 0x00003099 0x00003099
```

Stack Manipulation

The following commands can be used to work with the stack such as producing a dump of the stack frames, or checking for stack overflow.

Display the Stack Trace

You can display a list of function calls up to the point where the MCU halted with the `backtrace` or `bt` command, which will dump individual stack frame records:

```
(gdb) bt
#0  os_tick_idle (ticks=131072) at hal_os_tick.c:146
#1  0x000091f6 in os_idle_task (arg=<optimized out="">) at os.c:64
#2  0x00000000 in ?? ()
```

Each line shows the frame number, and the function name and return address. In this case, the code has stopped at `os_tick_idle` in `halostick.c`, which was called from `os_idle_task` in `os.c`.

Display Stack Frame Details

You can display detailed information about a specific stack frame via the `info frame [n]` command:

```
(gdb) info frame
Stack level 0, frame at 0x20001e60:
pc = 0x184aa in os_tick_idle (hal_os_tick.c:146); saved pc = 0x91f6
```

```
called by frame at 0x20001e80
source language c.
Arglist at 0x20001e40, args: ticks=131072
Locals at 0x20001e40, Previous frame's sp is 0x20001e60
Saved registers:
  r3 at 0x20001e48, r4 at 0x20001e4c, r5 at 0x20001e50, r6 at 0x20001e54, r7 at
0x20001e58, r8 at 0x20001e40,
  r9 at 0x20001e44, lr at 0x20001e5c
```

To display the arguments for the current stack frame you can run:

```
(gdb) info args
ticks = 131072
```

To display the local variables (one per line) for the stack frame run (data may or may not be available depending on build settings):

```
(gdb) info locals
ocmp = <optimized out="">
```

Displaying ARM Registers

You can also display a list of the ARM registers via `info registers`.

The following example shows the same `pc` value seen above where we are halted at 0x184aa on `os_tick_idle`, and the stack pointer (`sp`) is at 0x20001e40, one 32 byte (0x20) stack frame away from the value seen earlier.

```
(gdb) info registers
r0          0x800000  8388608
r1          0x4000b000 1073786880
r2          0xb4a5e0  11838944
r3          0x3ffffef 4194287
r4          0xffffffff 4294967295
r5          0xffffffff 16777215
r6          0x504    1284
r7          0x540    1344
r8          0xff800000 4286578688
r9          0x20000   131072
r10         0x1fff8000  536838144
r11         0x0     0
r12         0x20    32
sp          0x20001e40 0x20001e40 <g_idle_task_stack+192>
lr          0x18435   99381
pc          0x184aa 0x184aa <os_tick_idle+182>
```

```
xpsr      0x21000000  553648128
MSP       0x20008000  536903680
PSP       0x20001e40  536878656
PRIMASK   0x1  1
BASEPRI   0x0  0
FAULTMASK 0x0  0
CONTROL   0x2  2
```

Working with Source Code

You can display source code inside GDB with some basic commands.

Displaying a function

This will display the first 10 lines of the specified function:

```
(gdb) list os_idle_task
39 #endif
40 #define MAX_IDLE_TICKS (600 * OS_TICKS_PER_SEC) /* 10 minutes */
41
42 void
43 os_idle_task(void *arg)
44 {
45     os_sr_t sr;
46     os_time_t now;
47     os_time_t iticks, sticks, cticks;
48
```

To display the next 10 lines enter:

```
(gdb) list
49     /* For now, idle task simply increments a counter to show it is running. */
50     while (1) {
51         ++g_os_idle_ctr;
52         OS_ENTER_CRITICAL(sr);
53         now = os_time_get();
54         sticks = os_sched_wakeup_ticks(now);
55         cticks = os_callout_wakeup_ticks(now);
56         iticks = min(sticks, cticks);
57         if (iticks < MIN_IDLE_TICKS) {
58             iticks = 0;
```

> Tip: You can adjust the number of lines displayed via `set listsize [count]`

Displaying code at a specific address

If you wish to display the code at a specific address you can prefix the address with a `*` character as shown below:

```
(gdb) list *0x00009e54
0x9e54 is in crash_device (crash_test.c:46).
41     if (!strcmp(how, "div0")) {
42
43         val1 = 42;
44         val2 = 0;
45
46         val3 = val1 / val2;
47         console_printf("42/0 = %d\n", val3);
48     } else if (!strcmp(how, "jump0")) {
49         ((void (*)(void))0)();
50     } else if (!strcmp(how, "ref0")) {
```

Running an arbitrary function when halted at a breakpoint

When halted at a breakpoint, you can run a function via the `call` command. Tip via [Håkon Alseth](#).

> Make sure to include the parenthesis after the function name when issuing the `call` command, which will cause the device to go back to the halt state once the function has completed execution.

Connect to the target using `$ newt debug target_name` or via some variation of the following code:

```
arm-none-eabi-gdb _build/*.out
(gdb) target remote :2331
(gdb) load
```

Then run until you hit the BP, and at an appropriate moment execute the `call` command with your target function name:

```
(gdb) mon reset
(gdb) c
<break somewhere="" in="" your="" code,="" optionally="" using="" ctrl+c="">
(gdb) call test_function()
```

Command Line Debugging

The following tips and tricks can be used when debugging from the command line.

Grep'ing Source Code

The `grep` tool allows you to efficiently find references to strings in the current folder or recursively, such as looking for every reference to a specific function name or variable in your codebase.

Grep recursively for a partial string

To perform a recursive search for a partial string enter:

```
$ grep -rn "./" -e "SEARCH_PATTERN"
```

- `r` means recursive
- `n` means show line numbers
- `e` is the string to search for (`SEARCH_PATTERN` in this case)

This will return a reference to any instance starting with `SEARCH_PATTERN`, including the specific line number.

> NOTE: This search is case-sensitive. Adding the `i` flag will make the search case insensitive.

Grep recursively for an exact string

To perform a recursive search for an exact match of an entire string enter:

```
$ grep -rnw "./" -e "SEARCH_PATTERN"
```

- `w` means whole word

This will return a reference to any instances of `SEARCH_PATTERN`.

Grep recursively for a string ignoring the case

If you wish the search to be case insensitive you can also add the `-i` argument, as shown below:

```
$ grep -rni "./" -e "SeArCh_PaTtErN"
```

- `i` means case insensitive (default is case sensitive searching)

This would return 'SearchPattern' or 'SEARCHPATTERN' as valid matches.

Grep recursively with specific file types

If you wish to restrict your search to a specific file type, you can use the `--include` flag as follows:

```
$ grep --include=*\.{c,h} -rnw "./" -e "SEARCH_PATTERN"
```

This will only search files ending in `.c` or `.h`

You can also exclude certain file types with the `--exclude` flag:

```
$ grep --exclude=*.o -rnw "./" -e "SEARCH_PATTERN"
```

This will exclude all files ending in `.o` from the search.

Field Debugging Tips

Debugging devices deployed remotely is always a challenge but this documentation gives some basic techniques to help debug issues remotely.

Debugging Crash Dumps

If a task crashes (enter the `fault_handler` etc.) it will normally generate a simple dump of the system registers, which can be useful to find the cause of the problem if you have the matching `.elf` file locally.

This example uses the `crash_test` library from `apache-mynewt-core` to simulate various crash events locally via the `crash` command from shell:

```
crash div0
6976:Unhandled interrupt (3), exception sp 0x2000a960
6976: r0:0x00000000 r1:0x000242bd r2:0x00000000 r3:0x0000002a
6976: r4:0x00000000 r5:0x2000c002 r6:0x2000bffc r7:0x00025e34
6976: r8:0x0000d04d r9:0x0000000c r10:0x20009068 r11:0x55555556
6976:r12:0x00000000 lr:0x00009e45 pc:0x00009e54 psr:0x61000000
6976:ICSR:0x00419803 HFSR:0x40000000 CFSR:0x02000000
6976:BFAR:0xe000ed38 MMFAR:0xe000ed34
```

In the example above we see the output of a divide by zero crash.

The important register value is `pc`. Make a note of this address since you will use it in one of the debugging methods described below:

Option 1: Debugging Crash Dumps with GDB

If you have access to the matching app revision, build the target and deploy it to the device under test:

```
$ newt build throughput
$ newt create-image throughput 0.1.0
$ newt load throughput
```

To start GDB via the following command:

```
$ newt debug throughput
```

Please note that you will require a Segger J-Link to run the 'newt debug' command!

This will start up the GDB server and connect to the device under test.

See [GDB Debugging \(\)](#) for details on using GDB.

Run the following commands from the GDB shell:

```
(gdb) monitor go
(gdb) list *0x00009e54
0x9e54 is in crash_device (crash_test.c:46).
41     if (!strcmp(how, "div0")) {
42
43         val1 = 42;
44         val2 = 0;
45
46         val3 = val1 / val2;
47         console_printf("42/0 = %d\n", val3);
48     } else if (!strcmp(how, "jump0")) {
49         ((void (*)(void))0)();
50     } else if (!strcmp(how, "ref0")) {
```

You can see here that line 46 of crash_test.c caused the fault, which is where the divide by zero error occurs.

Option 2: Debugging Crash Dumps with `objdump`

If you have the .elf file but can't use GDB debugger you can see the code for the specified address from the command line using the `objdump` tool that is part of GCC.

From the command-line (with GCC available as part of the system path) run the following command:

Note: You must specify a `--stop-address` that is higher than the `--start-address` with this command, but you can increment the hex value by 1 byte to return only the line of code that caused the crash. You can play with the start and stop addresses to provide some context to the error.

```
$ arm-none-eabi-objdump -S --start-address=0x00009e54 --stop-address=0x00009e55 bin/
targets/throughput/app/apps/throughput/throughput.elf

bin/targets/throughput/app/apps/throughput/throughput.elf:      file format elf32-
littlearm

Disassembly of section .text:

00009e54 <crash_device+0x1c>:
    if (!strcmp(how, "div0")) {

        val1 = 42;
        val2 = 0;

        val3 = val1 / val2;
9e54:  fb93 f3f2  sdivr3, r3, r2
```

Debugging Repeatable Crashes

If you can repeat the crash scenario, you can find out the cause with the following sequence of events and two terminal windows:

Run GDB with the following command:

```
$ newt run &lt;target-name&gt; 0
```

When GDB comes up type `c` and press `enter`.

In a different shell, proceed to do whatever is required to cause the device to crash (such as sending a shell command, `newtmgr cmd`, etc..

If the device crashes, GDB should indicate such. You can then type the following commands in GDB to find out where is crashed:

```
(gdb) bt
(gdb) p *g_current_task
```

Adafruit_Mynewt

[Adafruit_Mynewt \(\)](#) is a sample Mynewt project that we use internally for various projects and experiments.

It isn't necessary to use this with the Adafruit Feather nRF52 Pro, but the repository contains a number of helper modules and apps, and can help you get started working with things like BLE UART and general Bluetooth Low Energy communication.

[View Adafruit_Mynewt on Github](#)

Installing Adafruit_Mynewt

To install and use the repo locally, first create a git clone of it using your favorite git IDE or from the command-line via:

```
$ git clone git@github.com:adafruit/Adafruit_Mynewt.git
```

Once the repository is installed locally, enter the Adafruit_Mynewt directory and install the Mynewt dependencies and libraries via:

```
$ newt install -v
```

NOTE: You only need to run this command once, since the mynewt core libraries are not part of the Adafruit_Mynewt repo itself and will be downloaded into the '/repos' folder by 'newt install'

Provided Apps

Not all of the apps included in the `/apps` folder are useful to everyone, but the following may serve as a good starting point for your own applications of learning requirements:

- [bleuart \(\)](#): This example initialises the BLE stack, and includes a BLE UART service and shell commands and statistics support to interact with the BLE UART service. This is the application that ships with our boards off the assembly line.
- [throughput \(\)](#): This example can be used to test the throughput speed of BLE using BLE UART. It includes a shell command ('`nustest`') to start and configure the throughput tests.

Helper Modules

The following modules are defined as part of Adafruit_Mynewt to make certain tasks easier to implement:

- [adautil \(\)](#): This module includes a number of helpers to facilitate working with UUIDs, config settings, data logs, timer timeouts, etc.
- [bledis \(\)](#): This module facilitates working with the Device Information Service (dis)
- [bleuart \(\)](#): This module facilitates working with the Nordic UART Service (nusa AKA 'BLE UART')
- [fifo \(\)](#): This module provides a First In First Out (FIFO) buffer implementation that plays well with the internal scheduler and RTOS architecture

External Content

The following external content may be useful understanding some of the many features available in Apache Mynewt:

Mynewt Bootloader

Build and Package Management

Downloads

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

- [Adafruit_nRF52_Arduino \(\)](#): The core code for this device (hosted on Github)
- [nRF52 Example Sketches \(\)](#): Browse the example code from the core repo on Github
- [nRF52832 Product Specification v1.4 \(\)](#): Key technical documentation for the nRF52832 SoC
- [EagleCAD PCB files on GitHub \(\)](#)

Feather Bluefruit NRF52 Pinout
Diagram

MyNewt Bootloader

This is if you want to program in MyNewt support - the bootloader will not work with Arduino!

feather52_boot_bleuart.bin

You can use adalink and a jlink to program this in via SWD to address 0 with something like:

```
adalink -v nrf52832 -p jlink -b feather52_boot_bleuart.bin 0
```

(Don't forget you may have to erase the chip)

Module Details

The Bluefruit nRF52 Feather uses the MDBT42Q module from Raytac. Details on the module, including FCC and other certifications are available in the document below:

MDBT42Q-Version_B.pdf

Schematic

