



Mynewt Documentation

Release master

The Apache Software Foundation

Sep 26, 2024

CONTENTS

1	Introduction	1
1.1	Welcome to Apache Mynewt	1
1.2	Newt	2
1.3	Newt Manager	2
1.4	Build your first Mynewt App with Newt	2
2	Setup & Get Started	3
2.1	Native Installation	3
2.1.1	Installing Native Toolchain	4
2.1.2	Installing the Cross Tools for ARM	6
2.2	Everything You Need in a Docker Container	10
2.2.1	Install Docker	10
2.2.2	Pull the Mynewt Container	11
2.2.3	Use a newt wrapper script	11
2.2.4	Enable USB2 Support for Mac or Windows	11
2.3	Creating Your First Mynewt Project	13
2.3.1	Prerequisites	13
2.3.2	Creating a New Project and Fetching the Source Repository	14
2.3.3	Testing the Project Packages	18
2.3.4	Building and Running the Simulated Blinky Application	20
2.3.5	Exploring other Mynewt OS Features	21
2.4	Using the Serial Port with Mynewt OS	21
2.4.1	Using the USB port on the nRF52DK	22
2.4.2	Using a USB to Serial Breakout Board	22
2.5	Debugging Mynewt	27
2.5.1	Prerequisites	27
2.5.2	Starting the Debugger	27
3	Concepts	29
3.1	Project	29
3.2	Package	30
3.3	Target	31
3.4	Configuration	32
4	Tutorials	33
4.1	Prerequisites	33
4.2	Tutorial categories	33
4.3	Project Blinky	34
4.3.1	Objective	35
4.3.2	Available Tutorials	35

4.3.3	Prerequisites	35
4.3.4	Overview of Steps	35
4.3.5	Blinky, your “Hello World!”, on Arduino Zero	36
4.3.6	Blinky, your “Hello World!”, on Arduino Primo	43
4.3.7	Blinky, your “Hello World!”, on Olimex	49
4.3.8	Blinky, your “Hello World!”, on a nRF52 Development Kit	54
4.3.9	Blinky, your “Hello World!”, on a PineTime smartwatch	58
4.3.10	Blinky, your “Hello World!”, on RedBear Nano 2	61
4.3.11	Blinky, your “Hello World!”, on STM32F4-Discovery	65
4.3.12	Blinky, your “Hello World!”, on STM32F303 Discovery	70
4.3.13	Pin Wheel Modifications to “Blinky” on STM32F3 Discovery	73
4.3.14	Enabling The Console and Shell for Blinky	75
4.4	Working with repositories	79
4.4.1	Adding Repositories to your Project	79
4.4.2	Create a Repo out of a Project	83
4.4.3	Accessing a private repository	86
4.4.4	Upgrade a repo	87
4.5	Project Slinky	87
4.5.1	Available Tutorials	87
4.5.2	Prerequisites	87
4.5.3	Overview of Steps	88
4.5.4	Project Sim Slinky	88
4.5.5	Project Slinky using the Nordic nRF52 Board	90
4.5.6	Project Slinky Using Olimex Board	94
4.6	Bluetooth Low Energy	100
4.6.1	Set up a bare bones NimBLE application	100
4.6.2	BLE iBeacon	104
4.6.3	BLE Eddystone	109
4.6.4	BLE Peripheral Project	115
4.6.5	Use HCI access to NimBLE controller	135
4.7	LoRaWAN App	139
4.7.1	Objective	139
4.7.2	Hardware needed	139
4.7.3	Create a project	139
4.7.4	Install Everything	139
4.7.5	Create the targets	140
4.7.6	Build the target executables	140
4.7.7	Sign and create the application image	141
4.7.8	Connect the board	141
4.7.9	Download bootloader and application	141
4.7.10	OTA Join	143
4.7.11	Opening/closing an application port	144
4.7.12	Sending data	144
4.8	OS Fundamentals	145
4.8.1	Events and Event Queues	145
4.8.2	Tasks and Priority Management	154
4.9	Remote Device Management	160
4.9.1	Enabling Newt Manager in Your Application	160
4.9.2	Over-the-Air Image Upgrade	165
4.10	Sensors	169
4.10.1	Sensor Tutorials Overview	169
4.10.2	Available Tutorials	169
4.10.3	Mynewt Smart Device Controller OIC App	170
4.10.4	Prerequisites	170

4.10.5	Sensor Framework	170
4.10.6	Air Quality Sensor Project	208
4.10.7	Adding an Analog Sensor on nRF52	229
4.10.8	Enabling and Calibrating Off-Board DRV2605 LRA Actuator	246
4.11	Tooling	250
4.11.1	SEGGER RTT Console	250
4.11.2	SEGGER SystemView	252
4.11.3	Diagnosing Errors	255
4.12	Other	264
4.12.1	How to Reduce Application Code Size	264
4.12.2	Write a Test Suite for a Package	265
4.12.3	Enable Wi-Fi on Arduino MKR1000	271
4.12.4	Charge control on PineTime	278
4.12.5	Rust Blinky application	284
5	Third-party Resources	289
5.1	Helpful Tutorials and Tools	289
6	OS User Guide	291
6.1	Kernel	291
6.1.1	Apache Mynewt Operating System Kernel	291
6.1.2	Scheduler	295
6.1.3	Task	297
6.1.4	Mutex	304
6.1.5	Semaphore	307
6.1.6	Event Queues	308
6.1.7	Callout	312
6.1.8	Heap	314
6.1.9	Memory Pools	315
6.1.10	Mbufs	321
6.1.11	CPU Time	339
6.1.12	OS Time	342
6.1.13	Sanity	348
6.2	System Modules	352
6.2.1	Config	352
6.2.2	Logging	362
6.2.3	Statistics Module	376
6.2.4	Console	386
6.2.5	Shell	393
6.3	Hardware Abstraction Layer	400
6.3.1	Description	400
6.3.2	General design principles	401
6.3.3	Example	401
6.3.4	Dependency	401
6.3.5	Platform Support	402
6.3.6	Timer	402
6.3.7	GPIO	405
6.3.8	UART	408
6.3.9	SPI	411
6.3.10	I2C	416
6.3.11	Flash	421
6.3.12	System	424
6.3.13	Watchdog	426
6.3.14	BSP	426

6.3.15	OS Tick	428
6.3.16	Creating New HAL Interfaces	429
6.3.17	Using HAL in Your Libraries	429
6.4	Bootloader	429
6.4.1	Limitations	430
6.4.2	Image Format	431
6.4.3	Flash Map	432
6.4.4	Image Slots	432
6.4.5	Boot States	432
6.4.6	Boot Vector	433
6.4.7	High-level Operation	435
6.4.8	Image Swapping	436
6.4.9	Swap Status	437
6.4.10	Reset Recovery	438
6.4.11	Integrity Check	439
6.4.12	Image Signing and Verification	439
6.5	Split Images	439
6.5.1	Description	439
6.5.2	Concept	439
6.5.3	Tutorial	440
6.5.4	Image Management	444
6.5.5	Syscfg	446
6.5.6	Loaders	446
6.5.7	Split Apps	446
6.5.8	Theory of Operation	446
6.6	Porting Mynewt OS	447
6.6.1	Description	447
6.6.2	Board Support Package (BSP) Dependency	447
6.6.3	MCU Dependency	448
6.6.4	MCU HAL	448
6.6.5	CPU Core Dependency	448
6.6.6	BSP Porting	448
6.6.7	Porting Mynewt to a new MCU	456
6.6.8	Porting Mynewt to a new CPU Architecture	457
6.7	Baselibc	457
6.7.1	Description	458
6.7.2	How to switch to baselibc	458
6.7.3	List of Functions	458
6.8	Drivers	458
6.8.1	Description	458
6.8.2	General design principles	459
6.8.3	Example	460
6.8.4	Implemented drivers	460
6.8.5	flash	460
6.8.6	mmc	462
6.8.7	Charge control drivers	464
6.9	Newt Manager	467
6.9.1	Invoking Newt Manager commands	467
6.9.2	newtmgr	468
6.9.3	Using the OIC Framework	468
6.9.4	Customizing Newt Manager Usage with mgmt	469
6.10	MCU Manager (mcumgr)	470
6.11	Image Manager	470
6.11.1	Description	470

6.11.2	List of Functions	470
6.11.3	imgmgr_module_init	471
6.11.4	imgr_ver_parse	471
6.11.5	imgr_ver_str	472
6.12	Compile-Time Configuration	472
6.12.1	System Configuration Setting Definitions and Values	473
6.12.2	Overriding System Configuration Setting Values	477
6.12.3	Conditional Settings	479
6.12.4	Generated syscfg.h and Referencing System Configuration Settings	479
6.12.5	Validation and Error Messages	481
6.13	System Initialization and System Shutdown	489
6.13.1	System Initialization	489
6.13.2	System Shutdown	492
6.14	Build-Time Hooks	493
6.14.1	Example	494
6.14.2	Custom Build Inputs	494
6.14.3	Details	495
6.15	File System Abstraction	496
6.15.1	Description	496
6.15.2	Support for multiple filesystems	497
6.15.3	Thread Safety	497
6.15.4	Header Files	498
6.15.5	Data Structures	498
6.15.6	Examples	498
6.15.7	API	501
6.15.8	Newtron Flash Filesystem (nffs)	503
6.15.9	The FAT File System	506
6.15.10	Other File Systems	507
6.15.11	Adding a new file system	508
6.16	Flash Circular Buffer (FCB)	510
6.16.1	Description	510
6.16.2	Usage	510
6.16.3	Data structures	511
6.16.4	API	512
6.16.5	fcb_append	514
6.16.6	fcb_append_finish	515
6.16.7	fcb_append_to_scratch	516
6.16.8	fcb_clear	516
6.16.9	fcb_getnext	517
6.16.10	fcb_init	517
6.16.11	fcb_is_empty	518
6.16.12	fcb_offset_last_n	518
6.16.13	fcb_rotate	519
6.16.14	fcb_walk	519
6.17	Mynewt Sensor Framework Overview	520
6.17.1	Overview of Sensor Support Packages	521
6.17.2	Sensor API	522
6.17.3	Sensor Manager API	537
6.17.4	Sensor Listener API	542
6.17.5	Sensor Notifier API	544
6.17.6	OIC Sensor Support	544
6.17.7	Sensor Shell Command	545
6.17.8	Sensor Device Driver	545
6.17.9	Creating and Configuring a Sensor Device	552

6.18	testutil	556
6.18.1	Description	556
6.18.2	Example	557
6.18.3	Data structures	557
6.18.4	API	558
6.19	JSON	561
6.19.1	Description	561
6.19.2	Data structures	561
6.19.3	API	563
6.20	Manufacturing Support	571
6.20.1	Description	571
6.20.2	Definitions	571
6.20.3	Details	572
6.20.4	Manufacturing ID	572
6.20.5	MMR Structure	572
6.21	Board support	576
6.21.1	PineTime smartwatch	576
7	BLE User Guide	579
7.1	Features	579
7.2	Bluetooth Mesh features	580
7.3	Components	580
7.4	Example NimBLE projects	581
7.5	NimBLE Security	581
7.5.1	Key Generation	581
7.5.2	Privacy Feature	582
7.6	NimBLE Setup	582
7.6.1	Configure clock for controller	582
7.6.2	Configure device address	583
7.6.3	Respond to <i>sync</i> and <i>reset</i> events	584
7.7	NimBLE Host	585
7.7.1	Introduction	585
7.7.2	NimBLE Host Return Codes	586
7.7.3	NimBLE Host GAP Reference	594
7.7.4	NimBLE Host GATT Client Reference	625
7.7.5	NimBLE Host GATT Server Reference	641
7.7.6	NimBLE Host Identity Reference	656
7.7.7	NimBLE Host ATT Client Reference	658
7.8	API for btshell app	660
7.8.1	Usage	660
7.8.2	GAP API for btshell	663
7.8.3	GATT feature API for btshell	673
7.8.4	Advertisement Data Fields	674
7.9	Bluetooth Mesh	676
7.9.1	Introduction to Mesh	676
7.9.2	Topology	676
7.9.3	Bearers	677
7.9.4	Provisioning	677
7.9.5	Models	677
7.9.6	Mesh Node features supported by Apache Mynewt	679
7.9.7	Sample application	679
8	Newt Tool Guide	681
8.1	Introduction	681

8.1.1	Rationale	681
8.1.2	Build System	681
8.1.3	More operations using Newt	683
8.1.4	Source Management and Repositories	683
8.1.5	Notes:	684
8.2	Theory of Operations	685
8.2.1	Building dependencies	685
8.2.2	Resolving dependencies	687
8.2.3	Producing artifacts	688
8.2.4	Download/Debug Support	689
8.3	Command Structure	690
8.3.1	newt build	691
8.3.2	newt clean	692
8.3.3	newt complete	693
8.3.4	newt create-image	694
8.3.5	newt debug	695
8.3.6	newt help	696
8.3.7	newt info	697
8.3.8	newt load	698
8.3.9	newt mfg	698
8.3.10	newt new	701
8.3.11	newt pkg	701
8.3.12	newt resign-image	703
8.3.13	newt run	703
8.3.14	newt size	704
8.3.15	newt target	706
8.3.16	newt test	710
8.3.17	newt upgrade	711
8.3.18	newt vals	711
8.3.19	newt version	713
8.4	Install	713
8.4.1	Installing Newt on macOS	713
8.4.2	Installing Newt on Linux	717
8.4.3	Installing Newt on Windows	720
8.4.4	Installing Previous Releases of Newt	724
9	Newt Manager Guide	727
9.1	Command List	727
9.1.1	Overview	727
9.1.2	newtmgr config	728
9.1.3	newtmgr conn	728
9.1.4	newtmgr crash	732
9.1.5	newtmgr datetime	732
9.1.6	newtmgr echo	733
9.1.7	newtmgr fs	734
9.1.8	newtmgr image	735
9.1.9	newtmgr log	738
9.1.10	newtmgr mpstat	740
9.1.11	newtmgr reset	742
9.1.12	newtmgr run	742
9.1.13	newtmgr stat	744
9.1.14	newtmgr taskstat	746
9.2	Install	747
9.2.1	Installing Newtmgr on Mac OS	747

9.2.2	Installing Newtmgr on Linux	750
9.2.3	Installing Newtmgr on Windows	753
9.2.4	Installing Previous Releases of Newtmgr	756
10	Mynewt FAQ	759
10.1	General FAQs	759
10.1.1	Reduce Code Size for Mynewt Image	759
10.1.2	<code>compiler.yml</code> vs. <code>pkg.yml</code>	759
10.1.3	Version Control Applications with Git	759
10.2	Mynewt FAQ - Administrative	760
10.2.1	Administrative questions:	760
10.3	Mynewt FAQ - Bluetooth	762
10.3.1	NimBLE on nRF52840	762
10.3.2	Trigger Exactly One BLE Advertisement Immediately	762
10.3.3	Supported Bluetooth Radio Transceivers	763
10.3.4	NimBLE Operational Questions	763
10.3.5	<code>bluetooth</code> Forgets All Keys on Restart	763
10.3.6	L2CAP Connection	764
10.3.7	Bitbang, BLE Mesh, BLE Advertising	764
10.3.8	Extended Advertising with <code>btshell</code>	764
10.3.9	Configuring Maximum Number of Connections for <code>blehci</code>	765
10.3.10	Disconnect/Crash While Writing Analog Value From Central Module	765
10.3.11	Documentation for <code>ble_gap_disc_params</code>	766
10.3.12	Multicast Messaging and Group Messaging	766
10.3.13	Read the Value of a Characteristic of a Peripheral Device from a Central Device	766
10.4	Mynewt FAQ - Bootloader and Firmware Upgrade	767
10.4.1	Firmware Upgrade Capability	767
10.4.2	Bootloader Documentation	767
10.4.3	MCUboot vs. Mynewt Bootloader	767
10.4.4	<code>boot_serial</code> vs. <code>bootutil</code>	767
10.5	Mynewt FAQ - Drivers and Modules	768
10.5.1	Drivers in Mynewt	768
10.5.2	Module Argument in Mynewt Logging Library	768
10.5.3	Log Name vs. Module Number	768
10.6	Mynewt FAQ - File System	769
10.6.1	<code>nffs</code> Setup	769
10.7	Mynewt FAQ - Hardware-Specific Questions	769
10.7.1	Nordic nRF52-DK	769
10.7.2	Redbear BLE Nano 2	770
10.8	Mynewt FAQ - Syntax, Semantics, Configuration	770
10.8.1	Floating Point in Mynewt	771
10.8.2	Ending the Delay of a Task Blocking a Call Early	771
10.8.3	Random Function / Device	771
10.8.4	Setting <code>serial</code> and <code>mfghash</code>	771
10.8.5	Leading Zeros Format in <code>printf</code>	771
10.8.6	Mynewt Equivalent of UNIX <code>sleep(3)</code>	772
10.8.7	Alternatives to <code>cmsis_nvic.c</code>	772
10.9	Mynewt FAQ - NFC	772
10.10	Mynewt FAQ - Newt	772
10.10.1	<code>newt size</code> Command vs. Elf File Size	772
10.11	Mynewt FAQ - Newt Manager	772
10.11.1	Connection Profile with <code>newtmgr</code>	773
10.11.2	NMP	773
10.11.3	Communicate with a Device based on <code>peer_id</code>	773

10.11.4	Newt Manager with the Adafruit nRF52DK	774
10.12	Mynewt FAQ - Porting Mynewt	774
10.12.1	Porting Mynewt to Core-M3 MCU	774
10.13	Mynewt FAQ - Troubleshooting/Debugging	775
10.13.1	Concurrent debug sessions with two boards	775
10.13.2	Error: Unsatisfied APIs detected	775
10.13.3	Greyed out files on iOS 11	775
10.13.4	<code>arm-none-eabi-gcc</code> Build Error for Project	776
10.13.5	Issues Running Image on Boot	776
10.13.6	Enable Trace in Mynewt	776
10.13.7	Bug With Older Versions of <code>gcc</code>	776
11	Appendix	777
11.1	Contributing to Newt or Newtmgr Tools	777
11.1.1	Step 1: Installing Go	778
11.1.2	Step 2: Setting Up Your Go Environment	778
11.1.3	Step 3: Downloading the Source and Installing the Tools	779
11.1.4	Step 4: Updating and Rebuilding the Tools:	780
11.2	Developing Mynewt Applications with Visual Studio Code	781
11.2.1	Installing Visual Studio Code	782
11.2.2	Installing the C/C++ and Debugger Extensions	782
11.2.3	Defining Tasks for Mynewt Projects	782
11.2.4	Defining Debugger Configurations	786
11.2.5	Debugging Your Application	787
11.2.6	Working with Multiple Mynewt Applications	788
Index		789

INTRODUCTION

1.1 Welcome to Apache Mynewt

Apache Mynewt is an operating system that makes it easy to develop applications for microcontroller environments where power and cost are driving factors. Examples of these devices are connected locks, lights, and wearables.

Microcontroller environments have a number of characteristics that makes the operating system requirements for them unique:

- Low memory footprint: memory on these systems range from 8-16KB (on the low end) to 16MB (on the high end).
- Reduced code size: code often runs out of flash, and total available code size ranges from 64-128KB to 16-32MB.
- Low processing speed: processor speeds vary from 10-12MHz to 160-200MHz.
- Low power operation: devices operate in mostly sleeping mode, in order to conserve battery power and maximize power usage.

As more and more devices get connected, these interconnected devices perform complex tasks. To perform these tasks, you need low-level operational functionality built into the operating system. Typically, connected devices built with these microcontrollers perform a myriad of functions:

- Networking Stacks: Bluetooth Low Energy and Thread
- Peripherals: PWM to drive motors, ADCs to measure sensor data, and RTCs to keep time.
- Scheduled Processing: actions must happen on a calendared or periodic basis.

Apache Mynewt accomplishes all the above easily, by providing a complete operating system for constrained devices, including:

- A fully open-source Bluetooth Low Energy stack with both Host and Controller implementations.
- A pre-emptive, multi-tasking Real Time operating system kernel
- A Hardware Abstraction Layer (HAL) that abstracts the MCU's peripheral functions, allowing developers to easily write cross-platform code.

1.2 Newt

In order to provide all this functionality, and operate in an extremely low resource environment, Mynewt provides a very fine-grained source package management and build system tool, called *newt*.

You can install *newt* for [Mac OS](#), [Linux](#), or [Windows](#).

1.3 Newt Manager

In order to enable a user to communicate with remote instances of Mynewt OS and query, configure, and operate them, Mynewt provides an application tool called Newt Manager or *newtmgr*.

You can install *newtmgr* for [Mac OS](#), [Linux](#), or [Windows](#).

1.4 Build your first Mynewt App with Newt

With the introductions out of the way, now is a good time to [get set up and started](#) with your first Mynewt application. Happy Hacking!

SETUP & GET STARTED

If you are curious about Mynewt and want to get a quick feel for the project, you've come to the right place. We have two options for you:

Option 1 (Recommended) allows you to install the Newt tool, instances of the Mynewt OS (for simulated targets), and toolchains for developing embedded software (e.g. GNU toolchain) natively on your laptop or computer. We have tried to make the process easy. For example, for the Mac OS we created brew formulas.

We recommend this option if you are familiar with such environments or are concerned about performance on your machine. Follow the instructions in “[Native Installation](#)” if you prefer this option.

Option 2 is an easy, self-contained way to get up and running with Mynewt - but has limitations! The Newt tool and build toolchains are all available in a single [Docker Container](#) that you can install on your laptop or computer.

However, this is not a long-term option since support is not likely for all features useful or critical to embedded systems development. For example, USB device mapping available in the Docker toolkit is no longer available in the new Docker releases. The Docker option is also typically slower than the native install option.

You can then proceed with the instructions on how to [Creating Your First Mynewt Project](#) - on simulated hardware.

Upon successful start, several tutorials await your eager attention!

Send us an email on the dev@ mailing list if you have comments or suggestions! If you haven't joined the mailing list, you will find the links [here](#).

2.1 Native Installation

This section shows you how to install the tools to develop and build Mynewt OS applications on Mac OS, Linux, and Windows, and run and debug the applications on target boards. For Mac OS and Linux, you can also build Mynewt OS applications that run on Mynewt's simulated hardware. These applications run natively on Mac OS and Linux.

The tools you need are:

- **Newt tool:** Tool to create, build, load, and debug Mynewt OS applications.
 - [Installing Newt on macOS](#).
 - [Installing Newt on Linux](#).
 - [Installing Newt on Windows](#).
- **Native toolchain:** Native toolchain to compile and build Mynewt OS applications that run on Mynewt's simulated hardware on Mac OS and Linux.
(See [Installing Native Toolchain](#)).
- **Cross tools for ARM:**

- Cross toolchain for ARM to compile and build Mynewt OS applications for target boards.
- Debuggers to load and debug applications on target boards.

(See [Installing the Cross Tools for ARM](#)).

If you would like to use an IDE to develop and debug Mynewt applications, see [Developing Mynewt Applications with Visual Studio Code](#). You must still perform the native installation outlined on this page.

2.1.1 Installing Native Toolchain

- [Setting Up the Toolchain for Mac](#)
 - [Installing Brew](#)
 - [Installing GCC/libc](#)
 - [Installing GDB](#)
- [Setting Up the Toolchain for Linux](#)
 - [Installing GCC/libc that will produce 32-bit executables](#)
 - [Installing GDB](#)
- [Next](#)

This page shows you how to install the toolchain to build Mynewt OS applications that run native on macOS and Linux. The applications run on Mynewt's simulated hardware. It also allows you to run the test suites for all packages that do not require HW support.

Note: This is not supported on Windows.

Setting Up the Toolchain for Mac

Installing Brew

If you have not already installed Homebrew from the newt tutorials pages install it ([Installing Newt on macOS](#)).

Installing GCC/libc

Since macOS with Xcode ships with a C compiler called Clang there is no need to install the other compiler. However, one can install gcc compiler with brew:

```
$ brew install gcc
...
...
==> Pouring gcc-10.2.0_2.big_sur.bottle.tar.gz
    /usr/local/Cellar/gcc/10.2.0_2: 1,455 files, 338.1MB
```

Check the GCC version you have installed (either using brew or previously installed). The brew-installed version can be checked using brew list gcc. To use gcc instead of clang <mynewt-src-directory>/repos/apache-mynewt-core/compiler/sim/compiler.yml file must be edited:

```
# OS X.
compiler.path.cc.DARWIN.OVERWRITE: "gcc"
compiler.path.as.DARWIN.OVERWRITE: "gcc"
compiler.path.objdump.DARWIN.OVERWRITE: "gobjdump"
compiler.path.objcopy.DARWIN.OVERWRITE: "gobjcopy"
compiler.flags.base.DARWIN: [-DMN OSX, -Wno-missing-braces]
compiler.ld.resolve_circular_deps.DARWIN.OVERWRITE: false
```

... with the following:

```
compiler.path.cc.DARWIN.OVERWRITE: "gcc-<version>"
compiler.path.as.DARWIN.OVERWRITE: "gcc-<version>"
```

Installing GDB

```
$ brew install gdb
...
...
==> Summary
/usr/local/Cellar/gdb/7.10.1: XXX files, YYM
```

NOTE: When running a program with GDB, you may need to sign your gdb executable. This page shows a recipe for gdb signing. Alternately you can skip this step and continue without the ability to debug your mynewt application on your PC.*

Setting Up the Toolchain for Linux

The below procedure can be used to set up a Debian-based Linux system (e.g., Ubuntu). If you are running a different Linux distribution, you will need to substitute invocations of *apt-get* in the below steps with the package manager that your distro uses.

Installing GCC/libc that will produce 32-bit executables

```
$ sudo apt-get install gcc-multilib libc6-i386
```

Installing GDB

```
$ sudo apt-get install gdb
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  gdb-doc gdbserver
The following NEW packages will be installed:
  gdb
...
```

(continues on next page)

(continued from previous page)

```
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Setting up gdb (7.7.1-0ubuntu5~14.04.2) ...
```

Next

At this point you have installed all the necessary software to build and run your first project on a simulator on your macOS or Linux computer. You may proceed to [Creating Your First Mynewt Project](#) or continue to the next section and install the cross tools for ARM.

2.1.2 Installing the Cross Tools for ARM

- *Installing the ARM Cross Toolchain*
 - *Installing the ARM Toolchain For Mac OS X*
 - *Installing the ARM Toolchain For Linux*
 - *Installing the ARM Toolchain for Windows*
- *Installing the Debuggers*
 - *Installing the OpenOCD Debugger*
 - *Installing OpenOCD on Mac OS*
 - *Installing OpenOCD on Linux*
 - *Installing OpenOCD on Windows*
 - *Installing SEGGER J-Link*

This page shows you how to install the tools to build, run, and debug Mynewt OS applications that run on supported ARM target boards. It shows you how to install the following tools on Mac OS, Linux and Windows:

- ARM cross toolchain to compile and build Mynewt applications for the target boards.
- Debuggers to load and debug applications on the target boards.

Installing the ARM Cross Toolchain

ARM maintains a pre-built GNU toolchain with gcc and gdb targeted at Embedded ARM Processors, namely Cortex-R/Cortex-M processor families. Mynewt OS has been tested with version 4.9 of the toolchain and we recommend you install this version to get started. Mynewt OS will eventually work with multiple versions available, including the latest releases.

Installing the ARM Toolchain For Mac OS X

Add the **PX4/homebrew-px4** homebrew tap and install version 4.9 of the toolchain. After installing, check that the symbolic link that homebrew created points to the correct version of the debugger.

```
$ brew tap PX4/homebrew-px4
$ brew update
$ brew install gcc-arm-none-eabi-49
$ 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.
$ ls -al /usr/local/bin/arm-none-eabi-gdb
lrwxr-xr-x 1 aditihilbert admin 69 Sep 22 17:16 /usr/local/bin/arm-none-eabi-gdb -> /
↪usr/local/Cellar/gcc-arm-none-eabi-49/20150609/bin/arm-none-eabi-gdb
```

Note: If no version is specified, brew will install the latest version available.

Installing the ARM Toolchain For Linux

On a Debian-based Linux distribution, gcc 4.9.3 for ARM can be installed with apt-get as documented below. The steps are explained in depth at <https://launchpad.net/~team-gcc-arm-embedded/+archive/ubuntu/ppa>.

```
$ 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
$ sudo apt-get install gdb-arm-none-eabi
```

Installing the ARM Toolchain for Windows

1. Download and run the [installer](#) to install arm-none-eabi-gcc and arm-none-eabi-gdb. Select the default destination folder: **C:\Program Files (x86)\GNU Arm Embedded Toolchain\10 2020-q4-major**.

Notes:

- Check the **Add path to environment variable** option before you click the **Finish** button for the installation.
- You may select a different folder but the installation instructions use the default values.

2. Check that you are using the installed versions arm-none-eabi-gcc and arm-none-eabi-gdb. Open a MinGW terminal and run the **which** commands.

Note: You must start a new MinGW terminal to inherit the new **Path** values.

```
$ which arm-none-eabi-gcc
/c/Program Files (x86)/GNU Arm Embedded Toolchain/10 2020-q4-major/bin/arm-none-
↪eabi-gcc
$ which arm-none-eabi-gdb
/c/Program Files (x86)/GNU Arm Embedded Toolchain/10 2020-q4-major/bin/arm-none-
↪eabi-gdb
```

Installing the Debuggers

Mynewt uses, depending on the board, either the OpenOCD or SEGGER J-Link debuggers.

Installing the OpenOCD Debugger

OpenOCD (Open On-Chip Debugger) is open-source software that allows your computer to interface with the JTAG debug connector on a variety of boards. A JTAG connection lets you debug and test embedded target devices. For more on OpenOCD go to <http://openocd.org>.

OpenOCD version 0.10.0 with nrf52 support is required. A binary for this version is available to download for Mac OS, Linux, and Windows.

Installing OpenOCD on Mac OS

1. Install latest OpenOCD from Homebrew:

```
$ brew update  
$ brew install openocd --HEAD
```

1. Check the OpenOCD version you are using:

```
$ which openocd  
/usr/local/bin/openocd  
  
$ 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
```

You should see version: **0.10.0+dev-<latest#>**.

Installing OpenOCD on Linux

1. Download the [binary tarball](#) for Linux
2. Change to the root directory:

```
$ cd /
```

3. Untar the tarball and install into **/usr/local/bin**. You will need to replace **~/Downloads** with the directory that the tarball is downloaded to.

Note: You must specify the **-p** option for the tar command.

```
$ sudo tar -xpf ~/Downloads/openocd-bin-0.10.0-Linux.tgz
```

4. Check the OpenOCD version you are using:

```
$ which openocd  
/usr/local/bin/openocd  
$ openocd -v
```

(continues on next page)

(continued from previous page)

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

You should see version: **0.10.0**.

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
```

Installing OpenOCD on Windows

1. Download the [binary zip file](#) for Windows.
2. Extract into the **C:\openocd-0.10.0** folder.
3. Add the path: **C:\openocd-0.10.0\bin** to your Windows User **Path** environment variable. Note: You must add **bin** to the path.
4. Check the OpenOCD version you are using. Open a new MinGW terminal and run the following commands:

Note: You must start a new MinGW terminal to inherit the new **Path** values.

```
$ which openocd
/c/openocd-0.10.0/bin/openocd
$ 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
```

You should see version: **0.10.0**.

Installing SEGGER J-Link

You can download and install Segger J-LINK Software and documentation pack from [SEGGER](#).

Note: On Windows, perform the following after the installation:

- Add the installation destination folder path to your Windows user **Path** environment variable. You do not need to add **bin** to the path.
- Open a new MinGW terminal to inherit the new **Path** values.

2.2 Everything You Need in a Docker Container

Docker provides a quick and easy way to get up and running with Mynewt. The newt command line tool and the entire build toolchain is available in a single docker container. The container is all that's needed to run your Mynewt based application in the simulator. Enabling USB2 with your docker installation will allow you to load your application on a supported device.

Docker is the only supported option if you are working on a Windows machine. If you are using Mac OS X or Linux, you have the choice of installing a Docker container of tools and toolchains or installing them natively. This chapter describes how to set up the Docker image for all three platforms.

- *Install Docker*
 - *Mac and Windows*
 - *Linux*
- *Pull the Mynewt Container*
- *Use a newt wrapper script*
- *Enable USB2 Support for Mac or Windows*
 - *Install VirtualBox extension pack*
 - *Enable USB2 and select your device*

2.2.1 Install Docker

Install docker for your platform. [Mac OS X / Windows / Linux](#)

Mac and Windows

Mac and Windows require Docker Toolbox to interact with USB devices. Docker for Mac and Docker for Windows do not support USB. Docker Toolbox uses VirtualBox and allows you to map USB devices into docker containers as described below.

Make sure to double click the Docker Quickstart Terminal application if you're on Mac or Windows.

Linux

The docker daemon listens on a Unix domain socket on Linux. That socket is owned by root, which means by default you must be root to start a container. Make sure to follow the optional step of adding yourself to the docker group so you can start the newt container as yourself.

2.2.2 Pull the Mynewt Container

```
$ docker pull mynewt/newt:latest
```

Note You can upgrade your container by repeating this command when updates are made available.

2.2.3 Use a newt wrapper script

Use the newt wrapper script to invoke newt. Create the following file, name it `newt`, make it executable, and put it in your path. This will allow you to run newt as if it was natively installed. You can now follow the normal tutorials using the newt wrapper script.

```
#!/bin/bash

if [ "$1" = "debug" ] || [ "$1" = "run" ]
then
    ti="-ti"
fi

docker run -e NEWT_USER=$(id -u) -e NEWT_GROUP=$(id -g) -e NEWT_HOST=$(uname) $ti --rm --
device=/dev/bus/usb --privileged -v $(pwd):/workspace -w /workspace mynewt/newt:latest
~/newt "$@"
```

Note Remember to point to the correct subdirectory level when invoking `newt`. For example, invoke it using `./newt` in the example below.

```
user@~/dockertest$ ls
myproj  newt
user@~/dockertest$ cd myproj
user@~/dockertest/myproj$ ./newt version
Apache Newt version: 1.1.0
```

2.2.4 Enable USB2 Support for Mac or Windows

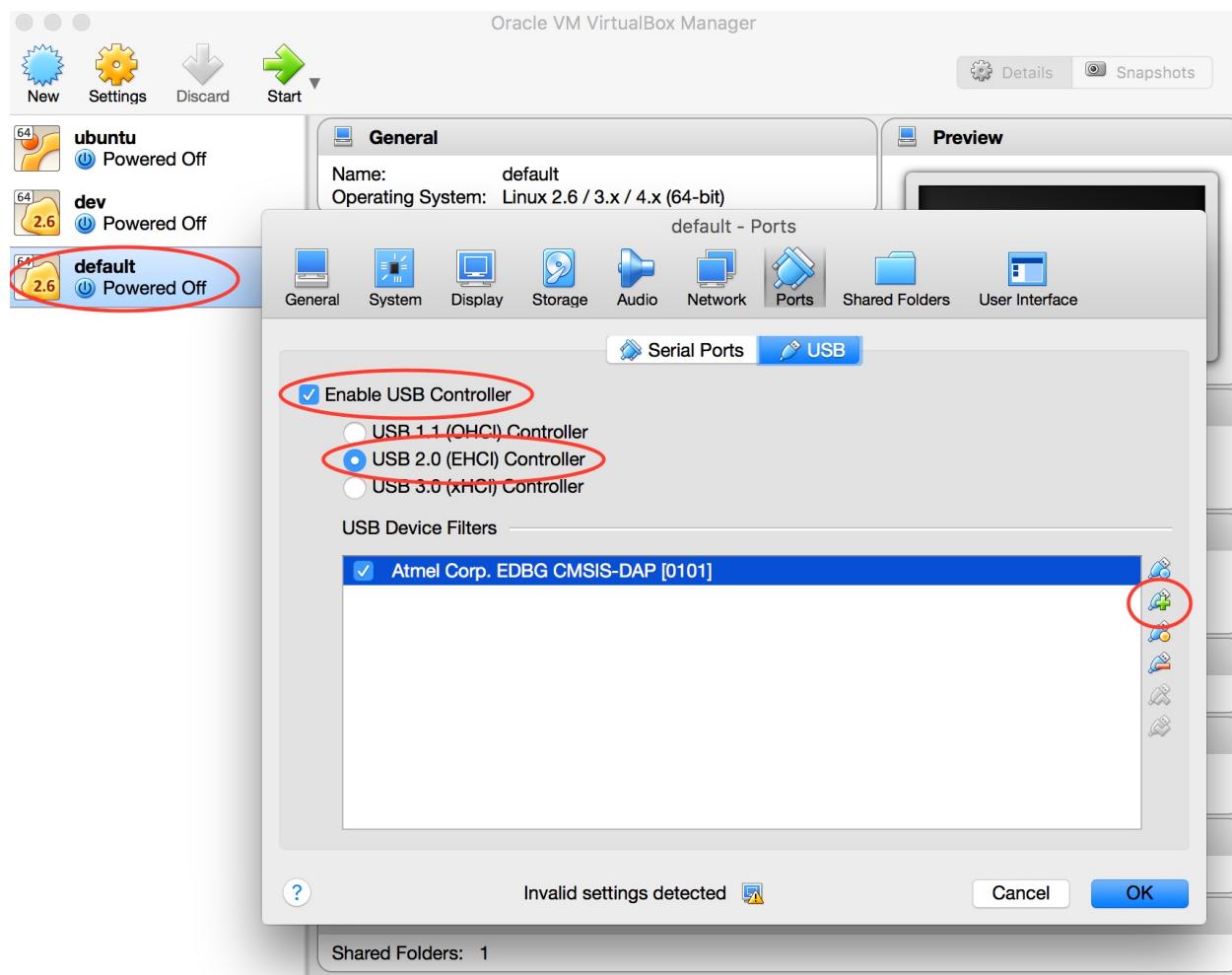
If you plan on loading your application on an actual device, do the steps below.

Install VirtualBox extension pack

Docker uses a VirtualBox Linux VM to run containers. A free VirtualBox extension pack is required to enable USB2 support. Download the [VirtualBox Extension Pack](#) version that matches your VirtualBox installation and double click to install

Enable USB2 and select your device

- The “default” VM created by docker-machine must first be stopped before you can enable USB2. You have two options:
 - Run the command `docker-machine stop default` in the terminal window or
 - Use the VirtualBox UI. Right click on `default` -> Close -> Power Off
- Enable USB2 using the VirtualBox UI. Select the “default” VM->Settings->Ports->USB2 to enable USB2. Add your device to the USB Device Filters to make the device visible in the docker container. See the image below.
- Restart the “default” VM. You have two options:
 - Run `docker-machine start default` in the terminal window or
 - Use the VirtualBox UI. Make sure the “default” machine is highlighted. Click the green “Start” button. Select “Headless Start”.



Note: When working with actual hardware, remember that each board has an ID. If you swap boards and do not refresh the USB Device Filter on the VirtualBox UI, the ID might be stale and the Docker instance may not be able to see the board correctly. For example, you may see an error message like `Error: unable to find CMSIS-DAP device` when you try to load or run an image on the board. In that case, you need to click on the USB link in VirtualBox UI, remove the existing USB Device Filter (e.g. “Atmel Corp. EDBG CMSIS-DAP[0101]”) by clicking on the “Removes selected USB filter” button, and add a new filter by clicking on the “Adds new USB filter” button.

2.3 Creating Your First Mynewt Project

This page shows you how to create a Mynewt project using the `newt` command-line tool. The project is a blinky application that toggles a pin. The application uses the Mynewt's simulated hardware and runs as a native application on Linux, FreeBSD and older versions of Mac OS.

Note: The Mynewt simulator is not yet supported natively on Windows or newer Mac OS versions.

If you are using the native install option (not the Docker option), you will need to create the blinky application for a target board. We recommend that you read the section on creating a new project and fetching the source repository to understand the Mynewt repository structure, create a new project, and fetch the source dependencies before proceeding to one of the [Project Blinky](#).

This guide shows you how to:

1. Create a new project and fetch the source repository and dependencies.
2. Test the project packages. (Using Docker image for Windows and MacOS.)
3. Build and run the simulated blinky application.

- *Prerequisites*
- *Creating a New Project and Fetching the Source Repository*
 - *Creating a New Project*
 - *Fetching the Mynewt Source Repository and Dependencies*
- *Testing the Project Packages*
- *Building and Running the Simulated Blinky Application*
 - *Building the Application*
 - *Running the Blinky Application*
- *Exploring other Mynewt OS Features*

2.3.1 Prerequisites

- Have Internet connectivity to fetch remote Mynewt components.
- Install the `newt` tool:
 - If you have taken the native install option, see the installation instructions for [Mac OS](#), [Linux](#), or [Windows](#).
 - If you have taken the Docker option, you have already installed Newt.
- Install the [*native toolchain*](#) to compile and build a Mynewt native application.

2.3.2 Creating a New Project and Fetching the Source Repository

This section describes how to use the newt tool to create a new project and fetch the core mynewt source repository.

Creating a New Project

Choose a name for your project. We name the project **myproj**.

Run the `newt new myproj` command, from your **dev** directory, to create a new project:

Note: This tutorial assumes you created a **dev** directory under your home directory.

```
$ cd ~/dev
$ newt new myproj
Downloading project skeleton from apache/mynewt-blinky...
Downloading repository mynewt-blinky (commit: master) ...
Installing skeleton in myproj...
Project myproj successfully created.
```

The newt tool creates a project base directory name **myproj**. All newt tool commands are run from the project base directory. The newt tool populates this new project with a base skeleton of a new Apache Mynewt project in the project base directory. It has the following structure:

Note: If you do not have `tree`, run `brew install tree` to install on Mac OS, `sudo apt-get install tree` to install on Linux, `pacman -Sv tree` from a MinGW terminal to install on Windows, and `pkg install tree` on FreeBSD.

```
$ cd myproj
$ tree

.
├── DISCLAIMER
├── LICENSE
├── NOTICE
├── README.md
└── apps
    └── blinky
        ├── pkg.yml
        └── src
            └── main.c
├── project.yml
└── targets
    ├── my_blinky_sim
    │   ├── pkg.yml
    │   └── target.yml
    └── unittest
        ├── pkg.yml
        └── target.yml

6 directories, 11 files
```

The newt tool installs the following files for a project in the project base directory:

1. The file `project.yml` contains the repository list that the project uses to fetch its packages. Your project is a collection of repositories. In this case, the project only comprises the core mynewt repository. Later, you will add more repositories to include other mynewt components.

2. The file `apps/blinky/pkg.yml` contains the description of your application and its package dependencies.
3. A `target` directory that contains the `my_blinky_sim` directory. The `my_blinky_sim` directory has target information to build a version of myproj. Use `newt target show` to see available build targets.
4. A non-buildable target called `unittest`. This is used internally by newt and is not a formal build target.

Note: The actual code and package files are not installed (except the template for `main.c`). See the next step to install the packages.

Fetching the Mynewt Source Repository and Dependencies

By default, Mynewt projects rely on a single repository: `apache-mynewt-core` and uses the source in the master branch. If you need to use a different branch, you need to change the `vers` value in the `project.yml` file:

```
repository.apache-mynewt-core:
  type: github
  vers: 1-latest
  user: apache
  repo: mynewt-core
```

Changing `vers` to `0-dev` will put you on the latest master branch. **This branch may not be stable and you may encounter bugs or other problems.**

Run the `newt upgrade` command, from your project base directory (`myproj`), to fetch the source repository and dependencies.

Note: It may take a while to download the `apache-mynewt-core` repository. Use the `-v` (verbose) option to see the installation progress.

```
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
Downloading repository mynewt-mcumgr (commit: master) ...
Downloading repository mynewt-nimble (commit: master) ...
Downloading repository mcuboot (commit: master) from ...
Making the following changes to the project:
apache-mynewt-core successfully upgraded to version 1.7.0
apache-mynewt-nimble successfully upgraded to version 1.2.0
mcuboot successfully upgraded to version 1.3.1
```

View the core of the Apache Mynewt OS that is downloaded into your local directory.

(The actual output will depend on what is in the latest ‘master’ branch)

```
$ repos/apache-mynewt-core/
├── apps
│   ├── bleprph_oic
│   ├── blesplit
│   ├── bleuart
│   ├── bsncent
│   ├── bsnprph
│   ├── bus_test
│   ├── coremark
│   ├── crypto_test
│   ├── ffs2native
│   └── flash_loader
```

(continues on next page)

(continued from previous page)

```
    ├── iptest
    ├── lora_app_shell
    ├── loraping
    ├── lorashell
    ├── metrics
    ├── ocf_sample
    ├── pwm_test
    ├── sensors_test
    ├── slinky
    ├── slinky_oic
    ├── spitest
    ├── splitty
    ├── testbench
    ├── timtest
    └── trng_test
  └── boot
    ├── split
    └── split_app
  └── CODING_STANDARDS.md
  └── compiler
    ├── arc
    ├── arm-none-eabi-m0
    ├── arm-none-eabi-m3
    ├── arm-none-eabi-m33
    ├── arm-none-eabi-m4
    ├── arm-none-eabi-m7
    ├── gdbmacros
    ├── mips
    ├── riscv64
    ├── sim
    ├── sim-armv7
    ├── sim-mips
    └── xc32
  └── crypto
    ├── mbedtls
    └── tinycrypt
  └── docs
    ├── conf.py
    ├── doxygen.xml
    ├── index.rst
    ├── Makefile
    ├── os
    └── README.rst
  └── encoding
    ├── base64
    ├── cborattr
    ├── json
    └── tinyccbor
  └── fs
    ├── disk
    ├── fatfs
    └── fcb
```

(continues on next page)

(continued from previous page)

```
    └── fcb2
    └── fs
    └── nffs
  └── hw
    ├── battery
    ├── bsp
    ├── bus
    ├── charge-control
    ├── cmsis-core
    ├── drivers
    ├── hal
    ├── mcu
    ├── mips-hal
    ├── scripts
    ├── sensor
    └── util
  └── kernel
    ├── os
    └── sim
  └── libc
    └── baselibc
  └── LICENSE
  └── mgmt
    ├── imgmgr
    ├── mgmt
    ├── newtmgr
    └── oicmgr
  └── net
    ├── ip
    ├── lora
    ├── mqtt
    ├── oic
    └── wifi
  └── NOTICE
  └── project.yml
  └── README.md
  └── RELEASE_NOTES.md
  └── repository.yml
  └── sys
    ├── config
    ├── console
    ├── coredump
    ├── defs
    ├── fault
    ├── flash_map
    ├── id
    ├── log
    ├── metrics
    ├── mfg
    ├── reboot
    ├── shell
    └── stats
```

(continues on next page)

(continued from previous page)

└── sys
└── sysdown
└── sysinit
└── sysview
── targets
└── unittest
── test
├── crash_test
├── flash_test
├── i2c_scan
├── runtest
└── spiflash_stress_test
└── testutil
── time
├── datetime
├── timepersist
└── timesched
── uncrustify.cfg
── util
├── cbmem
├── cmdarg
├── crc
├── debounce
├── easing
├── mem
├── parse
├── rwlock
├── streamer
└── taskpool
version.yml

131 directories, 14 files

2.3.3 Testing the Project Packages

Note: If you're running this with Docker, use `newt` wrapper script as described in [Docker Container](#). Unit tests depend on Mynewt simulator.

You can use the `newt` tool to execute the unit tests in a package. For example, run the following command to test the `sys/config` package in the `apache-mynewt-core` repo:

```
$ newt test @apache-mynewt-core/sys/config
Testing package @apache-mynewt-core/sys/config/selftest-fcb
Compiling repos/apache-mynewt-core/crypto/tinycrypt/src/aes_decrypt.c
Compiling repos/apache-mynewt-core/crypto/tinycrypt/src/aes_encrypt.c
Compiling repos/apache-mynewt-core/crypto/tinycrypt/src/cbc_mode.c
Compiling repos/apache-mynewt-core/crypto/tinycrypt/src/ccm_mode.c
Compiling repos/apache-mynewt-core/crypto/tinycrypt/src/cmac_mode.c
...

```

Linking ~/dev/myproj/bin/targets/unittest/sys_config_selftest-fcb/app/sys/config/

(continues on next page)

(continued from previous page)

```

↳ selftest-fcb/sys_config_selftest-fcb.elf
Executing test: ~/dev/myproj/bin/targets/unittest/sys_config_selftest-fcb/app/sys/config/
↳ selftest-fcb/sys_config_selftest-fcb.elf
Testing package @apache-mynewt-core/sys/config/selftest-nffs
Compiling repos/apache-mynewt-core/encoding/base64/src/hex.c
Compiling repos/apache-mynewt-core/fs/fs/src/fs_cli.c
Compiling repos/apache-mynewt-core/fs/fs/src/fs dirent.c
Compiling repos/apache-mynewt-core/fs/fs/src/fs mkdir.c
Compiling repos/apache-mynewt-core/fs/fs/src/fs mount.c
Compiling repos/apache-mynewt-core/encoding/base64/src/base64.c
Compiling repos/apache-mynewt-core/fs/fs/src/fs file.c
Compiling repos/apache-mynewt-core/fs/disk/src/disk.c
Compiling repos/apache-mynewt-core/fs/fs/src/fs_nmgr.c
Compiling repos/apache-mynewt-core/fs/fs/src/fsutil.c
Compiling repos/apache-mynewt-core/fs/nffs/src/nffs.c

...
Linking ~/dev/myproj/bin/targets/unittest/sys_config_selftest-nffs/app/sys/config/
↳ selftest-nffs/sys_config_selftest-nffs.elf
Executing test: ~/dev/myproj/bin/targets/unittest/sys_config_selftest-nffs/app/sys/
↳ config/selftest-nffs/sys_config_selftest-nffs.elf
Passed tests: [sys/config/selftest-fcb sys/config/selftest-nffs]
All tests passed

```

Note: If you installed the latest gcc using homebrew on your Mac, you are probably running gcc-6. Make sure you change the compiler.yml configuration to specify that you are using gcc-6 (See [Installing Native Toolchain](#)). You can also downgrade your installation to gcc-5 and use the default gcc compiler configuration for MyNewt:

```
$ brew uninstall gcc-6
$ brew link gcc-5
```

Note: If you are running the standard gcc for 64-bit machines, it does not support 32-bit. In that case you will see compilation errors. You need to install multilib gcc (e.g. gcc-multilib if you running on a 64-bit Ubuntu). **Note:** Running newt test all within Docker Container can take a long time.

To test all the packages in a project, specify all instead of the package name.

```

$ newt test all
Testing package @apache-mynewt-core/crypto/mbedtls/selftest
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/aesni.c
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/aria.c
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/arc4.c
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/aes.c

...
Linking ~/dev/myproj/bin/targets/unittest/crypto_mbedtls_selftest/app/@apache-mynewt-
↳ core/crypto/mbedtls/selftest/@apache-mynewt-core_crypto_mbedtls_selftest.elf
Executing test: ~/dev/myproj/bin/targets/unittest/crypto_mbedtls_selftest/app/@apache-
↳ mynewt-core/crypto/mbedtls/selftest/@apache-mynewt-core_crypto_mbedtls_selftest.elf
...lots of compiling and testing...

```

(continues on next page)

(continued from previous page)

```
Linking ~/dev/myproj/bin/targets/unittest/boot_boot_serial_test/app/@mcuboot/boot/boot_
↪serial/test/@mcuboot_boot_boot_serial_test.elf
Executing test: ~/dev/myproj/bin/targets/unittest/boot_boot_serial_test/app/@mcuboot/
↪boot/boot_serial/test/@mcuboot_boot_boot_serial_test.elf
Passed tests: [crypto/mbedtls/selftest encoding/base64/selftest encoding/cborattr/
↪selftest encoding/json/selftest fs/fcb/selftest fs/fcb2/selftest fs/nffs/selftest hw/
↪drivers/flash/enc_flash/selftest hw/drivers/trng/trng_sw/selftest hw/sensor/selftest_
↪kernel/os/selftest net/ip/mm_socket/selftest net/oic/selftest sys/config/selftest-fcb_
↪sys/config/selftest-nffs sys/flash_map/selftest sys/log/full/selftest/align1 sys/log/
↪full/selftest/align2 sys/log/full/selftest/align4 sys/log/full/selftest/align8 sys/log/
↪full/selftest/fcb_bookmarks sys/log/modlog/selftest util/cbmem/selftest util/debounce/
↪selftest util/rwlock/selftest cborattr/test nimble/controller/test nimble/host/test_
↪boot/boot_serial/test]
All tests passed
```

2.3.4 Building and Running the Simulated Blinky Application

The section shows you how to build and run the blinky application to run on Mynewt's simulated hardware.

Note: This is not yet supported on Windows or newer versions of MacOS. Refer to the [Project Blinky](#) to create a blinky application for a target board, or run the the application within Docker Container.

Building the Application

To build the simulated blinky application, run `newt build my_blinky_sim`:

```
$ newt build my_blinky_sim
Building target targets/my_blinky_sim
Compiling repos/apache-mynewt-core/hw/hal/src/hal_common.c
Compiling repos/apache-mynewt-core/hw/drivers/uart/src/uart.c
Compiling repos/apache-mynewt-core/hw/hal/src/hal_flash.c
Compiling repos/apache-mynewt-core/hw/bsp/native/src/hal_bsp.c
Compiling repos/apache-mynewt-core/hw/drivers/uart/uart_hal/src/uart_hal.c
Compiling apps/blinky/src/main.c

...
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/my_blinky_sim/app/apps/blinky/blinky.elf
Target successfully built: targets/my_blinky_sim
```

Running the Blinky Application

You can run the simulated version of your project and see the simulated LED blink.

If you natively install the toolchain execute the binary directly:

```
$ ./bin/targets/my_blinky_sim/app/apps/blinky/blinky.elf
hal_gpio set pin 1 to 0
```

If you are using newt docker, use `newt run` to run the simulated binary. Remember to use the `newt` wrapper script when doing that.

```
$ newt run my_blinky_sim
Loading app image into slot 1
...
Debugging ~/dev/myproj/bin/targets/my_blinky_sim/app/apps/blinky/blinky.elf
...
Reading symbols from /bin/targets/my_blinky_sim/app/apps/blinky/blinky.elf...done.
(gdb)
```

Type `r` at the `(gdb)` prompt to run the project. You will see an output indicating that the `hal_gpio` pin is toggling between 1 and 0 in a simulated blink.

2.3.5 Exploring other Mynewt OS Features

Congratulations, you have created your first project! The blinky application is not terribly exciting when it is run in the simulator, as there is no LED to blink. Apache Mynewt has a lot more functionality than just running simulated applications. It provides all the features you'll need to cross-compile your application, run it on real hardware and develop a full featured application.

If you're interested in learning more, a good next step is to dig in to one of the [Tutorials](#) and get a Mynewt project running on real hardware.

Happy Hacking!

2.4 Using the Serial Port with Mynewt OS

Some of the projects and tutorials here will allow you to use a serial port to interact with your Mynewt project. While most modern PCs and laptops no longer have a true serial port, almost all can use their USB ports as serial ports.

This page will show you how to connect to some of the development boards we use via a serial port.

- [Using the USB port on the nRF52DK](#)
- [Using a USB to Serial Breakout Board](#)
 - [Setup FT232H](#)
 - [Setup Nordic Semiconductor NRF52DK](#)
 - [Setup Arduino M0 Pro](#)
 - [Setup Serial Communications](#)

- * *Example for Mac OS and Linux Platforms*
- * *Example for Windows Platforms*

2.4.1 Using the USB port on the nRF52DK

The nRF52DK can communicate via a serial port directly through its USB port. By default, it uses the UART onboard, so if you need to use the UART for other purposes, consider using a USB<→Serial converter or the Segger RTT Console instead, instructions provided below.

To see your device, check the USB devices that are connected to your computer (macOS):

```
$ ls -l /dev/*USB*
crw-rw-rw- 1 <user> None 117, 5 May 9 04:24 /dev/cu.usbmodem1411
crw-rw-rw- 1 <user> None 117, 5 May 9 04:24 /dev/tty.usbmodem1411
$
```

The `/dev/cu.usbmodem####` and `/dev/tty.usbmodem####` represent your device. To get console access, you can either use the screen command or Minicom:

```
$ minicom -D /dev/tty.usbmodem1411

Welcome to minicom 2.7

OPTIONS:
Compiled on Apr 26 2018, 13:21:44
Port /dev/tty.usbmodem1411, 11:12:13

Press Meta-Z for help on special keys
```

2.4.2 Using a USB to Serial Breakout Board

If you need to use an external USB to Serial converter, this guide will show you how to set it up. The development boards covered here are:

- Nordic Semiconductor nRF52dk
- Arduino M0 Pro

For this tutorial, we'll be using the [AdaFruit FT232H Breakout Board](#), but almost any similar board should work just as well. You will also need Minicom or a similar Serial communications application. We'll show you how to use the screen command built in to Mac OS X, but later tutorials will also show Minicom setup.

Setup FT232H

This is a great board because it's so easy to set up, and it can do Serial UART, SPI, I2C and GPIO as well. There's full documentation on the board [here](#) but we're only covering the wiring for the Serial UART.

Start by connecting a jumper wire to Pin D0. This will be the UART Tx pin, which we'll then connect to the Rx pin on the Development Board.

Next connect a jumper wire to pin D1. This will be the UART Rx pin, which we'll connect to the Tx pin on the development board.

Finally connect a jumper wire to the GND pin.

It should look like this:



Fig. 1: FT232H Wiring

Setup Nordic Semiconductor NRF52DK

On the NRF52DK developer kit board, the Rx pin is P0.08, so you'll attach your jumper wire from the Tx pin (D0) of the FT232H board [here](#).

The TX Pin is pin P0.06, so you'll attach the jumper wire from the Rx Pin (D1) on the FT232H board [here](#).

Finally, the GND wire should go to the GND Pin on the NRF52DK. When you're done, your wiring should look like this:

Setup Arduino M0 Pro

On the Arduino M0 Pro, the Tx and Rx pins are clearly labeled as such, as is the GND pin. Just make sure you wire Rx from the FT232H to TX on the M0 Pro, and vice-versa.

Your Arduino M0 Pro should look like this:



Fig. 2: NRF52DK Wiring



Fig. 3: Arduino M0 Pro Wiring

Setup Serial Communications

You will need to know the serial port to connect to and use a terminal program to connect to the board.

Example for Mac OS and Linux Platforms

First check what USB devices are already connected before connecting the FT232H board to your computer. The ports are listed in the `/dev` directory and the format of the port name is platform dependent:

- Mac OS uses the format `tty.usbserial-<some identifier>`.
- Linux uses the format `TTYUSB<N>`, where N is a number. For example, `TTYUSB2`.

This example is run on a Mac OS system.

Check what USB devices are already connected:

```
$ ls -la /dev/*usb*
0 crw-rw-rw- 1 root wheel 20, 63 Nov 23 11:13 /dev/cu.usbmodem401322
0 crw-rw-rw- 1 root wheel 20, 62 Nov 23 11:13 /dev/tty.usbmodem401322
$
```

Plug in the FT232H board and check the ports again:

```
$ ls -la /dev/*usb*
0 crw-rw-rw- 1 root wheel 20, 63 Nov 23 11:13 /dev/cu.usbmodem401322
0 crw-rw-rw- 1 root wheel 20, 65 Nov 23 11:26 /dev/cu.usbserial-0020124
0 crw-rw-rw- 1 root wheel 20, 62 Nov 23 11:13 /dev/tty.usbmodem401322
0 crw-rw-rw- 1 root wheel 20, 64 Nov 23 11:26 /dev/tty.usbserial-0020124
$
```

The FT232H is connected to `/dev/tty.usbserial-0020124` (The number after `tty.usbserial` will be different on your machine.) Use the screen command to connect to the board:

```
$ screen /dev/tty.usbserial-0020124 115200
```

To exit out of `screen` you'll type `control-A` followed by `control-\` and you'll be back to a terminal prompt.

You can also use minicom:

```
$ minicom -D /dev/tty.usbserial-0020124

Welcome to minicom 2.7

OPTIONS:
Compiled on Nov 24 2015, 16:14:21.
Port /dev/tty.usbserial-0020124, 09:57:17

Press Meta-Z for help on special keys
```

If there's no Mynewt app running, or the Mynewt app doesn't have the Shell and Console enabled, you won't see anything there, but you can always refer back to this page from later tutorials if you need to.

Example for Windows Platforms

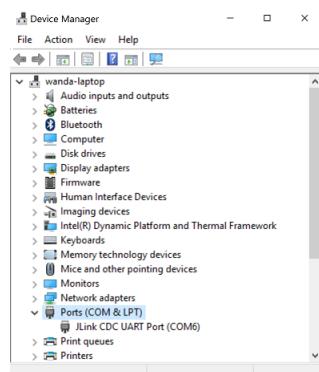
First check what USB devices are already connected before connecting the FT232H board to your computer. You can locate the ports from a MinGW terminal or use the Windows Device Manager.

On a MinGW terminal, the ports are listed in the /dev directory and the format of the port name is `ttyS<N>` where N is a number. You must map the port name to a Windows COM port: `/dev/ttyS<N>` maps to `COM<N+1>`. For example, `/dev/ttyS2` maps to `COM3`.

Check what USB devices are already connected:

```
$ ls -l /dev/ttyS*
crw-rw-rw- 1 <user> None 117, 5 May 9 04:24 /dev/ttyS5
$
```

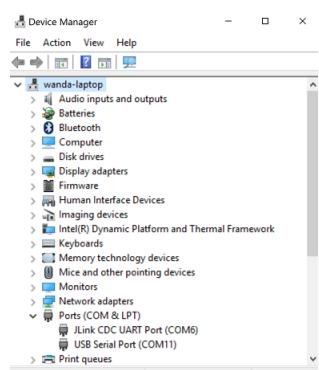
`/dev/ttyS5` maps to the Windows COM6 port. You can run Windows Device Manager to confirm:



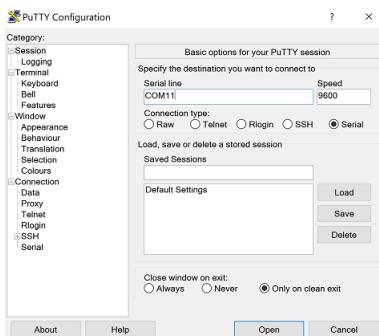
Plug in the FT232H board and check the ports again:

```
$ ls -l /dev/ttyS*
ls -l /dev/ttyS*
crw-rw-rw- 1 <user> None 117, 10 May 9 04:55 /dev/ttyS10
crw-rw-rw- 1 <user> None 117, 5 May 9 04:55 /dev/ttyS5
$
```

The FT232H board is connected to port `/dev/ttyS10` (or COM11):



We use the PuTTY terminal application to connect to the board on the COM11 port:



Press Open and you should get a terminal screen titled “COM11 - PuTTY”

If there’s no Mynewt app running, or the Mynewt app doesn’t have the Shell and Console enabled, you won’t see anything there, but you can always refer back to this page from later tutorials if you need to.

Now that you know how to communicate with your Mynewt application, let’s move on to creating one!

2.5 Debugging Mynewt

If you run into issues with Mynewt applications, you may need to use the debugger. This page will show you how to set up and use the debugger in Mynewt.

- *Prerequisites*
- *Starting the Debugger*

2.5.1 Prerequisites

- You have installed the Newt tool and toolchains on your machine. Visit the [Native Installation](#) page to get started.
- You have run through one of the tutorials, created a project, or have an existing one.
- Familiarity with GDB; for more information refer to the [GDB documentation](#).

2.5.2 Starting the Debugger

Mynewt uses GDB to debug applications. To open a debugger session for Mynewt, you can run either of the following newt commands:

- `newt debug <target_name>`
- `newt run <target_name> <version_number>`

The `newt debug` command will start the debugger for the specified target. That target should be running on the device you are debugging on. For more information on the command, take a look at the [command structure in the Newt Tool Guide](#).

The `newt run` command also starts the debugger, but will also build the target, create an image with the specified version number, and load it onto the board. More details on its usage and syntax are on the [Newt Tool command guide](#).

Starting the debugger will bring up the GDB session in the console:

```
~/test/1.5 $ newt debug pca40blinky
[/Users/dlee/test/1.5/repos/apache-mynewt-core/hw/bsp/nordic_pca10040/nordic_pca10040_
↳debug.sh /Users/dlee/test/1.5/repos/apache-mynewt-core/hw/bsp/nordic_pca10040 bin/
↳targets/pca40blinky/app/apps/blinky/blinky]

Debugging bin/targets/pca40blinky/app/apps/blinky/blinky.elf
GNU gdb (GNU Tools for Arm Embedded Processors 7-2018-q2-update) 8.1.0.20180315-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-apple-darwin10 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bin/targets/pca40blinky/app/apps/blinky/blinky.elf...done.
os_tick_idle (ticks=128)
at repos/apache-mynewt-core/hw/mcu/nordic/nrf52xxx/src/hal_os_tick.c:164
164      if (ticks > 0) {
(gdb)
```

CONCEPTS

This page is meant to introduce you to some of the concepts inherent to the Apache Mynewt Operating System, and *Newt* the tool that stitches a project built on Apache Mynewt together.

- *Project*
- *Package*
- *Target*
- *Configuration*

3.1 Project

The project is the base directory of your embedded software tree. It is a workspace that contains a logical collection of source code, for one or more of your applications. A project consists of the following items:

- Project Definition: defines project level dependencies, and parameters (located in `project.yml`)
- Packages

Package are described in detail in the section below.

Here is an example project definition file from the default Apache Mynewt project:

```
# project.yml
# <snip>
project.name: "my_project"

project.repositories:
    - apache-mynewt-core

# Use github's distribution mechanism for core ASF libraries.
# This provides mirroring automatically for us.
#
repository.apache-mynewt-core:
    type: github
    vers: 1-latest
    user: apache
    repo: mynewt-core
```

A couple of things to note in the project definition:

- `project.repositories`: Defines the remote repositories that this project relies upon.
- `repository.apache-mynewt-core`: Defines the repository information for the `apache-mynewt-core` repository.
- `vers=1-latest`: Defines the repository version. This string will use the latest stable version in the ‘Master’ github branch. To use the latest version in the master branch, just change it to `vers=0-dev`. Note that this branch might not be stable.

Repositories are versioned collections of packages.

Projects can rely on remote repositories for functionality, and the newt tool will resolve those remote repositories, and download the correct version into your local source tree. Newly fetched repositories are put in the `repos` directory of your project, and can be referenced throughout the system by using the `@` specifier.

By default, the `@apache-mynewt-core` repository is included in every project. Apache Mynewt Core contains all the base functionality of the Apache Mynewt Operating System, including the Real Time Kernel, Bluetooth Networking Stack, Flash File System, Console, Shell and Bootloader.

NOTE: Any project can be converted into a repository by providing it with a `repository.yml` file and putting it up onto Github. More information about repositories can be found in the Newt documentation.

3.2 Package

A package is a collection items that form a fundamental unit in the Mynewt Operating System. Packages can be:

- Applications
- Libraries
- Compiler definitions
- Targets

A package is identified by having a `pkg.yml` file in it’s base directory. Here is a sample `pkg.yml` file for the `blinky` applicaton:

```
# pkg.yml
# <snip>
pkg.name: apps/blinky
pkg.type: app
pkg.description: Basic example application which blinks an LED.
pkg.author: "Apache Mynewt <dev@mynewt.apache.org>"
pkg.homepage: "http://mynewt.apache.org/"
pkg.keywords:

pkg.deps:
  - "@apache-mynewt-core/libs/os"
  - "@apache-mynewt-core/hw/hal"
  - "@apache-mynewt-core/libs/console/full"
```

Packages have a few features worth noting:

- Dependencies: Packages can rely upon other packages, and when they do they will inherit their functionality (header files, library definitions, etc.)
- APIs: Packages can export named APIs, and they can require that certain APIs be present, in order to compile.

Everything that newt knows about within a project’s directory is a package. This makes it very clean and easy to write re-usable components, which can describe their Dependencies and APIs to the rest of the system.

3.3 Target

A target in Apache Mynewt is very similar to a target in *make*. It is the collection of parameters that must be passed to Newt in order to generate a reproducible build. A target represents the top of the build tree, and any packages or parameters specified at the target level, cascade down to all dependencies.

Targets are also packages, and are stored in the `targets/` directory at the base of your project. Most targets consist of:

- `app`: The application to build.
- `bsp`: The board support package to combine with that application
- `build_profile`: Either `debug` or `optimized`.

Targets can also have additional items specified, including:

- `aflags`: Any additional assembler flags you might want to specify to the build.
- `cflags`: Any additional compiler flags you might want to specify to the build.
- `lflags`: Any additional linker flags you might want to specify to the build.

In order to create and manipulate targets, the `newt` tool offers a set of helper commands, you can find more information about these by issuing:

```
$ newt target
Usage:
  newt target [flags]
  newt target [command]

Available Commands:
  amend      Add, change, or delete values for multi-value target variables
  cmake
  config     View or populate a target's system configuration
  copy       Copy target
  create     Create a target
  delete    Delete target
  dep        View target's dependency graph
  revdep    View target's reverse-dependency graph
  set        Set target configuration variable
  show       View target configuration variables

Global Flags:
  -h, --help           Help for newt commands
  -j, --jobs int       Number of concurrent build jobs (default 2)
  -l, --loglevel string Log level (default "WARN")
  -o, --outfile string Filename to tee output to
  -q, --quiet          Be quiet; only display error output
  -s, --silent         Be silent; don't output anything
  -v, --verbose        Enable verbose output when executing commands

Use "newt target [command] --help" for more information about a command.
```

3.4 Configuration

Additional help topics:

```
$ newt target config show <target-name>
...
* PACKAGE: sys/stats
  * Setting: STATS_CLI
    * Description: Expose the "stat" shell command.
    * Value: 0
  * Setting: STATS_NAMES
    * Description: Include and report the textual name of each statistic.
    * Value: 0
  * Setting: STATS_NEWTMGR
    * Description: Expose the "stat" newtmgr command.
    * Value: 0
...
$
```

TUTORIALS

If the introduction to Mynewt has piqued your interest and you want to familiarize yourself with some of its functionality, this series of tutorials is for you. The lessons are aimed at the beginner.

The full list of tutorials can be seen in the navigation bar on the left. New ones are being constantly added and will show up there automatically.

- *Prerequisites*
- *Tutorial categories*

4.1 Prerequisites

- You have installed Docker container of Newt tool and toolchains or you have installed them natively on your machine
- You have created a new project space (directory structure) and populated it with the core code repository (apache-mynewt-core) or know how to as explained in [Creating Your First Mynewt Project](#).
- You have at least one of the supported development boards:
 - [Arduino Zero](#)
 - [Olimex](#)
 - [nRF52](#)

The Nordic nrf52 developer kit supports Bluetooth Low Energy. We are always looking to add new hardware to the list, so if you want to develop the required Board Support Package (bsp) and/or Hardware Abstraction Layer (HAL) for a new board, you can look [Description](#) to get started.

4.2 Tutorial categories

The tutorials fall into a few broad categories. Some examples in each category are listed below.

- Making an LED Blink (the “Hello World” equivalent in the electronics world)
 - [Blinky, your “Hello World!”, on Arduino Zero](#)
 - [Blinky, your “Hello World!”, on Olimex](#)
 - [Blinky, your “Hello World!”, on a nRF52 Development Kit](#) **Note:** This supports BLE.

- *Blinky, your “Hello World!”, on RedBear Nano 2*
- Navigating the Code and Adding Functionality
 - *Adding More Repositories to Your Project*
 - *Adding a Unit Test For a Package*
- Using Newtmgr
 - *Enabling Remote Communication With a Device Running Mynewt OS*
- Bluetooth Low Energy
 - *Building a Bare Bones BLE Application*
 - *Building a BLE iBeacon Application*
- OS Fundamentals
 - *Events and Event Queues*
 - *Task and Priority Management*
- Remote Device Management
 - *Enabling Newt Manager in Any App*
 - *Upgrading an Image Over-The-Air*
- Sensors
 - *Enabling an Off-Board Sensor in an Existing Application*
 - *Developing an Application for an Onboard Sensor*
 - *Enabling OIC Sensor Data Monitoring*
- Tooling
 - *SEGGER RTT*
 - *SEGGER SystemView*

Send us an email on the dev@ mailing list if you have comments or suggestions! If you haven't joined the mailing list, you will find the links [here](#).

4.3 Project Blinky

The set of Blinky tutorials show you how to create, build, and run a “Hello World” application that blinks a LED on the various target boards that Mynewt supports. The tutorials use the same Blinky application from the [Creating Your First Mynewt Project](#) tutorial.

- *Objective*
- *Available Tutorials*
- *Prerequisites*
- *Overview of Steps*

4.3.1 Objective

Learn how to use packages from a default application repository of Mynewt to build your first *Hello World* application (Blinky) on a target board. Once built using the *newt* tool, this application will blink a LED light on the target board.

4.3.2 Available Tutorials

Tutorials are available for the following boards:

- *Blinky, your “Hello World!”, on Arduino Zero*
- *Blinky, your “Hello World!”, on Arduino Primo*
- *Blinky, your “Hello World!”, on Olimex*
- *Blinky, your “Hello World!”, on a nRF52 Development Kit*
- *Blinky, your “Hello World!”, on a PineTime smartwatch*
- *Blinky, your “Hello World!”, on RedBear Nano 2*
- *Blinky, your “Hello World!”, on STM32F4-Discovery*
- *Blinky, your “Hello World!”, on STM32F303 Discovery*
- *Pin Wheel Modifications to “Blinky” on STM32F3 Discovery*

We also have a tutorial that shows you how to add *Enabling The Console and Shell for Blinky*.

4.3.3 Prerequisites

Ensure that you meet the following prerequisites before continuing with one of the tutorials.

- Have Internet connectivity to fetch remote Mynewt components.
- Have a computer to build a Mynewt application and connect to the board over USB.
- Have a Micro-USB cable to connect the board and the computer.
- Install the newt tool and toolchains (See *Setup & Get Started*).
- Read the Mynewt OS *Concepts* section.
- Create a project space (directory structure) and populate it with the core code repository (apache-mynewt-core) or know how to as explained in *Creating Your First Mynewt Project*.

4.3.4 Overview of Steps

These are the general steps to create, load and run the Blinky application on your board:

- Create a project.
- Define the bootloader and Blinky application targets for the board.
- Build the bootloader target.
- Build the Blinky application target and create an application image.
- Connect to the board.
- Load the bootloader onto the board.
- Load the Blinky application image onto the board.

- See the LED on your board blink.

After you try the Blinky application on your boards, checkout out other tutorials to enable additional functionality such as [remote comms](#) on the current board. If you have BLE (Bluetooth Low Energy) chip (e.g. nRF52) on your board, you can try turning it into an [iBeacon](#) or [Eddystone Beacon](#)!

If you see anything missing or want to send us feedback, please sign up for appropriate mailing lists on our [Community Page](#).

4.3.5 Blinky, your “Hello World!”, on Arduino Zero

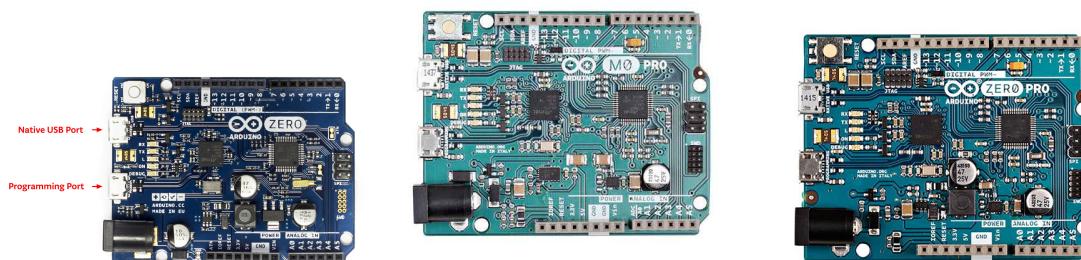
This tutorial shows you how to create, build and run the Blinky application on an Arduino Zero board.

- [Prerequisites](#)
- [Create a Project](#)
- [Fetch External Packages](#)
- [Create a Target for the Blinky Application](#)
- [Build the Bootloader](#)
- [Build the Blinky Application](#)
- [Connect to the Board](#)
- [Load the Bootloader onto the Board](#)
- [Run the Blinky Application](#)

Prerequisites

- Meet the prerequisites listed in [Project Blinky](#).
- Have an Arduino Zero board. Note: There are many flavors of Arduino. Make sure you are using an Arduino Zero. See below for the versions of Arduino Zero that are compatible with this tutorial.
- Install the [OpenOCD debugger](#).

This tutorial uses the Arduino Zero Pro board. The tutorial has been tested on the following three Arduino Zero boards - Zero, M0 Pro, and Zero-Pro.



Mynewt has not been tested on Arduino M0 which has no internal debugger support.

Create a Project

Create a new project if you do not have an existing one. You can skip this step and proceed to [Fetch External Packages](#) if you already created a project.

Run the following commands to create a new project:

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new myproj
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in myproj...
Project myproj successfully created.
$ cd myproj
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
apache-mynewt-core successfully upgraded to version 1.7.0
$
```

Fetch External Packages

Mynewt uses source code provided directly from the chip manufacturer for low level operations. Sometimes this code is licensed only for the specific manufacturer of the chipset and cannot live in the Apache Mynewt repository. That happens to be the case for the Arduino Zero board which uses Atmel SAMD21. Runtime's github repository hosts such external third-party packages and the newt tool can fetch them.

To fetch the package with MCU support for Atmel SAMD21 for Arduino Zero from the Runtime git repository, you need to add the repository to the `project.yml` file in your base project directory.

Here is an example `project.yml` file with the Arduino Zero repository added. The sections with `mynewt_arduino_zero` that need to be added to your project file are highlighted.

Note: On Windows platforms: You need to set `vers` to `0-dev` and use the latest master branch for both repositories.

```
# project.yml
project.name: "my_project"

project.repositories:
    - apache-mynewt-core
    - mynewt_arduino_zero

repository.apache-mynewt-core:
    type: github
    vers: 1-latest
    user: apache
    repo: mynewt-core

repository.mynewt_arduino_zero:
    type: github
    vers: 1-latest
    user: runtimeco
    repo: mynewt_arduino_zero
```

Install the project dependencies using the `newt upgrade` command (you can specify `-v` for verbose output):

```
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
Downloading repository mynewt_arduino_zero (commit: master) ...
apache-mynewt-core successfully upgraded to version 1.7.0
mynewt_arduino_zero successfully upgraded to version 1.7.0
$
```

You need to create two targets for the Arduino Zero Pro board, one for the bootloader and one for the Blinky application.

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `arduino_boot`.

```
$ newt target create arduino_boot
$ newt target set arduino_boot bsp=@mynewt_arduino_zero/hw/bsp/arduino_zero
$ newt target set arduino_boot app=@mcuboot/boot/mynewt
$ newt target set arduino_boot build_profile=optimized
Target targets/arduino_boot successfully set target.build_profile to optimized
$ newt target set arduino_boot syscfg=BSP_ARDUINO_ZERO_PRO=1
Target targets/arduino_boot successfully set target.syscfg to BSP_ARDUINO_ZERO_PRO=1
$
```

Note: If you have an Arduino Zero instead of an Arduino Zero Pro or Arduino M0 Pro board, replace `BSP_ARDUINO_ZERO_PRO` with `BSP_ARDUINO_ZERO` in the last `newt target set` command.

These commands perform the following:

- Create a target named `arduino_boot` for the Arduino Zero Bootloader.
- Set the application for the `arduino_boot` target to the default MCUBoot bootloader (`@mcuboot/boot/mynewt`)
- Set the board support package for the target to `@mynewt_arduino_zero/hw/bsp/arduino_zero`. This is a reference to the downloaded Arduino Zero support from Github.
- Use the “optimized” build profile for the `arduino_boot` target. This instructs Newt to generate smaller and more efficient code for this target. This setting is necessary due to the bootloader’s strict size constraints.
- Sets the system configuration setting for Board Support Package to support the Arduino Zero Pro.

See the [Concepts](#) for more information on setting options.

Create a Target for the Blinky Application

Run the following `newt target` commands to create the Blinky application target. We name the application target `arduino_blinky`.

```
$ newt target create arduino_blinky
Target targets/arduino_blinky successfully created
$ newt target set arduino_blinky app=apps/blinky
Target targets/arduino_blinky successfully set target.app to apps/blinky
$ newt target set arduino_blinky bsp=@mynewt_arduino_zero/hw/bsp/arduino_zero
Target targets/arduino_blinky successfully set target.bsp to @mynewt_arduino_zero/hw/bsp/
 ↵arduino_zero
$ newt target set arduino_blinky build_profile=debug
Target targets/arduino_blinky successfully set target.build_profile to debug
$ newt target set arduino_blinky syscfg=BSP_ARDUINO_ZERO_PRO=1
Target targets/arduino_boot successfully set target.syscfg to BSP_ARDUINO_ZERO_PRO=1
$
```

Note: If you have an Arduino Zero instead of a Arduino Zero Pro board, replace BSP_ARDUINO_ZERO_PRO with BSP_ARDUINO_ZERO in the last `newt target set` command.

Build the Bootloader

Run the `newt build arduino_boot` command to build the bootloader:

```
$ newt build arduino_boot
Building target targets/arduino_boot
Compiling bin/targets/arduino_boot/generated/src/arduino_boot-sysinit-app.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling bin/targets/arduino_boot/generated/src/arduino_boot-sysflash.c
Compiling repos/mcuboot/boot/bootutil/src/image_validate.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/mcuboot/boot/mynewt/src/main.c
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/arc4.c
Compiling repos/mcuboot/boot/bootutil/src/loader.c
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/aes.c

....
```

Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/arduino_boot/app/boot/mynewt/mynewt.elf
Target successfully built: targets/arduino_boot

Build the Blinky Application

Run the `newt build arduino_blinky` command to build the Blinky application image:

```
$ newt build arduino_blinky
Building target targets/arduino_blinky
Compiling repos/apache-mynewt-core/hw/hal/src/hal_flash.c
Compiling apps/blinky/src/main.c
Compiling repos/mynewt_arduino_zero/hw/mcu/atmel/samd21xx/src/sam0/drivers/i2s/i2s.c
Compiling repos/mynewt_arduino_zero/hw/bsp/arduino_zero/src/hal_bsp.c
Compiling repos/mynewt_arduino_zero/hw/mcu/atmel/samd21xx/src/sam0/drivers/i2s/i2s_
↪callback.c
Compiling repos/mynewt_arduino_zero/hw/mcu/atmel/samd21xx/src/sam0/drivers/nvm/nvm.c

...
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/arduino_blinky/app/apps/blinky/blinky.elf
Target successfully built: targets/arduino_blinky
```

Connect to the Board

Connect your computer to the Arduino Zero (from now on we'll call this the target) with a Micro-USB cable through the Programming Port as shown below. Mynewt will load the image onto the board and debug the target through this port. You should see a green LED come on that indicates the board has power.

No external debugger is required. The Arduino Zero comes with an internal debugger that can be accessed by Mynewt.

The images below show the Arduino Zero Programming Port.



Load the Bootloader onto the Board

Run the `newt load arduino_boot` command to load the bootloader onto the board:

```
$ newt load arduino_boot
Loading bootloader
$
```

The bootloader is loaded onto your board successfully when the `newt load` command returns to the command prompt after the `Loading bootloader` status message. You can proceed to load and run your Blinky application image (See [Run the Blinky Application](#)).

If the `newt load` command outputs the following error messages, you will need to erase the board.

```
$ newt load arduino_boot -v
Loading bootloader
Error: Downloading ~/dev/myproj/bin/targets/arduino_boot/boot/mynewt/mynewt.elf.bin to ↵
    ↵0x0
Open On-Chip Debugger 0.9.0 (2015-11-15-05:39)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'swd'
adapter speed: 500 kHz
adapter_nsrst_delay: 100
cortex_m reset_config sysresetreq
Info : CMSIS-DAP: SWD Supported
Info : CMSIS-DAP: JTAG Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 01.1F.0118
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 500 kHz
Info : SWD IDCODE 0x0bc11477
```

(continues on next page)

(continued from previous page)

```
Info : at91samd21g18.cpu: hardware has 4 breakpoints, 2 watchpoints
Error: Target not halted
```

To erase your board, start a debug session and enter the highlighted commands at the (gdb) prompts:

Note: On Windows, openocd and gdb are started in separate Windows Command Prompt terminals, and the terminals are automatically closed when you quit gdb. In addition, the output of openocd is logged to the openocd.log file in your project's base directory instead of the terminal.

```
$ newt debug arduino_blinky
(gdb) mon at91samd chip-erase
chip erased
chip erased
(gdb) x/32wx 0
0x0: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
0x10: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
0x20: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
0x30: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
0x40: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
0x50: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
0x60: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
0x70: 0xffffffff 0xffffffff 0xffffffff 0xffffffff
(gdb) q
```

Run the `newt load arduino_boot` command again after erasing the board.

⚠️ Warning

Reminder if you are using Docker: When working with actual hardware, remember that each board has an ID. If you swap boards and do not refresh the USB Device Filter on the VirtualBox UI, the ID might be stale and the Docker instance may not be able to see the board correctly. For example, you may see an error message like `Error: unable to find CMSIS-DAP device` when you try to load or run an image on the board. In that case, you need to click on the USB link in VirtualBox UI, remove the existing USB Device Filter (e.g. “Atmel Corp. EDBG CMSIS-DAP[0101]”) by clicking on the “Removes selected USB filter” button, and add a new filter by clicking on the “Adds new USB filter” button.

Run the Blinky Application

After you load the bootloader successfully onto your board, you can load and run the Blinky application.

Run the `newt run arduino_blinky 1.0.0` command to build the `arduino_blinky` target (if necessary), create an image with version 1.0.0, load the image onto the board, and start a debugger session.

Note The output of the debug session below is for Mac OS and Linux platforms. On Windows, openocd and gdb are started in separate Windows Command Prompt terminals. The output of openocd is logged to the `openocd.log` file in your project's base directory and not to the terminal. The openocd and gdb terminals will close automatically when you quit gdb.

```
$ newt run arduino_blinky 1.0.0
App image successfully generated: ~/dev/myproj/bin/targets/arduino_blinky/app/apps/blinky/
↳ blinky.img
Loading app image into slot 1
```

(continues on next page)

(continued from previous page)

```
[~/dev/myproj/repos/mynewt_arduino_zero/hw/bsp/arduino_zero/arduino_zero_debug.sh ~/dev/
 ↵myproj/repos/mynewt_arduino_zero/hw/bsp/arduino_zero ~/dev/myproj/bin/targets/arduino_
 ↵blinky/app/apps/blinky/blinky]
Open On-Chip Debugger 0.9.0 (2015-11-15-13:10)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'swd'
adapter speed: 500 kHz
adapter_nsrst_delay: 100
cortex_m reset_config sysresetreq
Info : CMSIS-DAP: SWD Supported
Info : CMSIS-DAP: JTAG Supported
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : CMSIS-DAP: FW Version = 01.1F.0118
Info : SWCLK/TCK = 1 SWDIO/TMS = 1 TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 500 kHz
Info : SWD IDCODE 0x0bc11477
Info : at91samd21g18.cpu: hardware has 4 breakpoints, 2 watchpoints
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x21000000 pc: 0x0000fcfa6 psp: 0x20002408
GNU gdb (GNU Tools for ARM Embedded Processors) 7.8.0.20150604-cvs
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-apple-darwin10 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ~/dev/myproj/bin/targets/arduino_blinky/app/apps/blinky/blinky.elf..
 ↵.(no debugging symbols found)...done.
Info : accepting 'gdb' connection on tcp/3333
Info : SAMD MCU: SAMD21G18A (256KB Flash, 32KB RAM)
0x0000fcfa6 in os_tick_idle ()
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x21000000 pc: 0x000000b8 msp: 0x20008000
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x21000000 pc: 0x000000b8 msp: 0x20008000
(gdb) r
The "remote" target does not support "run". Try "help target" or "continue".
(gdb) c
Continuing.
```

NOTE: The 1.0.0 is the version number to assign to the image. You may assign an arbitrary version number. If you are not providing remote upgrade, and are just developing locally, you can provide 1.0.0 for every image version.

If you want the image to run without the debugger connected, simply quit the debugger and restart the board. The image you programmed will come and run on the Arduino on next boot!

You should see the LED blink!

4.3.6 Blinky, your “Hello World!”, on Arduino Primo

This tutorial shows you how to create, build, and run the Blinky application on an Arduino Primo board.

- *Prerequisites*
 - *Option 1*
 - *Option 2*
- *Create a Project*
- *Create the Targets*
- *Build the Target Executables*
- *Sign and Create the Blinky Application Image*
- *Connect to the Board*
- *Load the Bootloader*
- *Load the Blinky Application Image*
- *Erase Flash*

Note that the Mynewt OS will run on the nRF52 chip in the Arduino Primo board. However, the board support package for the Arduino Primo is different from the nRF52 dev kit board support package.

Prerequisites

- Meet the the prerequisites listed in [Project Blinky](#).
- Have an Arduino Primo board.
- Install a debugger. Choose one of the two options below: Option 1 requires additional hardware but very easy to set up.

Option 1

- Segger J-Link Debug Probe - any model (this tutorial has been tested with J-Link EDU and J-Link Pro)
- J-Link 9 pin Cortex-M Adapter that allows JTAG, SWD and SWO connections between J-Link and Cortex M based target hardware systems
- Install the [Segger JLINK Software](#) and documentation pack.

Option 2

This board requires a patch version of OpenOCD 0.10.0 that is in development. See [Install OpenOCD](#) instructions to install it if you do not have this version installed.

You can now use openocd to upload to Arduino Primo board via the USB port itself.

Create a Project

Create a new project if you do not have an existing one. You can skip this step and proceed to [Create the Targets](#) if you already created a project.

Run the following commands to create a new project:

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new myproj
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in myproj...
Project myproj successfully created.
$ cd myproj
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
Downloading repository mynewt-nimble (commit: master) ...
Downloading repository mcuboot (commit: master) ...
Downloading repository mynewt-mcumgr (commit: master) ...
Making the following changes to the project:
apache-mynewt-core successfully upgraded to version 1.7.0
apache-mynewt-nimble successfully upgraded to version 1.2.0
mcuboot successfully upgraded to version 1.3.1
$
```

Create the Targets

Create two targets for the Arduino Primo board - one for the bootloader and one for the Blinky application.

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `primo_boot`.

```
$ newt target create primo_boot
$ newt target set primo_boot app=@mcuboot/boot/mynewt bsp=@apache-mynewt-core/hw/bsp/
  ↵arduino_primo_nrf52 build_profile=optimized
```

Run the following `newt target` commands to create a target for the Blinky application. We name the target `primoblinky`.

```
$ newt target create primoblinky
$ newt target set primoblinky app=apps/blinky bsp=@apache-mynewt-core/hw/bsp/arduino_
→primo_nrf52 build_profile=debug
```

If you are using openocd, run the following `newt target set` commands:

```
$ newt target set primoblinky syscfg=OPENOCD_DEBUG=1
$ newt target set primo_boot syscfg=OPENOCD_DEBUG=1
```

You can run the `newt target show` command to verify the target settings:

```
$ newt target show
targets/my_blinky_sim
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/native
    build_profile=debug
targets/primo_boot
    app=@mcuboot/boot/mynewt
    bsp=@apache-mynewt-core/hw/bsp/arduino_primo_nrf52
    build_profile=optimized
targets/primoblinky
    app=@apache-mynewt-core/apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/arduino_primo_nrf52
    build_profile=optimized
```

Build the Target Executables

Run the `newt build primo_boot` command to build the bootloader:

```
$ newt build primo_boot
Building target targets/primo_boot
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/aes.c
Compiling repos/mcuboot/boot/mynewt/src/main.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/loader.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c

...
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/primo_boot/app/boot/mynewt/mynewt.elf
Target successfully built: targets/primo_boot
```

Run the `newt build primoblinky` command to build the Blinky application:

```
$ newt build primoblinky
Building target targets/primoblinky
Compiling repos/apache-mynewt-core/hw/drivers/uart/src/uart.c
```

(continues on next page)

(continued from previous page)

```
Assembling repos/apache-mynewt-core/hw/bsp/arduino_primo_nrf52/src/arch/cortex_m4/gcc_
↪startup_nrf52.s
Compiling repos/apache-mynewt-core/hw/bsp/arduino_primo_nrf52/src/sbrk.c
Compiling repos/apache-mynewt-core/hw/cmsis-core/src/cmsis_nvic.c
Assembling repos/apache-mynewt-core/hw/bsp/arduino_primo_nrf52/src/arch/cortex_m4/gcc_
↪startup_nrf52_splits.s
Compiling apps/blinky/src/main.c
Compiling repos/apache-mynewt-core/hw/drivers/uart/uart_bitbang/src/uart_bitbang.c
Compiling repos/apache-mynewt-core/hw/bsp/arduino_primo_nrf52/src/hal_bsp.c

...
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/primoblinky/app/apps/blinky/blinky.elf
Target successfully built: targets/primoblinky
```

Sign and Create the Blinky Application Image

Run the `newt create-image primoblinky 1.0.0` command to create and sign the application image. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image primoblinky 1.0.0
App image successfully generated: ~/dev/myproj/bin/targets/primoblinky/app/apps/blinky/
↪blinky.img
```

Connect to the Board

- Connect a micro USB cable to the Arduino Primo board and to your computer's USB port.
- If you are using the Segger J-Link debug probe, connect the debug probe to the JTAG port on the Primo board using the Jlink 9-pin adapter and cable. Note that there are two JTAG ports on the board. Use the one nearest to the reset button as shown in the picture.

Note: If you are using the OpenOCD debugger, you do not need to attach this connector.

Load the Bootloader

Run the `newt load primo_boot` command to load the bootloader onto the board:

```
$ newt load primo_boot
Loading bootloader
$
```

Note: If you are using OpenOCD on a Windows platform and you get an `unable to find CMSIS-DAP device` error, you will need to download and install the mbed Windows serial port driver from <https://developer.mbed.org/handbook/Windows-serial-configuration>. Follow the instructions from the site to install the driver. Here are some additional notes about the installation:



Fig. 1: J-Link debug probe to Arduino

1. The instructions indicate that the mbed Windows serial port driver is not required for Windows 10. If you are using Windows 10 and get the unable to find CMSIS-DAP device error, we recommend that you install the driver.
2. If the driver installation fails, we recommend that you download and install the Arduino Primo CMSIS-DAP driver. Perform the following steps:
 - Download the [Arduino Primo CMSIS-DAP driver](#) and extract the zip file.
 - Start Device Manager.
 - Select Other Devices > CMSIS-DAP CDC > Properties > Drivers > Update Driver....
 - Select Browse my computer for driver software.
 - Select the Arduino Driver folder where extracted the drivers to (check the include subfolders). Click Next to install the driver.

Run the `newt load primo_boot` command again.

Load the Blinky Application Image

Run the `newt load primoblinky` command to load the Blinky application image onto the board.

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

You should see the orange LED (L13), below the ON LED, on the board blink!

Note: If the LED does not blink, try resetting the board. If that doesn't work, you may have an older version of the Arduino Primo, and will need to change the defined LED blink pin. To change the LED blink pin, go to the Arduino Primo BSP header file in the repos directory (`~/dev/myproj/repos/apache-mynewt-core/hw/bsp/arduino_primoo_nrf52/include/bsp/bsp.h`) and change the `LED_BLINK_PIN` from 20 to 25.

Erase Flash

If you want to erase the flash and load the image again, use JLinkExe and issue the `erase` command when you are using the Jlink debug probe:

Note: On Windows: Run the `jlink` command with the same arguments from a Windows Command Prompt terminal.

```
$ JLinkExe -device nRF52 -speed 4000 -if SWD
SEGGER J-Link Commander V5.12c (Compiled Apr 21 2016 16:05:51)
DLL version V5.12c, compiled Apr 21 2016 16:05:45

Connecting to J-Link via USB...O.K.
Firmware: J-Link OB-SAM3U128-V2-NordicSemi compiled Mar 15 2016 18:03:17
Hardware version: V1.00
S/N: 682863966
VTref = 3.300V

Type "connect" to establish a target connection, '?' for help
J-Link>erase
Cortex-M4 identified.
Erasing device (0;?i?)...
```

(continues on next page)

(continued from previous page)

```
Comparing flash [100%] Done.
Erasing flash [100%] Done.
Verifying flash [100%] Done.
J-Link: Flash download: Total time needed: 0.363s (Prepare: 0.093s, Compare: 0.000s, ↵
→Erase: 0.262s, Program: 0.000s, Verify: 0.000s, Restore: 0.008s)
Erasing done.
J-Link>exit
$
```

If you are using the OpenOCD debugger, run the `newt debug primoblinky` command and issue the highlighted command at the (gdb) prompt:

Note: The output of the debug session below is for Mac OS and Linux platforms. On Windows, openocd and gdb are started in separate Windows Command Prompt terminals, and the terminals are automatically closed when you quit gdb. In addition, the output of openocd is logged to the `openocd.log` file in your project's base directory instead of the terminal.

```
$ newt debug primoblinky
[~/dev/myproj/repos/apache-mynewt-core/hw/bsp/arduino_primo_nrf52/primo_debug.sh ~/dev/
←myproj/repos/apache-mynewt-core/hw/bsp/arduino_primo_nrf52 ~/dev/myproj/bin/targets/
→primoblinky/app/apps/blinky/blinky]
Open On-Chip Debugger 0.10.0-dev-snapshot (2017-03-28-11:24)

...
os_tick_idle (ticks=128)
    at repos/apache-mynewt-core/hw/mcu/nordic/nrf52xxx/src/hal_os_tick.c:200
warning: Source file is more recent than executable.
200     if (ticks > 0) {
(gdb) mon nrf52 mass_erase
```

4.3.7 Blinky, your “Hello World!”, on Olimex

This tutorial shows you how to create, build, and run the Blinky application on an Olimex STM32-E407 board.

- *Prerequisites*
- *Create a Project*
- *Create the Targets*
- *Build the Bootloader*
- *Build the Blinky Application*
- *Sign and Create the Blinky Application Image*
- *Connect to the Board*
- *Load the Bootloader and Blinky Application*

Prerequisites

- Meet the prerequisites listed in [Project Blinky](#).
- Have a STM32-E407 development board from Olimex.
- Have a ARM-USB-TINY-H connector with JTAG interface for debugging ARM microcontrollers (comes with the ribbon cable to hook up to the board)
- Have a USB A-B type cable to connect the debugger to your computer.
- Install the [OpenOCD debugger](#).

Create a Project

Create a new project if you do not have an existing one. You can skip this step and proceed to [Create the Targets](#) if you already created a project.

Run the following commands to create a new project:

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new myproj
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in myproj...
Project myproj successfully created.

$ cd myproj

$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
apache-mynewt-core successfully upgraded to version 1.7.0
$
```

Create the Targets

Create two targets for the Olimex board - one for the bootloader and one for the Blinky application.

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `boot_olimex`.

```
$ newt target create boot_olimex
$ newt target set boot_olimex build_profile=optimized
$ newt target set boot_olimex app=@mcuboot/boot/mynewt
$ newt target set boot_olimex bsp=@apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard
```

Run the following `newt target` commands to create a target for the Blinky application. We name the target `olimex_blinky`.

```
$ newt target create olimex_blinky
$ newt target set olimex_blinky build_profile=debug
$ newt target set olimex_blinky bsp=@apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard
$ newt target set olimex_blinky app=apps/blinky
```

Build the Bootloader

Run the `newt build boot_olimex` command to build the bootloader:

```
$ newt build boot_olimex
Building target targets/boot_olimex
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling bin/targets/boot_olimex/generated/src/boot_olimex-sysflash.c

...
Archiving libc_baselibc.a
Archiving sys_flash_map.a
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/boot_olimex/app/boot/mynewt/mynewt.elf
Target successfully built: targets/boot_olimex
```

Build the Blinky Application

Run the `newt build olimex_blinky` command to build the blinky application:

```
$ newt build olimex_blinky
Building target targets/olimex_blinky
Assembling repos/apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard/src/arch/cortex_m4/
↪ startup_STM32F40x.s
Compiling repos/apache-mynewt-core/hw/drivers/uart/src/uart.c
Compiling repos/apache-mynewt-core/hw/cmsis-core/src/cmsis_nvic.c
Compiling repos/apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard/src/sbrk.c
Compiling apps/blinky/src/main.c
Compiling repos/apache-mynewt-core/hw/drivers/uart/uart_hal/src/uart_hal.c
Compiling repos/apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard/src/hal_bsp.c
Compiling repos/apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard/src/system_
↪ stm32f4xx.c
Compiling repos/apache-mynewt-core/hw/hal/src/hal_common.c
Compiling repos/apache-mynewt-core/hw/hal/src/hal_flash.c

...
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/olimex_blinky/app/apps/blinky/blinky.elf
Target successfully built: targets/olimex_blinky
```

Sign and Create the Blinky Application Image

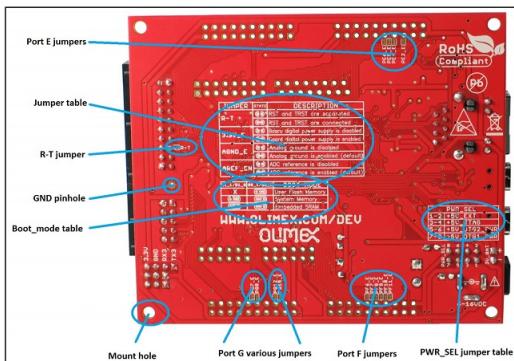
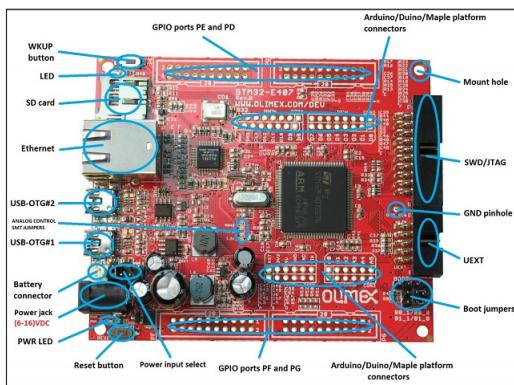
Run the `newt create-image olimex_blinky 1.0.0` command to sign and create an image file for the blinky application. You may assign an arbitrary version (e.g. 1.0.0) number.

```
$ newt create-image olimex_blinky 1.0.0
App image successfully generated: ~/dev/myproj/bin/targets/olimex_blinky/app/apps/blinky/
→ blinky.img
```

Connect to the Board

Configure the board to bootload from flash memory and to use USB-OTG2 for the power source. Refer to the following diagrams to locate the boot jumpers and power input select jumpers on the board.

Note: The labels for the **USB-OTG1** and **USB-OTG2** ports on the diagram are reversed. The port labeled **USB-OTG1** on the diagram is the **USB-OTG2** port and the port labeled **USB-OTG2** on the diagram is the **USB-OTG1** port.



- Locate the boot jumpers on the lower right corner of the board. **B1_1/B1_0** and **B0_1/B0_0** are PTH jumpers to control the boot mode when a bootloader is present. These two jumpers must be moved together. The board searches for the bootloader in three places: User Flash Memory, System Memory or the Embedded SRAM. For this Blinky project, we configure the board to boot from flash by jumpering **B0_0** and **B1_0**. **Note:** The markings on the board may not always be accurate, and you should always refer to the manual for the correct positioning.
- Locate the **Power Input Select** jumpers on the lower left corner of the board. Set the Power Select jumpers to position 5 and 6 to use the USB-OTG2 port for the power source. If you would like to use a different power source, refer to the [OLIMEX STM32-E407 user manual](#) for pin specifications.
- Connect the USB Micro-A cable to the USB-OTG2 port on the board.

- Connect the JTAG connector to the JTAG/SWD interface on the board.
- Connect the USB A-B cable to the ARM-USB-TINY-H connector and your computer.
- Check that the red PWR LED lights up.

Load the Bootloader and Blinky Application

Run the `newt load boot_olimex` command to load the bootloader image onto the board:

```
$ newt load -v boot_olimex
Loading bootloader
Load command: ~/dev/myproj/repos/apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard/
↳ olimex_stm32-e407_devboard_download.sh ~/dev/myproj/repos/apache-mynewt-core/hw/bsp/
↳ olimex_stm32-e407_devboard ~/dev/myproj/bin/targets/boot_olimex/app/boot/mynewt/mynewt
Successfully loaded image.
```

Note: If you are using Windows and get a `no device found` error, you will need to install the usb driver. Download [Zadig](#) and run it:

- Select Options > List All Devices.
- Select Olimex OpenOCD JTAG ARM-USB-TINY-H from the drop down menu.
- Select the WinUSB driver.
- Click Install Driver.
- Run the `newt load boot_olimex` command again.

Run the `newt load olimex_blinky` command to load the blinky application image onto the board:

```
$ newt load -v olimex_blinky
Loading app image into slot 1
Load command: ~/dev/myproj/repos/apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard/
↳ olimex_stm32-e407_devboard_download.sh ~/dev/myproj/repos/apache-mynewt-core/hw/bsp/
↳ olimex_stm32-e407_devboard ~/dev/myproj/bin/targets/olimex_blinky/app/apps/blinky/
↳ blinky
Successfully loaded image.
```

The LED should be blinking!

Let's double check that it is indeed booting from flash and making the LED blink from the image in flash. Pull the USB cable off the Olimex JTAG adaptor, severing the debug connection to the JTAG port. Next power off the Olimex board by pulling out the USB cable from the board. Wait for a couple of seconds and plug the USB cable back to the board.

The LED light will start blinking again. Success!

If you want to download the image to flash and open a gdb session, use `newt debug blinky`.

Note: The output of the debug session below is for Mac OS and Linux platforms. On Windows, openocd and gdb are started in separate Windows Command Prompt terminals, and the terminals are automatically closed when you quit gdb. In addition, the output of openocd is logged to the `openocd.log` file in your project's base directory instead of the terminal.

Type `c` to continue inside the gdb session.

```
$ newt debug blinky
Debugging with ~/dev/myproj/hw/bsp/olimex_stm32-e407...
Debugging ~/dev/myproj/project/blinky/bin/blinky/blinky.elf
```

(continues on next page)

(continued from previous page)

```
GNU gdb (GNU Tools for ARM Embedded Processors) 7.8.0.20150604-cvs
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 <http://gnu.org/licenses/gpl.html>
...
(info)
...
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x08000250 msp: 0x10010000
Info : accepting 'gdb' connection from 3333
Info : device id = 0x10036413
Info : flash size = 1024kbytes
Reset_Handler () at startup_STM32F40x.s:199
199     ldr      r1, =_etext
(gdb)
```

If you want to erase the flash and load the image again you may use the following commands from within gdb. `flash erase_sector 0 0 x` tells it to erase sectors 0 through x. When you ask it to display (in hex notation) the contents of the sector starting at location ‘lma,’ you should see all f’s. The memory location 0x8000000 is the start or origin of the flash memory contents and is specified in the olimex_stm32-e407_devboard.ld linker script. The flash memory locations is specific to the processor.

```
(gdb) monitor flash erase_sector 0 0 4
erased sectors 0 through 4 on flash bank 0 in 2.296712s
(gdb) monitor mdw 0x08000000 16
0x08000000: ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
(0x08000020: ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
(0x08000000: ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
(0x08000020: ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff ffffffff
(gdb) monitor flash info 0
```

4.3.8 Blinky, your “Hello World!”, on a nRF52 Development Kit

This tutorial shows you how to create, build, and run the Blinky application on a nRF52 Development Kit.

- *Prerequisites*
- *Create a Project*
- *Create the Targets*
- *Build the Target Executables*
- *Sign and Create the Blinky Application Image*
- *Connect to the Board*
- *Load the Bootloader and the Blinky Application Image*

Note that there are several versions of the nRF52 Development Kit in the market. The boards tested with this tutorial are listed under “Prerequisites”.

Prerequisites

- Meet the prerequisites listed in [Project Blinky](#).
- Have a nRF52 Development Kit (one of the following)
 - Nordic nRF52-DK Development Kit - PCA 10040
 - Rigado BMD-300 Evaluation Kit - BMD-300-EVAL-ES
- Install the [Segger JLINK Software](#) and documentation pack.

This tutorial uses the Nordic nRF52-DK board.

Create a Project

Create a new project if you do not have an existing one. You can skip this step and proceed to [Create the Targets](#) if you already have a project created.

Run the following commands to create a new project:

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new myproj
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in myproj...
Project myproj successfully created.
$ cd myproj
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
Downloading repository mynewt-nimble (commit: master) ...
Downloading repository mcuboot (commit: master) ...
Downloading repository mynewt-mcumgr (commit: master) ...
Making the following changes to the project:
apache-mynewt-core successfully upgraded to version 1.7.0
apache-mynewt-nimble successfully upgraded to version 1.2.0
mcuboot successfully upgraded to version 1.3.1
$
```

Create the Targets

Create two targets for the nRF52-DK board - one for the bootloader and one for the Blinky application.

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `nrf52_boot`:

Note: This tutorial uses the Nordic nRF52-DK board. You must specify the correct bsp for the board you are using.

- For the Rigado Eval Kit choose @apache-mynewt-core/hw/bsp/bmd300eval instead (in the highlighted lines)

```
$ newt target create nrf52_boot
$ newt target set nrf52_boot app=@mcuboot/boot/mynewt
$ newt target set nrf52_boot bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
$ newt target set nrf52_boot build_profile=optimized
```

Run the following `newt target` commands to create a target for the Blinky application. We name the target `nrf52_blinky`.

```
$ newt target create nrf52_blinky
$ newt target set nrf52_blinky app=apps/blinky
$ newt target set nrf52_blinky bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
$ newt target set nrf52_blinky build_profile=debug
```

You can run the `newt target show` command to verify the target settings:

```
$ newt target show
targets/nrf52_blinky
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
    build_profile=debug
targets/nrf52_boot
    app=@mcuboot/boot/mynewt
    bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
    build_profile=optimized
```

Build the Target Executables

Run the `newt build nrf52_boot` command to build the bootloader:

```
$ newt build nrf52_boot
Building target targets/nrf52_boot
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/aes.c
Compiling repos/mcuboot/boot/bootutil/src/loader.c
Compiling repos/mcuboot/boot/bootutil/src/image_validate.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/mcuboot/boot/mynewt/src/main.c

...
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/nrf52_boot/app/boot/mynewt/mynewt.elf
Target successfully built: targets/nrf52_boot
```

Run the `newt build nrf52_blinky` command to build the Blinky application:

```
$ newt build nrf52_blinky
Building target targets/nrf52_blinky
Assembling repos/apache-mynewt-core/hw/bsp/nrf52dk/src/arch/cortex_m4/gcc_startup_nrf52_
↪split.s
Compiling repos/apache-mynewt-core/hw/bsp/nrf52dk/src/sbrk.c
Compiling repos/apache-mynewt-core/hw/cmsis-core/src/cmsis_nvic.c
Compiling repos/apache-mynewt-core/hw/drivers/uart/uart_hal/src/uart_hal.c
Assembling repos/apache-mynewt-core/hw/bsp/nrf52dk/src/arch/cortex_m4/gcc_startup_nrf52.s
Compiling apps/blinky/src/main.c
```

(continues on next page)

(continued from previous page)

```
...
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/nrf52_blinky/app/apps/blinky/blinky.elf
Target successfully built: targets/nrf52_blinky
```

Sign and Create the Blinky Application Image

Run the `newt create-image nrf52_blinky 1.0.0` command to create and sign the application image. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image nrf52_blinky 1.0.0
App image successfully generated: ~/dev/myproj/bin/targets/nrf52_blinky/app/apps/blinky/
→blinky.img
```

Connect to the Board

- Connect a micro-USB cable from your computer to the micro-USB port on the nRF52-DK board.
- Turn the power on the board to ON. You should see the green LED light up on the board.

Load the Bootloader and the Blinky Application Image

Run the `newt load nrf52_boot` command to load the bootloader onto the board:

```
$ newt load nrf52_boot
Loading bootloader
$
```

Run the `newt load nrf52_blinky` command to load the Blinky application image onto the board.

```
$ newt load nrf52_blinky
Loading app image into slot 1
```

You should see the LED1 on the board blink!

Note: If the LED does not blink, try resetting your board.

If you want to erase the flash and load the image again, you can run `JLinkExe` to issue an `erase` command.

Note: On Windows: Run the `jlink` command with the same arguments from a Windows Command Prompt terminal.

```
$ JLinkExe -device nRF52 -speed 4000 -if SWD
SEGGER J-Link Commander V5.12c (Compiled Apr 21 2016 16:05:51)
DLL version V5.12c, compiled Apr 21 2016 16:05:45

Connecting to J-Link via USB...O.K.
Firmware: J-Link OB-SAM3U128-V2-NordicSemi compiled Mar 15 2016 18:03:17
Hardware version: V1.00
S/N: 682863966
```

(continues on next page)

(continued from previous page)

```
VTref = 3.300V
```

```
Type "connect" to establish a target connection, '?' for help
J-Link>erase
Cortex-M4 identified.
Erasing device (0;?i?)...
Comparing flash [100%] Done.
Erasing flash [100%] Done.
Verifying flash [100%] Done.
J-Link: Flash download: Total time needed: 0.363s (Prepare: 0.093s, Compare: 0.000s, ↵
    Erase: 0.262s, Program: 0.000s, Verify: 0.000s, Restore: 0.008s)
Erasing done.
J-Link>exit
$
```

4.3.9 Blinky, your “Hello World!”, on a PineTime smartwatch

This tutorial shows you how to create, build, and run the Blinky application on a PineTime smartwatch.

- *Prerequisites*
- *Create a Project*
- *Create the Targets*
- *Build the Target Executables*
- *Sign and Create the Blinky Application Image*
- *Connect to the Board*
- *Load the Bootloader and the Blinky Application Image*

Prerequisites

- Meet the prerequisites listed in [Project Blinky](#).
- Have a PineTime Dev Kit
- Have a ST-LINK programmer or a cheap clone
- Install a patched version of OpenOCD 0.10.0 described in [Install OpenOCD](#).

If you have not removed the flash protection yet, you should do that first. See the [PineTime wiki](#) for instructions.

Create a Project

Create a new project if you do not have an existing one. You can skip this step and proceed to [Create the Targets](#) if you already have a project created.

Run the following commands to create a new project:

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new blinky-pinetime
Downloading project skeleton from apache/mynewt-blinky...
Downloading repository mynewt-blinky
Installing skeleton in blinky-pinetime...
Project blinky-pinetime successfully created.
$ cd blinky-pinetime
$ newt upgrade
newt upgrade
Downloading repository mynewt-core (commit: master)
Downloading repository mynewt-nimble (commit: master)
Downloading repository mcuboot (commit: master)
Making the following changes to the project:
    install apache-mynewt-core (1.7.0)
    install apache-mynewt-nimble (1.2.0)
    install mcuboot (1.3.1)
$
```

Create the Targets

Create two targets for the PineTime - one for the bootloader and one for the Blinky application.

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `boot-pinetime`:

```
$ newt target create boot-pinetime
$ newt target set boot-pinetime app=@mcuboot/boot/mynewt
$ newt target set boot-pinetime bsp=@apache-mynewt-core/hw/bsp/pinetime
$ newt target set boot-pinetime build_profile=optimized
```

Run the following `newt target` commands to create a target for the Blinky application. We name the target `blinky-pinetime`.

```
$ newt target create blinky-pinetime
$ newt target set blinky-pinetime app=apps/blinky
$ newt target set blinky-pinetime bsp=@apache-mynewt-core/hw/bsp/pinetime
$ newt target set blinky-pinetime build_profile=debug
```

You can run the `newt target show` command to verify the target settings:

```
$ newt target show
targets/blinky-pinetime
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/pinetime
    build_profile=debug
targets/boot-pinetime
```

(continues on next page)

(continued from previous page)

```
app=@mcuboot/boot/mynewt
bsp=@apache-mynewt-core/hw/bsp/pinetime
build_profile=optimized
```

Build the Target Executables

Run the `newt build boot-pinetime` command to build the bootloader:

```
$ newt build boot-pinetime
Building target targets/boot-pinetime
Compiling bin/targets/boot-pinetime/generated/src/boot-pinetime-sysflash.c
Compiling bin/targets/boot-pinetime/generated/src/boot-pinetime-sysinit-app.c
Compiling bin/targets/boot-pinetime/generated/src/boot-pinetime-sysdown-app.c
Compiling repos/mcuboot/boot/bootutil/src/boot_record.c
Compiling repos/mcuboot/boot/bootutil/src/caps.c

...
Archiving sys_sysinit.a
Archiving util_mem.a
Archiving util_rwlock.a
Linking /tmp/blinky-pinetime/bin/targets/boot-pinetime/app/boot/mynewt/mynewt.elf
Target successfully built: targets/boot-pinetime
```

Run the `newt build blinky-pinetime` command to build the Blinky application:

```
$ newt build blinky-pinetime
Building target targets/blinky-pinetime
Compiling bin/targets/blinky-pinetime/generated/src/blinky-pinetime-sysinit-app.c
Compiling bin/targets/blinky-pinetime/generated/src/blinky-pinetime-sysdown-app.c
Compiling bin/targets/blinky-pinetime/generated/src/blinky-pinetime-sysflash.c
Compiling repos/apache-mynewt-core/hw/bsp/pinetime/src/sbrk.c
Compiling apps/blinky/src/main.c
Assembling repos/apache-mynewt-core/hw/bsp/pinetime/src/arch/cortex_m4/gcc_startup_nrf52.
→S
Compiling repos/apache-mynewt-core/hw/bsp/pinetime/src/hal_bsp.c

...
Archiving sys_sysinit.a
Archiving util_mem.a
Archiving util_rwlock.a
Linking /tmp/blinky-pinetime/bin/targets/blinky-pinetime/app/apps/blinky/blinky.elf
Target successfully built: targets/blinky-pinetime
```

Sign and Create the Blinky Application Image

Run the `newt create-image blinky-pinetime 1.0.0` command to create and sign the application image. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image blinky-pinetime 1.0.0
App image successfully generated: ~/dev/blinky-pinetime/bin/targets/blinky-pinetime/app/
↳ apps/blinky/blinky.img
```

Connect to the Board

- Connect a ST-LINK programmer via USB to your computer.
- Connect the ST-LINK programmer to the SWD connectors of the PineTime. The easiest way is to mount the PineTime on a cradle and let the programmer cable rest on the contact pads. See [PineTime wiki](#) for the pinout.

Load the Bootloader and the Blinky Application Image

Run the `newt load boot-pinetime` command to load the bootloader onto the board:

```
$ newt load boot-pinetime
Loading bootloader
$
```

Run the `newt load blinky-pinetime` command to load the Blinky application image onto the board.

```
$ newt load blinky-pinetime
Loading app image into slot 1
```

You should see the backlight of the screen board blink!

Note: The screen itself will stay black, this could make the blinking difficult to see.

4.3.10 Blinky, your “Hello World!”, on RedBear Nano 2

This tutorial shows you how to create, build and run the Blinky application on a RedBear Nano 2 board.

- *Prerequisites*
- *Create a Project*
- *Create the Targets*
- *Build the Target Executables*
- *Sign and Create the Blinky Application Image*
- *Connect to the Board*
- *Load the Bootloader*
 - *Clear the Write Protection on the Flash Memory*
- *Load the Blinky Application Image*

Prerequisites

- Meet the prerequisites listed in [Project Blinky](#).
- Have a RedBear Nano 2 board.
- Install a patched version of OpenOCD 0.10.0 described in [Install OpenOCD](#).

Create a Project

Create a new project if you do not have an existing one. You can skip this step and proceed to Create the Targets if you already have a project created.

Run the following commands to create a new project:

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new myproj
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in myproj...
Project myproj successfully created.
$ cd myproj
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
Downloading repository mynewt-nimble (commit: master) ...
Downloading repository mcuboot (commit: master) ...
Downloading repository mynewt-mcumgr (commit: master) ...
Making the following changes to the project:
apache-mynewt-core successfully upgraded to version 1.7.0
apache-mynewt-nimble successfully upgraded to version 1.2.0
mcuboot successfully upgraded to version 1.3.1
$
```

Create the Targets

Create two targets for the RedBear Nano 2 board - one for the bootloader and one for the Blinky application.

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `rbnano2_boot`:

```
$ newt target create rbnano2_boot
$ newt target set rbnano2_boot app=@mcuboot/boot/mynewt
$ newt target set rbnano2_boot bsp=@apache-mynewt-core/hw/bsp/rb-nano2
$ newt target set rbnano2_boot build_profile=optimized
```

Run the following `newt target` commands to create a target for the Blinky application. We name the target `nrf52_blinky`.

```
$ newt target create rbnano2_blinky
$ newt target set rbnano2_blinky app=apps/blinky
$ newt target set rbnano2_blinky bsp=@apache-mynewt-core/hw/bsp/rb-nano2
$ newt target set rbnano2_blinky build_profile=debug
```

You can run the `newt target show` command to verify the target settings:

```
$ newt target show
targets/rbnano2_blinky
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/rb-nano2
    build_profile=debug
targets/rbnano2_boot
    app=@mcuboot/boot/mynewt
    bsp=@apache-mynewt-core/hw/bsp/rb-nano2
    build_profile=optimized
```

Build the Target Executables

Run the `newt build rbnano2_boot` command to build the bootloader:

```
$ newt build rbnano2_boot
Building target targets/rbnano2_boot
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/loader.c
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/aes.c
Compiling repos/mcuboot/boot/bootutil/src/image_validate.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/mcuboot/boot/mynewt/src/main.c

...
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/rbnano2_boot/app/boot/mynewt/mynewt.elf
Target successfully built: targets/rbnano2_boot
```

Run the `newt build rbnano2_blinky` command to build the Blinky application:

```
$ newt build rbnano2_blinky
Building target targets/rbnano2_blinky
Assembling repos/apache-mynewt-core/hw/bsp/rb-nano2/src/arch/cortex_m4/gcc_startup_nrf52_
→split.s
Compiling repos/apache-mynewt-core/hw/drivers/uart/src/uart.c
Compiling repos/apache-mynewt-core/hw/cmsis-core/src/cmsis_nvic.c
Compiling repos/apache-mynewt-core/hw/bsp/rb-nano2/src/sbrk.c
Compiling apps/blinky/src/main.c

...
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/rbnano2_blinky/app/apps/blinky/blinky.elf
Target successfully built: targets/rbnano2_blinky
```

Sign and Create the Blinky Application Image

Run the `newt create-image rbnano2_blinky 1.0.0` command to create and sign the application image. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image rbnano2_blinky 1.0.0
App image successfully generated: ~/dev/myproj/bin/targets/rbnano2_blinky/app/apps/blinky/
→blinky.img
```

Connect to the Board

Connect the RedBear Nano 2 USB to a USB port on your computer. You should see an orange LED light up on the board.

Load the Bootloader

Run the `newt load rbnano2_boot` command to load the bootloader onto the board:

```
$ newt load rbnano2_boot
Loading bootloader
$
```

Note: On Windows platforms, if you get an `unable to find CMSIS-DAP device` error, you will need to download and install the mbed Windows serial port driver from <https://developer.mbed.org/handbook/Windows-serial-configuration>. Follow the instructions from the site to install the driver. Here are some additional notes about the installation:

1. The instructions indicate that the mbed Windows serial port driver is not required for Windows 10. If you are using Windows 10 and get the `unable to find CMSIS-DAP device` error, we recommend that you install the driver.
2. If the driver installation fails, we recommend that you unplug the board, plug it back in, and retry the installation.

Run the `newt load rbnano2_boot` command again.

Clear the Write Protection on the Flash Memory

The flash memory on the RedBear Nano 2 comes write protected from the factory. If you get an error loading the bootloader and you are using a brand new chip, you need to clear the write protection from the debugger and then load the bootloader again. Run the `newt debug rbnano2_blinky` command and issue the following commands at the highlighted (gdb) prompts.

Note: The output of the debug session below is for Mac OS and Linux platforms. On Windows, openocd and gdb are started in separate Windows Command Prompt terminals, and the terminals are automatically closed when you quit gdb. In addition, the output of openocd is logged to the `openocd.log` file in your project's base directory instead of the terminal.

```
$ newt debug rbnano2_blinky
[~/dev/myproj/repos/apache-mynewt-core/hw/bsp/rb-nano2/rb-nano2_debug.sh ~/dev/myproj/
→repos/apache-mynewt-core/hw/bsp/rb-nano2 ~/dev/myproj/bin/targets/rbnano2_blinky/app/
→apps/blinky/blinky]
Open On-Chip Debugger 0.10.0-dev-snapshot (2017-03-28-11:24)
Licensed under GNU GPL v2
```

(continues on next page)

(continued from previous page)

```
...
(gdb) set {unsigned long}0x4001e504=2
(gdb) x/1wx 0x4001e504
0x4001e504:0x00000002
(gdb) set {unsigned long}0x4001e50c=1
Info : SWD DPIDR 0x2ba01477
Error: Failed to read memory at 0x00009ef4
(gdb) x/32wx 0x00
0x0:0xffffffff0xffffffff0xffffffff0xffffffff
0x10:0xffffffff0xffffffff0xffffffff0xffffffff
0x20:0xffffffff0xffffffff0xffffffff0xffffffff
0x30:0xffffffff0xffffffff0xffffffff0xffffffff
0x40:0xffffffff0xffffffff0xffffffff0xffffffff
0x50:0xffffffff0xffffffff0xffffffff0xffffffff
0x60:0xffffffff0xffffffff0xffffffff0xffffffff
0x70:0xffffffff0xffffffff0xffffffff0xffffffff
(gdb)
```

Load the Blinky Application Image

Run the `newt load rbnano2_blinky` command to load the Blinky application image onto the board:

```
$ newt load rbnano2_blinky
Loading app image into slot 1
```

You should see a blue LED on the board blink!

Note: If the LED does not blink, try resetting your board.

4.3.11 Blinky, your “Hello World!”, on STM32F4-Discovery

This tutorial shows you how to create, build, and run the Blinky application on the STM32F4-Discovery board.

Prerequisites

- Meet the prerequisites listed in [Project Blinky](#).
- Have a STM32F4-Discovery board.
- Have a USB type A to Mini-B cable.
- Install a patched version of OpenOCD 0.10.0 described in [Install OpenOCD](#).

Create a Project

Create a new project if you do not have an existing one. You can skip this step and proceed to Create the Targets if you already have a project created.

Run the following commands to create a new project:

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new myproj
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in myproj...
Project myproj successfully created.
$ cd myproj
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
apache-mynewt-core successfully upgraded to version 1.7.0
$
```

Create the Targets

Create two targets for the STM32F4-Discovery board - one for the bootloader and one for the Blinky application.

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `stm32f4disc_boot`:

```
$ newt target create stm32f4disc_boot
$ newt target set stm32f4disc_boot app=@mcuboot/boot/mynewt
$ newt target set stm32f4disc_boot bsp=@apache-mynewt-core/hw/bsp/stm32f4discovery
$ newt target set stm32f4disc_boot build_profile=optimized
```

Run the following `newt target` commands to create a target for the Blinky application. We name the target `stm32f4disc_blinky`:

```
$ newt target create stm32f4disc_blinky
$ newt target set stm32f4disc_blinky app=apps/blinky
$ newt target set stm32f4disc_blinky bsp=@apache-mynewt-core/hw/bsp/stm32f4discovery
$ newt target set stm32f4disc_blinky build_profile=debug
```

You can run the `newt target show` command to verify the target settings:

```
$ newt target show
targets/my_blinky_sim
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/native
    build_profile=debug
targets/stm32f4disc_blinky
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/stm32f4discovery
    build_profile=debug
targets/stm32f4disc_boot
    app=@mcuboot/boot/mynewt
    bsp=@apache-mynewt-core/hw/bsp/stm32f4discovery
    build_profile=optimized
```

Build the Target Executables

Run the `newt build stm32f4disc_boot` command to build the bootloader:

```
$ newt build stm32f4disc_boot
Building target targets/stm32f4disc_boot
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/mcuboot/boot/mynewt/src/main.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/mcuboot/boot/bootutil/src/image_validate.c
Compiling repos/mcuboot/boot/bootutil/src/loader.c

...
Archiving sys_flash_map.a
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/stm32f4disc_boot/app/boot/mynwet/mynewt.elf
Target successfully built: targets/stm32f4disc_boot
```

Run the `newt build stm32f4disc_blinky` command to build the Blinky application:

```
$newt build stm32f4disc_blinky
Building target targets/stm32f4disc_blinky
Compiling apps/blinky/src/main.c
Compiling repos/apache-mynewt-core/hw/bsp/stm32f4discovery/src/sbrk.c
Compiling repos/apache-mynewt-core/hw/bsp/stm32f4discovery/src/system_stm32f4xx.c
Compiling repos/apache-mynewt-core/hw/bsp/stm32f4discovery/src/hal_bsp.c
Assembling repos/apache-mynewt-core/hw/bsp/stm32f4discovery/src/arch/cortex_m4/startup_
→STM32F40x.s
Compiling repos/apache-mynewt-core/hw/cmsis-core/src/cmsis_nvic.c
Compiling repos/apache-mynewt-core/hw/drivers/uart/src/uart.c
Compiling repos/apache-mynewt-core/hw/drivers/uart/uart_hal/src/uart_hal.c
Compiling repos/apache-mynewt-core/hw/hal/src/hal_common.c
Compiling repos/apache-mynewt-core/hw/hal/src/hal_flash.c

...
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/stm32f4disc_blinky/app/apps/blinky/blinky.elf
Target successfully built: targets/stm32f4disc_blinky
```

Sign and Create the Blinky Application Image

Run the `newt create-image stm32f4disc_blinky 1.0.0` command to create and sign the application image. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image stm32f4disc_blinky 1.0.0
App image successfully generated: ~/dev/myproj/bin/targets/stm32f4disc_blinky/app/apps/
blink/blinky.img
```

Connect to the Board

Connect a USB type A to Mini-B cable from your computer to the port the board indicated on the diagram:



You should see the small PWR red LED light up.

Load the Bootloader and the Blinky Application Image

Run the `newt load stm32f4disc_boot` command to load the bootloader onto the board:

```
$ newt load stm32f4disc_boot
Loading bootloader
```

Note: If you are using Windows and get an `open failed` or `no device found` error, you will need to install the usb driver. Download [Zadig](#) and run it:

- Select Options > List All Devices.
- Select STM32 STLink from the drop down menu.
- Select the WinUSB driver.
- Click Install Driver.
- Run the `newt load stm32f4disc_boot` command again.

Note: If you are running Linux and get an `open failed` message, there are two common issues with this board. If you have a board produced before mid-2016, it is likely that you have an older version of the ST-LINK programmer. To correct this, open the `repos/apache-mynewt-core/hw/bsp/stm32f4discovery/f4discovery.cfg` file in a text editor, and change the line:

```
source [find interface/stlink-v2-1.cfg]
```

to:

```
source [find interface/stlink-v2.cfg]
```

If you receive an error like `libusb_open()` failed with `LIBUSB_ERROR_ACCESS`, it means that your udev rules are not correctly set up for this device. You can find some example udev rules for ST-LINK programmers [here](#).

Run the `newt load stm32f4disc_blinky` command to load the Blinky application image onto the board.

```
$ newt load stm32f4disc_blinky
Loading app image into slot 1
```

You should see the small green LD4 LED on the board blink!

Note: If the LED does not blink, try resetting your board.

If you want to erase the flash and load the image again, start a debug session, and enter `mon stm32f2x mass_erase 0` at the gdb prompt:

Note: The output of the debug session below is for Mac OS and Linux platforms. On Windows, openocd and gdb are started in separate Windows Command Prompt terminals, and the terminals are automatically closed when you quit gdb. In addition, the output of openocd is logged to the openocd.log file in your project's base directory instead of the terminal.

```
$ newt debug stm32f4disc_blinky
[~/dev/myproj/repos/apache-mynewt-core/hw/bsp/stm32f4discovery/stm32f4discovery_debug.sh
 ↵~/dev/myproj/repos/apache-mynewt-core/hw/bsp/stm32f4discovery ~/dev/myproj/bin/targets/
 ↵stm32f4disc_blinky/app/apps/blinky/blinky]
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
      http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might
      differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v25 API v2 SWIM v14 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 2.881129
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
target halted due to debug-request, current mode: Thread
...
Reading symbols from ~/dev/myproj/bin/targets/stm32f4disc_blinky/app/apps/blinky/blinky.
elf...done.
target halted due to debug-request, current mode: Thread
xPSR: 0x41000000 pc: 0x08021e90 psp: 0x20002290
Info : accepting 'gdb' connection on tcp/3333
Info : device id = 0x10076413
Info : flash size = 1024kbytes
0x08021e90 in __WFI () at repos/apache-mynewt-core/hw/cmsis-core/src/ext/core_cmInstr.
↪h:342
342     __ASM volatile ("wfi");
(gdb) mon stm32f2x mass_erase 0
stm32x mass erase complete
stm32x mass erase complete
(gdb)
```

4.3.12 Blinky, your “Hello World!”, on STM32F303 Discovery

Objective

Learn how to use packages from a default application repository of Mynewt to build your first *Hello World* application (Blinky) on a target board. Once built using the *newt* tool, this application will blink the LED lights on the target board.

Create a project with a simple app that blinks an LED on the STM F303 discovery board. In the process import some external libraries into your project. Download the application to the target and watch it blink!

What you need

- Discovery kit with STM32F303VC MCU
- Laptop running Mac OSX.
- It is assumed you have already installed newt tool.
- It is assumed you already installed native tools as described [here](#)

Also, we assume that you're familiar with UNIX shells. Let's gets started!

Create a project

Create a new project to hold your work. For a deeper understanding, you can read about project creation in [Get Started – Creating Your First Project](#) or just follow the commands below.

If you've already created a project from another tutorial, you can re-use that project.

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new myproj
Downloading project skeleton from apache/incubator-mynewt-blinky...
Installing skeleton in myproj...
Project myproj successfully created.

$ cd myproj
```

Note: Don't forget to change into the `myproj` directory.

Install dependencies

Now you can install this into the project using:

```
$ newt install -v
Downloading repository description for apache-mynewt-core... success!
...
apache-mynewt-core successfully installed version 0.7.9-none
...
Downloading repository description for mynewt_stm32f3... success!
Downloading repository mynewt_stm32f3
...
Resolving deltas: 100% (65/65), done.
Checking connectivity... done.
mynewt_stm32f3 successfully installed version 0.0.0-none
```

Create targets

Create two targets to build using the stmf3 board support package and the app blinky example from mynewt. The output of these commands are not shown here for brevity.

The first target is the application image itself. The second target is the bootloader which allows you to upgrade your mynewt applications.

```
$ newt target create stmf3_blinky
$ newt target set stmf3_blinky build_profile=optimized
$ newt target set stmf3_blinky bsp=@apache-mynewt-core/hw/bsp/stm32f3discovery
$ newt target set stmf3_blinky app=apps/blinky

$ newt target create stmf3_boot
$ newt target set stmf3_boot app=@mcuboot/boot/mynewt
$ newt target set stmf3_boot bsp=@apache-mynewt-core/hw/bsp/stm32f3discovery
$ newt target set stmf3_boot build_profile=optimized

$ newt target show

targets/stmf3_blinky
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/stm32f3discovery
    build_profile=optimized
targets/stmf3_boot
    app=@mcuboot/boot/mynewt
    bsp=@apache-mynewt-core/hw/bsp/stm32f3discovery
    build_profile=optimized
```

Build the target executables

To build the images, use the `newt build` command below.

```
$ newt build stmf3_blinky
...
Archiving stm32f3discovery.a
Linking blinky.elf
App successfully built: ~/dev/myproj/bin/stmf3_blinky/apps/blinky/blinky.elf

$ newt build stmf3_boot
Compiling log_shell.c
Archiving log.a
Linking boot.elf
App successfully built: ~/dev/myproj/bin/stmf3_boot/apps/boot/boot.elf
```

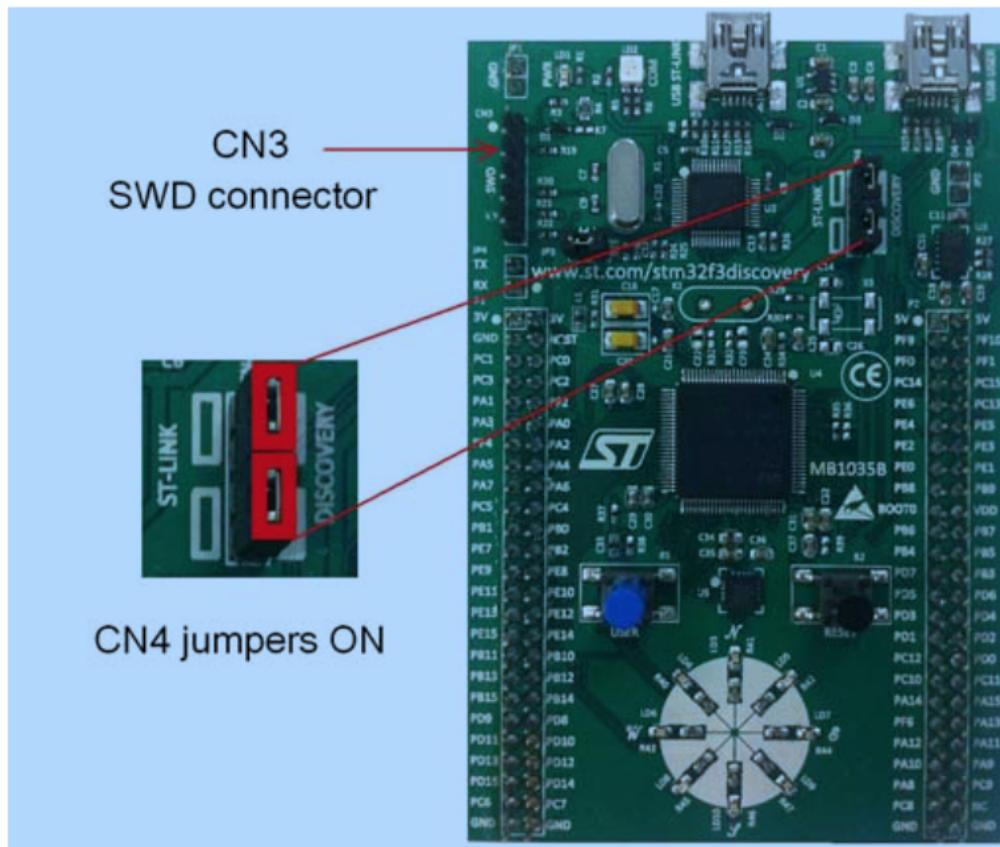
Sign and create the blinky application image

You must sign and version your application image to download it using newt. Use the `newt create-image` command to perform this action. Here we assign this image an arbitrary version 1.2.3.

```
$ newt create-image stmf3_blinky 1.2.3
App image successfully generated: ~/dev/myproj/bin/stmf3_blinky/apps/blinky/blinky.img
Build manifest:~/dev/myproj/bin/stmf3_blinky/apps/blinky/manifest.json
```

Configure the hardware

The STM32F3DISCOVERY board includes an ST-LINK/V2 embedded debug tool interface that will be used to program/debug the board. To program the MCU on the board, simply plug in the two jumpers on CN4, as shown in the picture in red. If you want to learn more about the board you will find the User Manual at http://www.st.com/st-web-ui/static/active/jp/resource/technical/document/user_manual/DM00063382.pdf



Download the Images

Use the `newt load` command to download the images to the target board.

```
$ newt -v load stmf3_boot
$ newt -v load stmf3_blinky
```

Watch the LED blink

Congratulations! You have built, downloaded, and run your first application using mynewt for the stm32f3 discovery board. One of the LEDs on the LED wheel should be blinking at 1 Hz.

Want more?

Want to make your board do something a little more exciting with the LEDs? Then try making the modifications to the Blinky app to make it a *pin wheel app* and you can light all the LEDs in a pin-wheel fashion.

We have more fun tutorials for you to get your hands dirty. Be bold and try other Blinky-like *tutorials* or try enabling additional functionality such as *remote comms* on the current board.

If you see anything missing or want to send us feedback, please do so by signing up for appropriate mailing lists on our [Community Page](#)

Keep on hacking and blinking!

4.3.13 Pin Wheel Modifications to “Blinky” on STM32F3 Discovery

Objective

Learn how to modify an existing app – the *blinky* app – to light all the LEDs on the STM32F3 Discovery board.

What you need

- Discovery kit with STM32F303VC MCU
- Laptop running Mac OSX.
- It is assumed you have already installed and run the *blinky* app successfully.

Since you've already successfully created your *blinky* app project, you'll need to modify only one file, `main.c`, in order to get this app working.

The `main.c` file resides in the `apps/blinky/src` directory in your project folder so you can edit it with your favorite editor. You'll make the following changes:

Replace the line:

```
int g_led_pin;
```

With the line:

```
int g_led_pins[8] = {LED_BLINK_PIN_1, LED_BLINK_PIN_2, LED_BLINK_PIN_3, LED_BLINK_PIN_4,
                     LED_BLINK_PIN_5, LED_BLINK_PIN_6, LED_BLINK_PIN_7, LED_BLINK_PIN_8};
```

So that you now have an array of all 8 LED Pins on the board.

Delete the line:

```
g_led_pin = LED_BLINK_PIN;
```

And in its place, add the following lines to initialize all the LED_PINS correctly:

```
int x;
for(x = 0; x < 8; x++){
    hal_gpio_init_out(g_led_pins[x], 1);
}
int p = 0;
```

We'll use that 'p' later. Next you'll want to change the line:

```
os_time_delay(1000);
```

to a shorter time in order to make it a little more interesting. A full 1 second delay doesn't look great, so try 100 for starters and then you can adjust it to your liking.

Finally, change the line:

```
hal_gpio_toggle(g_led_pin);
```

to look like this:

```
hal_gpio_toggle(g_led_pins[p++]);
p = (p > 7) ? 0 : p;
```

Build the target and executables and download the images

Run the same commands you used on the blinky app to build and load this one:

```
$ newt create-image stmf3_blinky 1.2.3
App image successfully generated: ~/dev/myproj/bin/stmf3_blinky/apps/blinky/blinky.img
Build manifest:~/dev/myproj/bin/stmf3_blinky/apps/blinky/manifest.json
$ newt -v load stmf3_boot
$ newt -v load stmf3_blinky
```

Watch the LEDs go round and round

The colored LEDs should now all light up in succession, and once they're all lit, they should then go off in the same order. This should repeat continuously.

If you see anything missing or want to send us feedback, please do so by signing up for appropriate mailing lists on our [Community Page](#).

Keep on hacking and blinking!

4.3.14 Enabling The Console and Shell for Blinky

This tutorial shows you how to add the Console and Shell to the Blinky application and interact with it over a serial line connection.

- *Prerequisites*
- *Use an Existing Project*
- *Modify the Dependencies and Configuration*
- *Use the OS Default Event Queue to Process Blinky Timer and Shell Events*
- *Modify main.c*
- *Build, Run, and Upload the Blinky Application Target*
- *Set Up a Serial Connection*
- *Communicate with the Application*

Prerequisites

- Work through one of the [Blinky Tutorials](#) to create and build a Blinky application for one of the boards.
- Have a *serial setup*.

Use an Existing Project

Since all we're doing is adding the shell and console capability to blinky, we assume that you have worked through at least some of the other tutorials, and have an existing project. For this example, we'll be modifying the [blinky on nRF52](#) project to enable the shell and console connectivity. You can use blinky on a different board.

Modify the Dependencies and Configuration

Modify the package dependencies in your application target's `pkg.yml` file as follows:

- Add the shell package: `@apache-mynewt-core/sys/shell`.
- Replace the `@apache-mynewt-core/sys/console/stub` package with the `@apache-mynewt-core/sys/console/full` package.

Note: If you are using version 1.1 or lower of blinky, the `@apache-mynewt-core/sys/console/full` package may be already listed as a dependency.

The updated `pkg.yml` file should have the following two lines:

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

This lets the newt system know that it needs to pull in the code for the console and the shell.

Modify the system configuration settings to enable Shell and Console ticks and prompt. Add the following to your application target's `syscfg.yml` file:

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

Use the OS Default Event Queue to Process Blinky Timer and Shell Events

Mynewt creates a main task that executes the application `main()` function. It also creates an OS default event queue that packages can use to queue their events. Shell uses the OS default event queue for Shell events, and `main()` can process the events in the context of the main task.

Blinky's `main.c` is very simple. It only has a `main()` function that executes an infinite loop to toggle the LED and sleep for one second. We will modify `blinky`:

- To use `os_callout` to generate a timer event every one second instead of sleeping. The timer events are added to the OS default event queue.
- To process events from the OS default event queue inside the infinite loop in `main()`.

This allows the main task to process both Shell events and the timer events to toggle the LED from the OS default event queue.

Modify `main.c`

Initialize a `os_callout` timer and move the toggle code from the while loop in `main()` to the event callback function. Add the following code above the `main()` function:

```
/* The timer callout */  
static struct os_callout blinky_callout;  
  
/*  
 * Event callback function for timer events. It toggles the led pin.  
 */  
static void  
timer_ev_cb(struct os_event *ev)  
{  
    assert(ev != NULL);  
  
    ++g_task1_loops;  
    hal_gpio_toggle(g_led_pin);  
  
    os_callout_reset(&blinky_callout, OS_TICKS_PER_SEC);  
}  
  
static void  
init_timer(void)  
{  
    /*  
     * Initialize the callout for a timer event.  
     */  
    os_callout_init(&blinky_callout, os_eventq_dflt_get(),  
                   timer_ev_cb, NULL);
```

(continues on next page)

(continued from previous page)

```
    os_callout_reset(&blinky_callout, OS TICKS_PER_SEC);
}
```

In `main()`, add the call to the `init_timer()` function before the while loop and modify the while loop to process events from the OS default event queue:

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

#ifdef ARCH_sim
    mcu_sim_parse_args(argc, argv);
#endif

    sysinit();

    g_led_pin = LED_BLINK_PIN;
    hal_gpio_init_out(g_led_pin, 1);
    init_timer();
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }
    assert(0);
    return rc;
}
```

Build, Run, and Upload the Blinky Application Target

We're not going to build the bootloader here since we are assuming that you have already built and loaded it during previous tutorials.

We will use the `newt run` command to build and deploy our improved blinky image. The run command performs the following tasks for us:

1. Builds a binary Mynewt executable
2. Wraps the executable in an image header and footer, turning it into a Mynewt image.
3. Uploads the image to the target hardware.
4. Starts a gdb process to remotely debug the Mynewt device.

Run the `newt run nrf52_blinky 0` command. The `0` is the version number that should be written to the image header. Any version will do, so we choose 0.

```
$ newt run nrf52_blinky 0
...
Archiving util_mem.a
Linking /home/me/dev/myproj/bin/targets/nrf52_blinky/app/apps/blinky/blinky.elf
App image successfully generated: /home/me/dev/myproj/bin/targets/nrf52_blinky/app/apps/

```

(continues on next page)

(continued from previous page)

```
→blinky/blinky.elf
Loading app image into slot 1
[/home/me/dev/myproj/repos/apache-mynewt-core/hw/bsp/nrf52dk/nrf52dk_debug.sh /home/me/
→dev/myproj/repos/apache-mynewt-core/hw/bsp/nrf52dk /home/me/dev/myproj/bin/targets/
→nrf52_blinky/app/apps/blinky]
Debugging /home/me/dev/myproj/bin/targets/nrf52_blinky/app/apps/blinky/blinky.elf
```

Set Up a Serial Connection

You'll need a Serial connection to see the output of your program. You can reference the [Using the Serial Port with Mynewt OS](#) Tutorial for more information on setting up your serial communication.

Communicate with the Application

Once you have a connection set up, you can connect to your device as follows:

- On Mac OS and Linux platforms, you can run `minicom -D /dev/tty.usbserial-<port> -b 115200` to connect to the console of your app. Note that on Linux, the format of the port name is `/dev/ttyUSB<N>`, where N is a number.
- On Windows, you can use a terminal application such as PuTTY to connect to the device.

If you located your port from a MinGW terminal, the port name format is `/dev/ttyS<N>`, where N is a number. You must map the port name to a Windows COM port: `/dev/ttyS<N>` maps to `COM<N+1>`. For example, `/dev/ttyS2` maps to `COM3`.

You can also use the Windows Device Manager to locate the COM port.

To test and make sure that the Shell is running, first just hit :

```
004543 shell>
```

You can try some commands:

```
003005 shell> help
003137 Available modules:
003137 os
003138 prompt
003138 To select a module, enter 'select <module name>'.
003140 shell> prompt
003827 help
003827 ticks          shell ticks command
004811 shell> prompt ticks off
005770 Console Ticks off
shell> prompt ticks on
006404 Console Ticks on
006404 shell>
```

4.4 Working with repositories

4.4.1 Adding Repositories to your Project

What is a Repository

A repository is a version-ed Mynewt project, which is a collection of Mynewt packages organized in a specific way for redistribution.

What differentiates a repository from a Mynewt project is the presence of a `repository.yml` file describing the repository. This will be described below. For a basic understanding of repositories you may read the [Newt Tool Guide](#) and [How to create repos](#).

Note: For the remainder of this document we'll use the term repo as shorthand for a Mynewt repository.

Repos are useful because they are an organized way for the community to share Mynewt packages and projects. In fact, the Mynewt-core is distributed as a repo.

Why does Mynewt need additional repos?

Repos add functionality not included in the Mynewt core. New repos might be created for several reasons.

- **Expertise.** Individuals or organizations may have expertise that they want to share in the form of repos. For example a chip vendor may create a repo to hold the Mynewt support for their chips.
- **Non-Core component.** Some components, although very useful to Mynewt users are not core to all Mynewt users. These are likely candidates to be held in different repos.
- **Software licensing.** Some software have licenses that make them incompatible with the ASF (Apache Software Foundation) license policies. These may be valuable components to some Mynewt users, but cannot be contained in the `apache-mynewt-core`.

What Repos are in my Project

The list of repos used by your project are contained within the `project.yml` file. An example can be seen by creating a new project:

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new myproj
$ cd myproj
```

View the `project.yml` section and you will see a line describing the repos:

```
project.repositories:
  - apache-Mynewt-core
```

By default, this newly created project uses a single repo called `apache-Mynewt-core`.

If you wish to add additional repos, you would add additional lines to the `project.repositories` variable like this.

```
project.repositories:
  - apache-Mynewt-core
  - another_repo_named_x
```

Repo Descriptors

In addition to the repo name, the `project.yml` file must also contain a repo descriptor for each repository you include that gives `newt` information on obtaining the repo.

In the same `myproj` above you will see the following repo descriptor.

```
repository.apache-Mynewt-core:  
  type: github  
  vers: 1-latest  
  user: apache  
  repo: mynewt-core
```

A repo descriptor starts with `repository.<name>`. In this example, the descriptor specifies the information for the `apache-Mynewt-core`.

The fields within the descriptor have the following definitions:

- **type** – The type of code storage the repo uses. The current version of `newt` only supports `github`. Future versions may support generic `git` or other code storage mechanisms.
- **vers** – The version of the repo to use for your project. A source code repository contains many versions of the source. This field is used to specify the one to use for this project. See the section on versions below for a detailed description of the format of this field.
- **user** – The username for the repo. On `github`, this is the name after `github.com` in the repo path. Consider the repository `https://github.com/apache/mynewt-core`. It has username `apache`.
- **repo** – The name of the repo. On `github`, this is the name after the username described above. Consider the repository `https://github.com/apache/mynewt-core`. It has path `mynewt-core`. This is a path to the source control and should not be confused with the name of the repo that you used in the `repository.<name>` declaration above. That name is contained elsewhere within the repo. See Below.

Adding Existing Repos to my Project

To add a new repo to your project, you have to complete two steps.

- Edit the `project.yml` file and add a new repo descriptor. The previous section includes information on the field required in your repo descriptor.
- Edit the `project.yml` file and add a new line to the `project.repositories` variable with the name of the repo you are adding.

An example of a `project.yml` file with two repositories is shown below:

```
project.name: "my_project"  
  
project.repositories:  
  - apache-mynewt-core  
  - mynewt_arduino_zero  
  
# Use github's distribution mechanism for core ASF libraries.  
# This provides mirroring automatically for us.  
#  
repository.apache-mynewt-core:  
  type: github  
  vers: 1-latest
```

(continues on next page)

(continued from previous page)

```

user: apache
repo: mynewt-core

# a special repo to hold hardware specific stuff for arduino zero
repository.mynewt_arduino_zero:
  type: github
  vers: 1-latest
  user: runtimeco
  repo: mynewt_arduino_zero

```

What Version of the Repo to use

Mynewt repos are version-ed artifacts. They are stored in source control systems like github. The repo descriptor in your `project.yml` file must specify the version of the repo you will accept into your project.

For now, we are at the beginnings of Mynewt. For testing and evaluation please use `1-latest` in the `vers` field in your repo descriptor.

```
vers:1-latest
```

See [Create a Repo](#) for a description of the versioning system and all the possible ways to specify a version to use.

Identifying a Repo

A repo contains Mynewt packages organized in a specific way and stored in one of the supported code storage methods described above. In other words, it is a Mynewt project with an additional file `repository.yml` which describes the repo for use by `newt` (and humans browsing them). It contains a mapping of version numbers to the actual github branches containing the source code.

Note that the `repository.yml` file lives only in the master branch of the git repository. `Newt` will always fetch this file from the master branch and then use that to determine the actual branch required depending on the version specified in your `project.yml` file. Special care should be taken to ensure that this file exists only in the master branch.

Here is the `repository.yml` file from the `apache-mynewt-core`:

```

repo.name: apache-mynewt-core
repo.versions:
  "0.0.0": "master"
  "0.0.1": "master"
  "0.7.9": "mynewt_0_8_0_b2_tag"
  "0.8.0": "mynewt_0_8_0_tag"
  "0.9.0": "mynewt_0_9_0_tag"
  "0.9.9": "mynewt_1_0_0_b1_tag"
  "0.9.99": "mynewt_1_0_0_b2_tag"
  "0.9.999": "mynewt_1_0_0_rc1_tag"
  "1.0.0": "mynewt_1_0_0_tag"

  "0-latest": "1.0.0"      # 1.0.0
  "0-dev": "0.0.0"         # master

  "0.8-latest": "0.8.0"

```

(continues on next page)

(continued from previous page)

```
"0.9-latest": "0.9.0"  
"1.0-latest": "1.0.0" # 1.0.0
```

It contains the following:

- **repo.name** The external name that is used to include the library in your `project.yml` file. This is the name you include in the `project.repositories` variable when adding this repository to your project.
- **repo.versions** A description of what versions to give the user depending on the settings in their `project.yml` file.

Repo Version

The repo version number resolves to an actual git branch depending on the mapping specified in `repository.yml` for that repo. The version field argument in your `project.yml` file supports multiple formats for flexibility:

```
<major_num>.<minor_num>.<revision_num>
```

or

```
<major_num>.<minor_num>-<stability_string>
```

or

```
<major_num>-<stability_string>
```

The stability string can be one of 3 pre-defined stability values.

1. stable – A stable release version of the repository
2. dev – A development version from the repository
3. latest – The latest from the repository

In your `project.yml` file you can specify different combinations of the version number and stability value. For example:

- 1-latest – The latest version with major number 1
- 1.2-stable – The latest stable version with major and minor number 1.2
- 1.2-dev – The development version from 1.2
- 1.1.1 – a specific version 1.1.1

You cannot specify a stability string with a fully numbered version, e.g.

```
1.2.8-stable
```

Repo Versions Available

A `repository.yml` file contains information to match a version request into a git branch to fetch for your project.

It's up to the repository maintainer to map these to branches of the repository. For example, let's say in a fictitious repository the following are defined.

```
repo.versions:
    "0.8.0": "xxx_branch_0_8_0"
    "1.0.0": "xxx_branch_1_0_0"
    "1.0.2": "xxx_branch_1_0_2"
    "1.1.1": "xxx_branch_1_1_0"
    "1.1.2": "xxx_branch_1_1_2"
    "1.2.0": "xxx_branch_1_2_0"
    "1.2.1": "xxx_branch_1_2_1"
    "1.2-dev": "1.2.1"
    "1-dev": "1.2-dev"
    "1.2-stable": "1.2.0"
    "0-latest": "0.8.0"
    "1-latest": "1-dev"
    ....
```

When the `project.yml` file asks for `1.2-stable` it is resolved to version `1.2.0` (perhaps `1.2.1` is not stable yet), which in turn resolves to a specific branch `xxx_branch_1_2_0`. This is the branch that `newt` fetches into your project.

Note: Make sure a repo version exists in the `repository.yml` file of a repo you wish to add. Otherwise Newt will not be able to resolve the version and will fail to fetch the repo into your project.

How to find out what Repos are available for Mynewt components

Currently, there is no `newt` command to locate/search Mynewt package repositories. However, since the `newt` tool supports only github, searching github by keyword is a satisfactory option until a search tool is created.

When searching Github, recall that a Mynewt repository must have a `repository.yml` file in its root directory. If you don't see that file, it's not a Mynewt repository and can't be included in your project via the `newt` tool.

Once you find a repository, the Github URL and `repository.yml` file should give you all the information to add it to your `project.yml` file.

4.4.2 Create a Repo out of a Project

In order to create a repository out of a project, all you need to do is create a `repository.yml` file, and check it into the master branch of your project.

NOTE: Currently only github source control service is supported by our package management system, but support for plain git will be added soon.

The `repository.yml` defines all versions of the repository and the corresponding source control tags that these versions correspond to. As an example, if the `repository.yml` file has the following content, it means there is one version of the apache-mynewt-core operating system available, which is `0.0.0` (implying we haven't released yet!). Such a version number corresponds to the "develop" branch in this repository. `0-latest` would also resolve to this same `0.0.0` version. The next section explains the versioning system a bit more.

```
$ more repository.yml
repo.name: apache-mynewt-core
repo.versions:
    "0.0.0": "develop"
    "0-latest": "0.0.0"
```

Where should the repository.yml file be?

The `repository.yml` file lives only in the master branch of the git repository. Newt will always fetch this file from the master branch and then use that to resolve the actual branch required depending on the version specified in the project. **Special care should be taken to ensure that this file exists only in the master branch.**

Here is the `repository.yml` file from a certain snapshot of apache-Mynewt-core:

```
repo.name: apache-mynewt-core
repo.versions:
    "0.7.9": "Mynewt_0_8_0_b2_tag"
    "0-latest": "0.7.9"
    "0.8-latest": "0.7.9"
```

It contains the following:

- **repo.name** The external name that is used to include the library in your `project.yml` file. This is the name you include in the `project.repositories` variable when adding this repository to your project.
- **repo.versions** A description of what versions to give the user depending on the settings in their `project.yml` file. See below for a thorough description on versioning. Its a flexible mapping between version numbers and git branches.

Repo Version Specification

The version field argument for a repo has the following format:

```
<major_num>.<minor_num>.<revision_num>
```

or

```
<major_num>.<minor_num>-<stability_string>
```

or

```
<major_num>-<stability_string>
```

The stability string can be one of 3 pre-defined stability values.

1. stable – A stable release version of the repository
2. dev – A development version from the repository
3. latest – The latest from the repository

In your `project.yml` file you can specify different combinations of the version number and stability value. For example:

- 1-latest – The latest version with major number 1
- 1.2-stable – The latest stable version with major and minor number 1.2

- 1.2-dev – The development version from 1.2
- 1.1.1 – a specific version 1.1.1

You **cannot** specify a stability string with a fully numbered version, e.g.

```
1.2.8-stable
```

Repo Version Resolution

A `repository.yml` file contains information to match this version request into a git branch to fetch for your project.

It's up to you as the repository maintainer to map these to actual github branches of the repository. For example, let's say in a fictitious repository the following are defined.

```
repo.versions:
  "0.8.0": "xxx_branch_0_8_0"
  "1.0.0": "xxx_branch_1_0_0"
  "1.0.2": "xxx_branch_1_0_2"
  "1.1.1": "xxx_branch_1_1_0"
  "1.1.2": "xxx_branch_1_1_2"
  "1.2.0": "xxx_branch_1_2_0"
  "1.2.1": "xxx_branch_1_2_1"
  "1.2-dev": "1.2.1"
  "1-dev": "1.2-dev"
  "1.2-stable": "1.2.0"
  "0-latest": "0.8.0"
  "1-latest": "1-dev"
  ....
```

When the `project.yml` file asks for `1.2-stable` it will be resolved to version `1.2.0` which in turn will resolve to a specific branch `xxx_branch_1_2_0`. This is the branch that `newt` will fetch into the project with that `project.yml` file.

Dependencies on other repos

Repositories can also have dependencies on other repositories. These dependencies should be listed out on a per-tag basis. So, for example, if `apache-mynewt-core` were to depend on `sterlys-little-repo`, you might have the following directives in the `repository.yml`:

```
develop.repositories:
  sterlys-little-repo:
    type: github
    vers: 0.8-latest
    user: sterlingshughes
    repo: sterlys-little-repo
```

This would tell Newt that for anything that resolves to the `develop` branch, this repository requires the `sterlys-little-repo` repository.

Dependencies are resolved circularly by the `newt` tool, and every dependent repository is placed as a sibling in the `repos` directory. Currently, if two repositories have the same name, they will conflict and bad things will happen.

When a repository is installed to the `repos/` directory, the current version of that repository is written to the “`project.state`” file. The project state file contains the currently installed version of any given repository. This way,

the current set of repositories can be recreated from the project.state file reliably, whereas the project.yml file can have higher level directives (i.e. include 0.8-stable.)

Resolving dependencies

At the moment, all dependencies must match, otherwise newt will provide an error. As an example, if you have a set of dependencies such that:

```
apache-mynewt-core depends on sterlys-little-repo 0.6-stable
apache-mynewt-core depends on sterlys-big-repo 0.5.1
sterlys-big-repo-0.5.1 depends on sterlys-little-repo 0.6.2
```

where 0.6-stable is 0.6.3, the newt tool will try and resolve the dependency to sterlys-little-repo. It will notice that there are two conflicting versions of the same repository, and not perform installation.

In the future Newt will be smarter about loading in all dependencies, and then looking to satisfy those dependencies to the best match of all potential options.

4.4.3 Accessing a private repository

To access a private repository, newt needs to be configured with one of the following:

- Access token for the repository
- Basic auth login and password for the user

NOTE: To create a github access token, see <https://help.github.com/articles/creating-an-access-token-for-command-line-use/>

There are two ways to specify this information, as shown below. In these examples, both a token and a login/password are specified, but you only need to specify one of these.

1. project.yml (probably world-readable and therefore not secure):

```
repository.my-private-repo:
  type: github
  vers: 0-dev
  user: owner-of-repo
  repo: repo-name
  token: '8ab6433f8971b05c2a9c3341533e8ddb754e404e'
  login: githublogin
  password: githubpassword
```

2. \$HOME/.newt/repos.yml

```
repository.my-private-repo:
  token: '8ab6433f8971b05c2a9c3341533e8ddb754e404e'
  login: githublogin
  password: githubpassword
```

If both a token and a login+password are specified, newt uses the token. If both the project.yml file and the private repos.yml file specify security credentials, newt uses the project.yml settings.

NOTE: When newt downloads the actual repo content, as opposed to just the repository.yml file, it does not use the same mechanism. Instead, it invokes the git command line tool. This is an annoyance because the user cannot use the same access token for all git operations. This is something that will be fixed in the future.

4.4.4 Upgrade a repo

In order to upgrade a previously installed repository, the “newt upgrade” command should be issued:

```
$ newt upgrade
```

Newt upgrade will look at the current desired version in `project.yml`, and compare it to the version in `project.state`. If these two differ, it will upgrade the dependency. Upgrade works not just for the dependency in `project.yml`, but for all the sub-dependencies that they might have.

4.5 Project Slinky

The goal of the project is to use a sample application called “Slinky” included in the Mynewt repository to enable remote communications with a device running the Mynewt OS. The protocol for remote communications is called newt manager (`newtmgr`).

If you have an existing project using a target that does not use the Slinky application and you wish to add `newtmgr` functionality to it, check out the tutorial titled [Enabling Newt Manager in Your Application](#)

- [Available Tutorials](#)
- [Prerequisites](#)
- [Overview of Steps](#)

4.5.1 Available Tutorials

Tutorials are available for the following boards:

- [Project Sim Slinky](#). This is supported on Mac OS and Linux platforms.
- [Project Slinky using the Nordic nRF52 Board](#).
- [Project Slinky Using Olimex Board](#).

4.5.2 Prerequisites

Ensure that you meet the following prerequisites before continuing with this tutorial:

- Have Internet connectivity to fetch remote Mynewt components.
- Have a computer to build a Mynewt application and connect to the board over USB.
- Have a Micro-USB cable to connect the board and the computer.
- Have a [serial port setup](#).
- Install the newt tool and the toolchains (See [Basic Setup](#)).
- Install the `newtmgr tool`.
- Read the Mynewt OS [Concepts](#) section.
- Create a project space (directory structure) and populated it with the core code repository (apache-mynewt-core) or know how to as explained in [Creating Your First Project](#).

4.5.3 Overview of Steps

- Install dependencies.
- Define the bootloader and Slinky application target for the target board.
- Build the bootloader target.
- Build the Slinky application target and create an application image.
- Set up serial connection with the targets.
- Create a connection profile using the newtmgr tool.
- Use the newtmgr tool to communicate with the targets.

4.5.4 Project Sim Slinky

This tutorial shows you how to create, build and run the Slinky application and communicate with newtmgr for a simulated device. This is supported on Mac OS and Linux platforms.

- *Prerequisites*
- *Creating a new project*
- *Setting up your target build*
- *Building Your target*
- *Run the target*
- *Setting up a connection profile*
- *Executing newtmgr commands with the target*

Prerequisites

Meet the prerequisites listed in [Project Slinky](#)

Creating a new project

Instructions for creating a project are located in the [Basic Setup](#) section of the [Mynewt Documentation](#)

We will list only the steps here for brevity. We will name the project `slinky`.

```
$ newt new slinky
Downloading project skeleton from apache/mynewt-blinky...
...
Installing skeleton in slinky...
Project slinky successfully created.
$ cd slinky
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
...
```

(continues on next page)

(continued from previous page)

```
apache-mynewt-core successfully upgraded to version 1.7.0
mcuboot successfully upgraded to version 1.3.1
```

Setting up your target build

Create a target for slinky using the native bsp. We will list only the steps and suppress the tool output here for brevity.

```
$ newt target create sim_slinky
$ newt target set sim_slinky bsp=@apache-mynewt-core/hw/bsp/native
$ newt target set sim_slinky build_profile=debug
$ newt target set sim_slinky app=@apache-mynewt-core/apps/slinky
```

Building Your target

To build your target, use `newt build`. When complete, an executable file is created.

```
$ newt build sim_slinky
Building target targets/sim_slinky
Compiling repos/mcuboot/boot/bootutil/src/caps.c
Compiling repos/mcuboot/boot/bootutil/src/encrypted.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/apache-mynewt-core/apps/slinky/src/main.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/image_ed25519.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/apache-mynewt-core/boot/split/src/split_config.c
Compiling repos/mcuboot/boot/bootutil/src/loader.c

...
Archiving @apache-mynewt-core_util_rwlock.a
Archiving @apache-mynewt-core_util_streamer.a
Linking ~/dev/slinky/bin/targets/sim_slinky/app/@apache-mynewt-core/apps/slinky/slinky.
→elf
Target successfully built: targets/sim_slinky
```

Run the target

Run the executable you have build for the simulated environment. The serial port name on which the simulated target is connected is shown in the output when mynewt slinky starts.

```
$ ~/dev/slinky/bin/targets/sim_slinky/app/apps/slinky/slinky.elf
uart0 at /dev/ttys005
```

In this example, the slinky app opened up a com port `/dev/ttys005` for communications with newtmgr.

NOTE: This application will block. You will need to open a new console (or execute this in another console) to continue the tutorial.*

Setting up a connection profile

You will now set up a connection profile using `newtmgr` for the serial port connection and start communicating with the simulated remote device.

```
$ newtmgr conn add sim1 type=serial connstring=/dev/ttys005
Connection profile sim1 successfully added
$ newtmgr conn show
Connection profiles:
    sim1: type=serial, connstring='/dev/ttys005'
```

Executing `newtmgr` commands with the target

You can now use connection profile `sim1` to talk to the running `sim_slinky`. As an example, we will query the running mynewt OS for the usage of its memory pools.

```
$ newtmgr -c sim1 mpstat
Return Code = 0
          name blksz  cnt  free   min
      msys_1     292    12    10    10
```

As a test command, you can send an arbitrary string to the target and it will echo that string back in a response to `newtmgr`.

```
$ newtmgr -c sim1 echo "Hello Mynewt"
Hello Mynewt
```

In addition to these, you can also examine running tasks, statistics, logs, image status (not on sim), and configuration.

4.5.5 Project Slinky using the Nordic nRF52 Board

This tutorial shows you how to create, build and run the Slinky application and communicate with `newtmgr` for a Nordic nRF52 board.

- *Prerequisites*
- *Create a New Project*
- *Create the Targets*
- *Build the Targets*
- *Sign and Create the Slinky Application Image*
- *Connect to the Board*
- *Load the Bootloader and the Slinky Application Image*
- *Connect Newtmgr with the Board using a Serial Connection*
- *Use Newtmgr to Query the Board*

Prerequisites

- Meet the prerequisites listed in [Project Slinky](#)
- Have a Nordic nRF52-DK board.
- Install the [Segger JLINK Software](#) and documentation pack.

Create a New Project

Create a new project if you do not have an existing one. You can skip this step and proceed to [create the targets](#) if you already have a project created or completed the [Sim Slinky](#) tutorial.

Run the following commands to create a new project. We name the project `slinky`.

```
$ newt new slinky
Downloading project skeleton from apache/mynewt-blinky...
...
Installing skeleton in slinky...
Project slinky successfully created
$ cd slinky
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
...
mcuboot successfully upgraded to version 1.3.1
```

Create the Targets

Create two targets for the nRF52-DK board - one for the bootloader and one for the Slinky application.

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `nrf52_boot`.

```
$ newt target create nrf52_boot
$ newt target set nrf52_boot bsp=@apache-mynewt-core/hw/bsp/nrf52dk
$ newt target set nrf52_boot build_profile=optimized
$ newt target set nrf52_boot app=@mcuboot/boot/mynewt
```

Run the following `newt target` commands to create a target for the Slinky application. We name the target `nrf52_slinky`.

```
$ newt target create nrf52_slinky
$ newt target set nrf52_slinky bsp=@apache-mynewt-core/hw/bsp/nrf52dk
$ newt target set nrf52_slinky build_profile=debug
$ newt target set nrf52_slinky app=@apache-mynewt-core/apps/slinky
```

Build the Targets

Run the `newt build nrf52_boot` command to build the bootloader:

```
$ newt build nrf52_boot
Building target targets/nrf52_boot
Compiling repos/mcuboot/boot/bootutil/src/caps.c
Compiling repos/mcuboot/boot/bootutil/src/encrypted.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/apache-mynewt-core/apps/slinky/src/main.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/image_ed25519.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/apache-mynewt-core/boot/split/src/split_config.c
...
Archiving @apache-mynewt-core_util_rwlock.a
Archiving @apache-mynewt-core_util_streamer.a
Linking ~/dev/slinky/bin/targets/nrf52_boot/app/@mcuboot/boot/mynewt/mynewt.elf
Target successfully built: targets/nrf52_boot
```

Run the `newt build nrf52_slinky` command to build the Slinky application:

```
$newt build nrf52_slinky
Building target targets/nrf52_slinky
Compiling repos/mcuboot/boot/bootutil/src/caps.c
Compiling repos/mcuboot/boot/bootutil/src/encrypted.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/apache-mynewt-core/apps/slinky/src/main.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/image_ed25519.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/apache-mynewt-core/boot/split/src/split_config.c
...
Archiving @apache-mynewt-core_util_rwlock.a
Archiving @apache-mynewt-core_util_streamer.a
Linking ~/dev/slinky/bin/targets/nrf52_slinky/app/@apache-mynewt-core/apps/slinky/slinky.
→elf
Target successfully built: targets/nrf52_slinky
```

Sign and Create the Slinky Application Image

Run the `newt create-image nrf52_slinky 1.0.0` command to create and sign the application image. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image nrf52_slinky 1.0.0
App image successfully generated: ~/dev/slinky/bin/targets/nrf52_slinky/app/@apache-
→mynewt-core/apps/slinky/slinky.img
$
```

Connect to the Board

- Connect a micro-USB cable from your computer to the micro-USB port on the nRF52-DK board.
- Turn the power on the board to ON. You should see the green LED light up on the board.

Load the Bootloader and the Slinky Application Image

Run the `newt load nrf52_boot` command to load the bootloader onto the board:

```
$ newt load nrf52_boot
Loading bootloader
$
```

Run the `newt load nrf52_slinky` command to load the Slinky application image onto the board:

```
$ newt load nrf52_slinky
Loading app image into slot 1
$
```

Connect Newtmgr with the Board using a Serial Connection

Set up a serial connection from your computer to the nRF52-DK board (See [Serial Port Setup](#)).

Locate the port, in the /dev directory on your computer, that the serial connection uses. The format of the port name is platform dependent:

- Mac OS uses the format `tty.usbserial-<some identifier>`.
- Linux uses the format `TTYUSB<N>`, where `N` is a number. For example, `TTYUSB2`.
- MinGW on Windows uses the format `ttyS<N>`, where `N` is a number. You must map the port name to a Windows COM port: `/dev/ttyS<N>` maps to `COM<N+1>`. For example, `/dev/ttyS2` maps to `COM3`.

You can also use the Windows Device Manager to find the COM port number.

```
$ ls /dev/tty*usbserial*
/dev/tty.usbserial-1d11
$
```

Setup a newtmgr connection profile for the serial port. For our example, the port is `/dev/tty.usbserial-1d11`.

Run the `newtmgr conn add` command to define a newtmgr connection profile for the serial port. We name the connection profile `nrf52serial`.

Note:

- You will need to replace the `connstring` with the specific port for your serial connection.
- On Windows, you must specify `COM<N+1>` for the `connstring` if `/dev/ttyS<N>` is the serial port.

```
$ newtmgr conn add nrf52serial type=serial connstring=/dev/tty.usbserial-1d11
Connection profile nrf52serial successfully added
$
```

You can run the `newt conn show` command to see all the newtmgr connection profiles:

```
$ newtmgr conn show
Connection profiles:
  nrf52serial: type=serial, connstring='/dev/tty.usbserial-1d11'
  sim1: type=serial, connstring='/dev/ttys012'
$
```

Use Newtmgr to Query the Board

Run some newtmgr commands to query and receive responses back from the board (See the [Newt Manager Guide](#) for more information on the newtmgr commands)

Run the `newtmgr echo hello -c nrf52serial` command. This is the simplest command that requests the board to echo back the text.

```
$ newtmgr echo hello -c nrf52serial
hello
$
```

Run the `newtmgr image list -c nrf52serial` command to list the images on the board:

```
$ newtmgr image list -c nrf52serial
Images:
slot=0
    version: 1.0.0
    bootable: true
    flags: active confirmed
    hash: f411a55d7a5f54eb8880d380bf47521d8c41ed77fd0a7bd5373b0ae87ddabd42
Split status: N/A
$
```

Run the `newtmgr taskstat -c nrf52serial` command to display the task statistics on the board:

```
$ newtmgr taskstat -c nrf52serial
  task pri tid  runtime      csw    stksz   stkuse last_checkin next_checkin
  idle 255  0    43484      539     64        32       0         0
  main 127  1      1      90    1024      353       0         0
  task1  8   2      0     340      192      114       0         0
  task2  9   3      0     340       64       31       0         0
$
```

4.5.6 Project Slinky Using Olimex Board

This tutorial shows you how to create, build and run the Slinky application and communicate with newtmgr for an Olimex STM-E407 board.

- [Prerequisites](#)
- [Create a New Project](#)
- [Create the Targets](#)

- *Build the Targets*
- *Sign and Create the Slinky Application Image*
- *Connect to the Board*
- *Load the Bootloader and the Slinky Application Image*
- *Connect Newtmgr with the Board using a Serial Connection*
- *Use Newtmgr to Query the Board*

Prerequisites

- Meet the prerequisites listed in [Project Slinky](#)
- Have a STM32-E407 development board from Olimex.
- Have a ARM-USB-TINY-H connector with JTAG interface for debugging ARM microcontrollers (comes with the ribbon cable to hook up to the board)
- Have a USB A-B type cable to connect the debugger to your computer.
- Have a USB to TTL Serial Cable with female wiring harness.
- Install the [OpenOCD debugger](#).

Create a New Project

Create a new project if you do not have an existing one. You can skip this step and proceed to *Create the Targets* if you already have a project created or completed the [Sim Slinky](#) tutorial.

```
$ newt new slinky
Downloading project skeleton from apache/mynewt-blinky...
...
Installing skeleton in slinky...
Project slink successfully created
$ cd slinky
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
...
apache-mynewt-core successfully upgraded to version 1.7.0
mcuboot successfully upgraded to version 1.3.1
```

Create the Targets

Create two targets for the STM32-E407 board - one for the bootloader and one for the Slinky application.

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `stm32_boot`.

```
$ newt target create stm32_boot
$ newt target set stm32_boot bsp=@apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard
$ newt target set stm32_boot build_profile=optimized
$ newt target set stm32_boot target.app=@mcuboot/boot/mynewt
```

Run the following `newt target` commands to create a target for the Slinky application. We name the target `stm32_slinky`.

```
$ newt target create stm32_slinky
$ newt target set stm32_slinky bsp=@apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard
$ newt target set stm32_slinky build_profile=debug
$ newt target set stm32_slinky app=@apache-mynewt-core/apps/slinky
```

Build the Targets

Run the `newt build stm32_boot` command to build the bootloader:

```
$ newt build stm32_boot
Building target targets/stm32_boot
Compiling repos/mcuboot/boot/bootutil/src/caps.c
Compiling repos/mcuboot/boot/bootutil/src/encrypted.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/apache-mynewt-core/apps/slinky/src/main.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/image_ed25519.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/apache-mynewt-core/boot/split/src/split_config.c
...
Archiving @apache-mynewt-core_util_rwlock.a
Archiving @apache-mynewt-core_util_streamer.a
Linking ~/dev/slinky/bin/targets/stm32_boot/app/@mcuboot/boot/mynewt/mynewt.elf
Target successfully built: targets/stm32_boot
$
```

Run the `newt build stm32_slinky` command to build the Slinky application:

```
$newt build stm32_slinky
Building target targets/stm32_slinky
Compiling repos/mcuboot/boot/bootutil/src/caps.c
Compiling repos/mcuboot/boot/bootutil/src/encrypted.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/apache-mynewt-core/apps/slinky/src/main.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/image_ed25519.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/apache-mynewt-core/boot/split/src/split_config.c
...
Archiving @apache-mynewt-core_util_rwlock.a
Archiving @apache-mynewt-core_util_streamer.a
Linking ~/dev/slinky/bin/targets/stm32_slinky/app/@apache-mynewt-core/apps/slinky/slinky.elf
Target successfully built: targets/stm32_slinky
$
```

Sign and Create the Slinky Application Image

Run the `newt create-image stm32_slinky 1.0.0` command to create and sign the application image. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image stm32_slinky 1.0.0
App image successfully generated: ~/dev/slinky/bin/targets/stm32_slinky/app/apps/slinky/
˓→slinky.img
$
```

Connect to the Board

- Connect the USB A-B type cable to the ARM-USB-TINY-H debugger connector.
- Connect the ARM-USB-Tiny-H debugger connector to your computer and the board.
- Connect the USB Micro-A cable to the USB-OTG2 port on the board.
- Set the Power Sel jumper on the board to pins 5 and 6 to select USB-OTG2 as the power source. If you would like to use a different power source, refer to the [OLIMEX STM32-E407 user manual](#) for pin specifications.

You should see a red LED light up on the board.

Load the Bootloader and the Slinky Application Image

Run the `newt load stm32_boot` command to load the bootloader onto the board:

```
$ newt load stm32_boot
Loading bootloader
$
```

Note: If you are using Windows and get a `no device found` error, you will need to install the usb driver. Download [Zadig](#) and run it:

- Select Options > List All Devices.
- Select Olimex OpenOCD JTAG ARM-USB-TINY-H from the drop down menu.
- Select the WinUSB driver.
- Click Install Driver.
- Run the `newt load stm32_boot` command again.

Run the `newt load stm32_slinky` command to load the Slinky application image onto the board:

```
$ newt load stm32_slinky
Loading app image into slot 1
$
```

Connect Newtmgr with the Board using a Serial Connection

Locate the PC6/USART6_TX (pin 3), PC7/USART6_RX (pin 4), and GND (pin 2) of the UEXT connector on the Olimex board. More information on the UEXT connector can be found at <https://www.olimex.com/Products/Modules/UEXT/>. The schematic of the board can be found at https://www.olimex.com/Products/ARM/ST/STM32-E407/resources/STM32-E407_sch.pdf for reference.

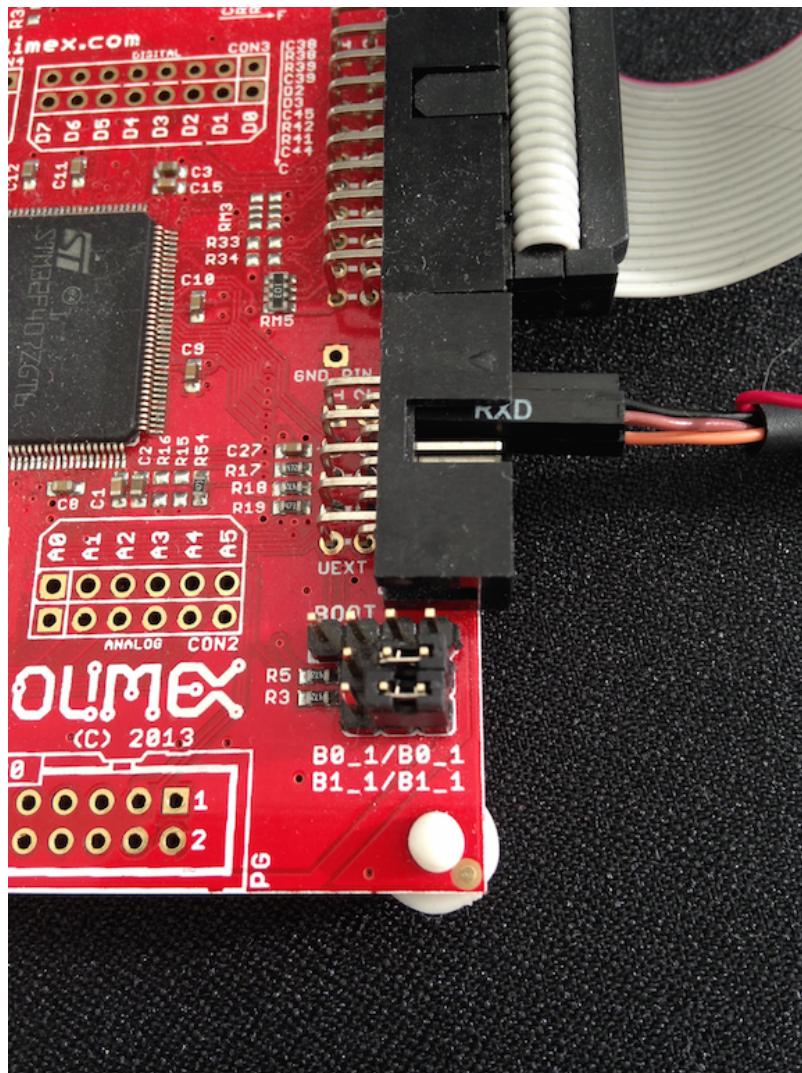


Fig. 2: Alt Layout - Serial Connection

- Connect the female RX pin of the USB-TTL serial cable to the TX (Pin 3) of the UEXT connector on the board.
- Connect the female TX pin of the USB-TTL serial cable to the RX (Pin 4) of the UEXT connector on the board.
- Connect the GND pin of the USB-TTL serial cable to the GND (Pin 2) of the UEXT connector on the board.

Locate the port, in the /dev directory on your computer, that the serial connection uses. The format of the port name is platform dependent:

- Mac OS uses the format `tty.usbserial-<some identifier>`.
- Linux uses the format `PTYUSB<N>`, where N is a number. For example, `PTYUSB2`.

- MinGW on Windows uses the format `ttyS<N>`, where `N` is a number. You must map the port name to a Windows COM port: `/dev/ttyS<N>` maps to `COM<N+1>`. For example, `/dev/ttyS2` maps to `COM3`.

You can also use the Windows Device Manager to find the COM port number.

```
$ ls /dev/tty*usbserial*
/dev/tty.usbserial-1d13
$
```

Setup a newtmgr connection profile for the serial port. For our example, the port is `/dev/tty.usbserial-1d13`.

Run the `newtmgr conn add` command to define a newtmgr connection profile for the serial port. We name the connection profile `stm32serial`.

Note:

- You will need to replace the `connstring` with the specific port for your serial connection.
- On Windows, you must specify `COM<N+1>` for the `connstring` if `/dev/ttyS<N>` is the serial port.

```
$ newtmgr conn add stm32serial type=serial connstring=/dev/tty.usbserial-1d13
Connection profile stm32serial successfully added
$
```

You can run the `newt conn show` command to see all the newtmgr connection profiles:

```
$ newtmgr conn show
Connection profiles:
  stm32serial: type=serial, connstring='/dev/tty.usbserial-1d13'
  sim1: type=serial, connstring='/dev/ttys012'
$
```

Use Newtmgr to Query the Board

Run some newtmgr commands to query and receive responses back from the board (See the [Newt Manager Guide](#) for more information on the newtmgr commands).

Run the `newtmgr echo hello -c stm32serial` command. This is the simplest command that requests the board to echo back the text.

```
$ newtmgr echo hello -c stm32serial
hello
$
```

Run the `newtmgr image list -c stm32serial` command to list the images on the board:

```
$ newtmgr image list -c stm32serial
Images:
slot=0
  version: 1.0.0
  bootable: true
  flags: active confirmed
  hash: 9cf8af22b1b573909a8290a90c066d4e190407e97680b7a32243960ec2bf3a7f
Split status: N/A
$
```

Run the `newtmgr taskstat -c stm32serial` command to display the task statistics on the board:

```
$ newtmgr taskstat -c stm32serial
  task pri tid runtime      csw    stksz   stkuse last_checkin next_checkin
  idle 255  0    157179  157183       64      25      0      0
  main 127  1        4      72    1024     356      0      0
task1  8    2        0     158     192     114      0      0
task2  9    3        0     158      64      30      0      0
$
```

4.6 Bluetooth Low Energy

The tutorials in this section will help you get started with Mynewt's BLE stack called "NimBLE". The first tutorial simply explains how to define a new (albeit empty) application for your target. The tutorials get progressively complex, explaining how to write your own beacon, leverage an example application from the repository to make your own end device (peripheral), use the HCI interface etc.

These tutorials assume that you have already installed the newt tool and are familiar with its concepts. And as always, contributions to this section are very welcome!

4.6.1 Set up a bare bones NimBLE application

This tutorial explains how to set up a minimal application using the NimBLE stack. It assumes that you have already installed the newt tool and are familiar with its concepts.

- *Create a Mynewt project*
- *Create an application package*
- *Create the target*
- *Enter BLE*
- *Build the target*
- *Conclusion*

Create a Mynewt project

We start by creating a project space for your own application work using the Newt tool. We will call our project `my_proj`.

```
~/dev$ newt new my_proj1
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in my_proj1...
Project my_proj1 successfully created.
```

The above command created the following directory tree:

```
~/dev$ tree my_proj1
my_proj1
├── DISCLAIMER
```

(continues on next page)

(continued from previous page)

```

LICENSE
NOTICE
README.md
apps
└── blinky
    ├── pkg.yml
    └── src
        └── main.c
project.yml
targets
└── my_blinky_sim
    ├── pkg.yml
    └── target.yml
unittest
└── pkg.yml
    └── target.yml

```

6 directories, 11 files

Next, we need to retrieve the Mynewt repositories that our app will depend on. When you retrieve a repository, your project gains access to the libraries and applications that the repo contains.

A new project automatically depends on the Apache Mynewt core repo (`apache-mynewt-core`). The core repo contains the Apache Mynewt operating system, NimBLE stack, and other system libraries. Later, our app may need packages from additional repos, but for now the core repo suits our needs.

We download the dependent repos using the `newt upgrade` command:

```

~/dev$ cd my_proj1
~/dev/my_proj1$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
apache-mynewt-core successfully upgraded to version 1.7.0

```

Now it's time to create your own app.

Create an application package

```

~/dev/my_proj1$ newt pkg new apps/ble_app -t app
Download package template for package type app.
Package successfully installed into /home/me/dev/my_proj1/apps/ble_app.

```

You now have an application called `apps/ble_app`. It isn't terribly interesting as far as applications go, but it does all the configuration and set up required of a Mynewt app. It will be a useful starting point for our BLE application.

Create the target

Now you have to create the target that ties your application to a BSP. We will call this target “ble_tgt”.

```
~/dev/my_proj1$ newt target create ble_tgt
Target targets/ble_tgt successfully created
```

We now have a new target:

```
~/dev/my_proj1$ tree targets/ble_tgt
targets/ble_tgt
└── pkg.yml
    └── target.yml
```

We need to fill out a few details in our target before it is usable. At a minimum, a target must specify three bits of information:

- Application package
- BSP package
- Build profile

The application package is straightforward; this is the ble_app package that we created in the previous step.

For the BSP package, this tutorial chooses to target the nRF52dk BSP. If you would like to use a different platform, substitute the name of the appropriate BSP package in the command below.

Finally, the build profile specifies the set of compiler and linker options to use during the build. Apache Mynewt supports two build profiles: `debug` and `optimized`.

```
~/dev/my_proj1$ newt target set ble_tgt \
    app=apps/ble_app \
    bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040 \
    build_profile=optimized
Target targets/ble_tgt successfully set target.app to apps/ble_app
Target targets/ble_tgt successfully set target.bsp to @apache-mynewt-core/hw/bsp/nordic_
˓→pca10040
Target targets/ble_tgt successfully set target.build_profile to optimized
```

Enter BLE

Since our application will support BLE functionality, we need to give it access to a BLE stack. We do this by adding the necessary NimBLE packages to the app’s dependency list in `apps/ble_app/pkg.yml`:

```
pkg.deps:
- "@apache-mynewt-core/kernel/os"
- "@apache-mynewt-core/sys/console/full"
- "@apache-mynewt-core/sys/log/full"
- "@apache-mynewt-core/sys/stats/full"
- "@apache-mynewt-nimble/nimble/host"
- "@apache-mynewt-nimble/nimble/host/store/config"
- "@apache-mynewt-nimble/nimble/transport"
```

To enable a combined host-controller in the app set correct transport which will include NimBLE controller into build. For this update `apps/ble_app/syscfg.yml`

```
syscfg.vals:
    BLE_HCI_TRANSPORT: builtin
```

Your `main.c` will also need to include the nimble host header:

```
#include "host/ble_hs.h"
```

Important note: The controller package affects system configuration, see [this page](#) for details.

Build the target

Now would be a good time for a basic sanity check. Let's make sure the target builds.

```
~/dev/my_proj1$ newt build ble_tgt
Building target targets/ble_tgt
Compiling repos/apache-mynewt-core/hw/hal/src/hal_common.c
Compiling repos/apache-mynewt-core/hw/drivers/uart/src/uart.c
<...snip...
Linking /home/me/dev/my_proj1/bin/targets/ble_tgt/app/apps/ble_app/ble_app.elf
Target successfully built: targets/ble_tgt
```

Now let's try running our minimal application on actual hardware. Attach the target device to your computer and run the application with `newt run`:

```
~/dev/my_proj1$ newt run ble_tgt @
App image successfully generated: /home/me/dev/my_proj1/bin/targets/ble_tgt/app/apps/ble_
 ↲app/ble_app.img
<...snip...
Resetting target
[Switching to Thread 57005]
0x000000dc in ?? ()
(gdb)
```

You can start the application by pressing `c <enter>` at the `gdb` prompt. When the excitement of watching the idle loop run wears off, quit `gdb` with `<ctrl-c> q <enter>`.

If your target fails to build or run, you might want to revisit the [project blinky tutorial](#) to see if there is a setup step you missed. You may also find help by posting a question to the [mailing list](#) or searching the archives.

Conclusion

You now have a fully functional BLE app (never mind that it doesn't actually do anything yet!). With all the necessary infrastructure in place, you can now start turning this into a real application. A good next step would be to turn your app into a beaconing device. The [BLE iBeacon tutorial](#) builds on this one and ends with a functioning iBeacon. For something a little more ambitious, the [BLE peripheral project tutorial](#) describes a NimBLE peripheral application in detail.

4.6.2 BLE iBeacon

This tutorial guides you through the process of creating an iBeacon application using NimBLE and Mynewt. At the conclusion of this tutorial, you will have a fully functional iBeacon app.

- *iBeacon Protocol*
- *Create an Empty BLE Application*
- *Add beaconing*
 - 1. *Wait for host-controller sync*
 - 2. *Configure the NimBLE stack with an address*
 - 3. *Advertise indefinitely*
- *Conclusion*
- *Source Listing*

iBeacon Protocol

A beaconing device announces its presence to the world by broadcasting advertisements. The iBeacon protocol is built on top of the standard BLE advertisement specification. [This page](#) provides a good summary of the iBeacon sub-fields.

Create an Empty BLE Application

This tutorial picks up where the [BLE bare bones application tutorial](#) document concludes. The first step in creating a beaconing device is to create an empty BLE app, as explained in that tutorial. Before proceeding, you should have:

- An app called “ble_app”.
- A target called “ble_tgt”.
- Successfully executed the app on your target device.

Add beaconing

Here is a brief specification of how we want our beaconing app to behave:

1. Wait until the host and controller are in sync.
2. Configure the NimBLE stack with an address to put in its advertisements.
3. Advertise indefinitely.

Let's take these one at a time.

1. Wait for host-controller sync

The first step, waiting for host-controller-sync, is mandatory in all BLE applications. The NimBLE stack is inoperable while the two components are out of sync. In a combined host-controller app, the sync happens immediately at startup. When the host and controller are separate, sync typically occurs in less than a second.

We achieve this by configuring the NimBLE host with a callback function that gets called when sync takes place:

```
static
void ble_app_set_addr()
{ }

static void ble_app_advertise()
{ }

static void ble_app_on_sync(void)
{
    /* Generate a non-resolvable private address. */
    ble_app_set_addr();

    /* Advertise indefinitely. */
    ble_app_advertise();
}

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

    ble_hs_cfg.sync_cb = ble_app_on_sync;

    /* As the last thing, process events from default event queue. */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }
}
```

`ble_hs_cfg.sync_cb` points to the function that should be called when sync occurs. Our callback function, `ble_app_on_sync()`, kicks off the control flow that we specified above. Now we need to fill in the two stub functions.

2. Configure the NimBLE stack with an address

A BLE device needs an address to do just about anything. Some devices have a public Bluetooth address burned into them, but this is not always the case. Furthermore, the NimBLE controller might not know how to read an address out of your particular hardware. For a beaconing device, we generally don't care what address gets used since nothing will be connecting to us.

A reliable solution is to generate a *non-resolvable private address* (nRPA) each time the application runs. Such an address contains no identifying information, and they are expected to change frequently.

```
static void
ble_app_set_addr(void)
{ }
```

(continues on next page)

(continued from previous page)

```

ble_addr_t addr;
int rc;

rc = ble_hs_id_gen_rnd(1, &addr);
assert(rc == 0);

rc = ble_hs_id_set_rnd(addr.val);
assert(rc == 0);
}

static void
ble_app_advertise();
{ }

static void
ble_app_on_sync(void)
{
    /* Generate a non-resolvable private address. */
    ble_app_set_addr();

    /* Advertise indefinitely. */
    ble_app_advertise();
}

```

Our new function, `ble_app_set_addr()`, makes two calls into the stack:

- `ble_hs_id_gen_rnd`: Generate an nRPA.
- `ble_hs_id_set_rnd`: Configure NimBLE to use the newly-generated address.

You can click either of the function names for more detailed documentation.

3. Advertise indefinitely

The first step in advertising is to configure the host with advertising data. This operation tells the host what data to use for the contents of its advertisements. The NimBLE host provides a special helper function for configuring iBeacon advertisement data: `ble_ibeacon_set_adv_data`

If you follow the API link, you'll see that this function takes four parameters: a 128-bit UUID, a major version, a minor version, and a RSSI value. This corresponds with the iBeacon specification, as these three items are the primary components in an iBeacon advertisement.

For now, we'll advertise the following:

- *UUID*: 11:11:11:11:11:11:11:11:11:11:11:11:11:11:11:11
- *Major*: 2
- *Minor*: 10

```

static void
ble_app_advertise(void)
{
    uint8_t uuid128[16];
    int rc;

```

(continues on next page)

(continued from previous page)

```

/* Fill the UUID buffer with a string of 0x11 bytes. */
memset(uuid128, 0x11, sizeof uuid128);

/* Major version=2; minor version=10. */
rc = ble_ibeacon_set_adv_data(uuid128, 2, 10, 0);
assert(rc == 0);

/* TODO: Begin advertising. */
}

```

Now that the host knows what to advertise, the next step is to actually begin advertising. The function to initiate advertising is: `ble_gap_adv_start`. This function takes several parameters. For simplicity, we reproduce the function prototype here:

```

int
ble_gap_adv_start(
    uint8_t own_addr_type,
    const ble_addr_t *direct_addr,
    int32_t duration_ms,
    const struct ble_gap_adv_params *adv_params,
    ble_gap_event_fn *cb,
    void *cb_arg
)

```

This function gives an application quite a bit of freedom in how advertising is to be done. The default values are mostly fine for our simple beaconing application. We will pass the following values to this function:

Parameter	Value	Notes
own_addr_type	BLE_OWN_ADDR_RANDOM	Use the nRPA we generated earlier.
direct_addr	NULL	We are broadcasting, not targeting a peer.
duration_ms	BLE_HS_FOREVER	Advertise indefinitely.
adv_params	defaults	Can be used to specify low level advertising parameters.
cb	NULL	We are non-connectable, so no need for an event callback.
cb_arg	NULL	No callback implies no callback argument.

These arguments are mostly self-explanatory. The exception is `adv_params`, which can be used to specify a number of low-level parameters. For a beaconing application, the default settings are appropriate. We specify default settings by providing a zero-filled instance of the `ble_gap_adv_params` struct as our argument.

```

static void
ble_app_advertise(void)
{
    struct ble_gap_adv_params adv_params;
    uint8_t uuid128[16];
    int rc;

    /* Arbitrarily set the UUID to a string of 0x11 bytes. */
    memset(uuid128, 0x11, sizeof uuid128);

    /* Major version=2; minor version=10. */
}

```

(continues on next page)

(continued from previous page)

```

rc = ble_ibeacon_set_adv_data(uuid128, 2, 10, 0);
assert(rc == 0);

/* Begin advertising. */
adv_params = (struct ble_gap_adv_params){ 0 };
rc = ble_gap_adv_start(BLE_OWN_ADDR_RANDOM, NULL, BLE_HS_FOREVER,
                      &adv_params, NULL, NULL);
assert(rc == 0);

}

```

Conclusion

That's it! Now when you run this app on your board, you should be able to see it with all your iBeacon-aware devices. You can test it out with the `newt run` command.

Source Listing

For reference, here is the complete application source:

```

#include "sysinit/sysinit.h"
#include "os/os.h"
#include "console/console.h"
#include "host/ble_hs.h"

static void
ble_app_set_addr(void)
{
    ble_addr_t addr;
    int rc;

    rc = ble_hs_id_gen_rnd(1, &addr);
    assert(rc == 0);

    rc = ble_hs_id_set_rnd(addr.val);
    assert(rc == 0);
}

static void
ble_app_advertise(void)
{
    struct ble_gap_adv_params adv_params;
    uint8_t uuid128[16];
    int rc;

    /* Arbitrarily set the UUID to a string of 0x11 bytes. */
    memset(uuid128, 0x11, sizeof(uuid128));

    /* Major version=2; minor version=10. */
    rc = ble_ibeacon_set_adv_data(uuid128, 2, 10, 0);
}

```

(continues on next page)

(continued from previous page)

```

assert(rc == 0);

/* Begin advertising. */
adv_params = (struct ble_gap_adv_params){ 0 };
rc = ble_gap_adv_start(BLE_OWN_ADDR_RANDOM, NULL, BLE_HS_FOREVER,
                      &adv_params, NULL, NULL);
assert(rc == 0);
}

static void
ble_app_on_sync(void)
{
    /* Generate a non-resolvable private address. */
    ble_app_set_addr();

    /* Advertise indefinitely. */
    ble_app_advertise();
}

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

    ble_hs_cfg.sync_cb = ble_app_on_sync;

    /* As the last thing, process events from default event queue. */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }
}

```

4.6.3 BLE Eddystone

Eddystone Beacon Protocol

A beaconing device announces its presence to the world by broadcasting advertisements. The Eddystone protocol is built on top of the standard BLE advertisement specification. Eddystone supports multiple data packet types:

- Eddystone-UID: a unique, static ID with a 10-byte Namespace component and a 6-byte Instance component.
- Eddystone-URL: a compressed URL that, once parsed and decompressed, is directly usable by the client.
- Eddystone-TLM: “telemetry” packets that are broadcast alongside the Eddystone-UID or Eddystone-URL packets and contains beacon’s “health status” (e.g., battery life).
- Eddystone-EID to broadcast an ephemeral identifier that changes every few minutes and allow only parties that can resolve the identifier to use the beacon.

[This page](#) describes the Eddystone open beacon format developed by Google.

Apache Mynewt currently supports Eddystone-UID and Eddystone-URL formats only. This tutorial will explain how to get an Eddystone-URL beacon going on a peripheral device.

Create an Empty BLE Application

This tutorial picks up where the [BLE bare bones application tutorial](#) concludes. The first step in creating a beaconing device is to create an empty BLE app, as explained in that tutorial. Before proceeding, you should have:

- An app called “ble_app”.
- A target called “ble_tgt”.
- Successfully executed the app on your target device.

Add beaconing

Here is a brief specification of how we want our beaconing app to behave:

1. Wait until the host and controller are in sync.
2. Configure the NimBLE stack with an address to put in its advertisements.
3. Advertise an eddystone URL beacon indefinitely.

Let's take these one at a time.

1. Wait for host-controller sync

The first step, waiting for host-controller-sync, is mandatory in all BLE applications. The NimBLE stack is inoperable while the two components are out of sync. In a combined host-controller app, the sync happens immediately at startup. When the host and controller are separate, sync typically occurs in less than a second.

We achieve this by configuring the NimBLE host with a callback function that gets called when sync takes place:

```
static void  
ble_app_set_addr()  
{ }  
  
static void  
ble_app_advertise()  
{ }  
  
static void  
ble_app_on_sync(void)  
{  
    /* Generate a non-resolvable private address. */  
    ble\_\_app\_\_set\_\_addr();  
  
    /* Advertise indefinitely. */  
    ble_app_advertise();  
}  
  
int  
main(int argc, char \*\*argv)  
{  
    sysinit();  
  
    ble_hs_cfg.sync_cb = ble_app_on_sync;
```

(continues on next page)

(continued from previous page)

```
/* As the last thing, process events from default event queue. */
while (1) {
    os_eventq_run(os_eventq_dflt_get());
}
}
```

`ble_hs_cfg.sync_cb` points to the function that should be called when sync occurs. Our callback function, `ble_app_on_sync()`, kicks off the control flow that we specified above. Now we need to fill in the two stub functions.

2. Configure the NimBLE stack with an address

A BLE device needs an address to do just about anything. Some devices have a public Bluetooth address burned into them, but this is not always the case. Furthermore, the NimBLE controller might not know how to read an address out of your particular hardware. For a beacons device, we generally don't care what address gets used since nothing will be connecting to us.

A reliable solution is to generate a *non-resolvable private address* (nRPA) each time the application runs. Such an address contains no identifying information, and they are expected to change frequently.

```
static void
ble_app_set_addr(void)
{
    ble_addr_t addr;
    int rc;

    rc = ble_hs_id_gen_rnd(1, &addr);
    assert(rc == 0);

    rc = ble_hs_id_set_rnd(addr.val);
    assert(rc == 0);
}

static void ble_app_advertise()
{ }

static void ble_app_on_sync(void)
{
    /* Generate a non-resolvable private address. */
    ble_app_set_addr();

    /* Advertise indefinitely. */
    ble_app_advertise();
}
```

Our new function, `ble_app_set_addr()`, makes two calls into the stack:

- `ble_hs_id_gen_rnd` : Generate an nRPA.
- `ble_hs_id_set_rnd` : Configure NimBLE to use the newly-generated address.

You can click either of the function names for more detailed documentation.

3. Advertise indefinitely

The first step in advertising is to configure the host with advertising data. This operation tells the host what data to use for the contents of its advertisements. The NimBLE host provides special helper functions for configuring eddystone advertisement data:

- `ble_eddystone_set_adv_data_uid`
- `ble_eddystone_set_adv_data_url`

Our application will advertise eddystone URL beacons, so we are interested in the second function. We reproduce the function prototype here:

```
int
ble_eddystone_set_adv_data_url(
    struct ble_hs_adv_fields *adv_fields,
    uint8_t url_scheme,
    char *url_body,
    uint8_t url_body_len,
    uint8_t url_suffix
    int8_t measured_power
)
```

We'll advertise the Mynewt URL: <https://mynewt.apache.org>. Eddystone beacons use a form of URL compression to accommodate the limited space available in Bluetooth advertisements. The `url_scheme` and `url_suffix` fields implement this compression; they are single byte fields which correspond to strings commonly found in URLs. The following arguments translate to the <https://mynewt.apache.org> URL:

Parameter	Value
url_scheme	BLE_EDDYSTONE_URL_SCHEME_HTTPS
url_body	“mynewt.apache”
url_suffix	BLE_EDDYSTONE_URL_SUFFIX_ORG

```
static void
ble_app_advertise(void)
{
    struct ble_hs_adv_fields fields;
    int rc;

    /* Configure an eddystone URL beacon to be advertised;
     * URL: https://apache.mynewt.org
     */
    fields = (struct ble_hs_adv_fields){ 0 };
    rc = ble_eddystone_set_adv_data_url(&fields,
                                         BLE_EDDYSTONE_URL_SCHEME_HTTPS,
                                         "mynewt.apache",
                                         13,
                                         BLE_EDDYSTONE_URL_SUFFIX_ORG,
                                         0);
    assert(rc == 0);

    /* TODO: Begin advertising. */
}
```

Now that the host knows what to advertise, the next step is to actually begin advertising. The function to initiate advertising is: `ble_gap_adv_start. This function takes several parameters. For simplicity, we reproduce the function prototype here:

```
int
ble_gap_adv_start(
    uint8_t own_addr_type,
    const ble_addr_t *direct_addr,
    int32_t duration_ms,
    const struct ble_gap_adv_params *adv_params,
    ble_gap_event_fn *cb,
    void *cb_arg
)
```

This function gives an application quite a bit of freedom in how advertising is to be done. The default values are mostly fine for our simple beaconing application. We will pass the following values to this function:

Parameter	Value	Notes
own_addr_type	BLE_OWN_ADDR_RANDOM	Use the nRPA we generated earlier.
direct_addr	NULL	We are broadcasting, not targeting a peer.
duration_ms	BLE_HS_FOREVER	Advertise indefinitely.
adv_params	defaults	Can be used to specify low level advertising parameters.
cb	NULL	We are non-connectable, so no need for an event callback.
cb_arg	NULL	No callback implies no callback argument.

These arguments are mostly self-explanatory. The exception is adv_params, which can be used to specify a number of low-level parameters. For a beaconing application, the default settings are appropriate. We specify default settings by providing a zero-filled instance of the ble_gap_adv_params struct as our argument.

```
static void
ble_app_advertise(void)
{
    struct ble_gap_adv_params adv_params;
    struct ble_hs_adv_fields fields; int rc;

    /* Configure an eddystone URL beacon to be advertised;
     * URL: https://apache.mynewt.org
     */
    fields = (struct ble_hs_adv_fields){ 0 };
    rc = ble_eddystone_set_adv_data_url(&fields,
                                         BLE_EDDYSTONE_URL_SCHEME_HTTPS,
                                         "mynewt.apache",
                                         13,
                                         BLE_EDDYSTONE_URL_SUFFIX_ORG,
                                         0);
    assert(rc == 0);

    /* Begin advertising. */
    adv_params = (struct ble_gap_adv_params){ 0 };
    rc = ble_gap_adv_start(BLE_OWN_ADDR_RANDOM, NULL, BLE_HS_FOREVER,
                          &adv_params, NULL, NULL);
    assert(rc == 0);
}
```

Conclusion

That's it! Now when you run this app on your board, you should be able to see it with all your eddystone-aware devices. You can test it out with the `newt run` command.

Source Listing

For reference, here is the complete application source:

```
#include "sysinit/sysinit.h"
#include "os/os.h"
#include "console/console.h"
#include "host/ble_hs.h"

static void
ble_app_set_addr(void)
{
    ble_addr_t addr;
    int rc;

    rc = ble_hs_id_gen_rnd(1, &addr);
    assert(rc == 0);

    rc = ble_hs_id_set_rnd(addr.val);
    assert(rc == 0);
}

static void
ble_app_advertise(void)
{
    struct ble_gap_adv_params adv_params;
    struct ble_hs_adv_fields fields;
    int rc;

    /* Configure an eddystone URL beacon to be advertised;
     * URL: https://apache.mynewt.org
     */
    fields = (struct ble_hs_adv_fields){ 0 };
    rc = ble_eddystone_set_adv_data_url(&fields,
                                         BLE_EDDYSTONE_URL_SCHEME_HTTPS,
                                         "mynewt.apache",
                                         13,
                                         BLE_EDDYSTONE_URL_SUFFIX_ORG,
                                         0);
    assert(rc == 0);

    /* Begin advertising. */
    adv_params = (struct ble_gap_adv_params){ 0 };
    rc = ble_gap_adv_start(BLE_OWN_ADDR_RANDOM, NULL, BLE_HS_FOREVER,
                           &adv_params, NULL, NULL);
    assert(rc == 0);
}
```

(continues on next page)

(continued from previous page)

```

static void
ble_app_on_sync(void)
{
    /* Generate a non-resolvable private address. */
    ble_app_set_addr();

    /* Advertise indefinitely. */
    ble_app_advertise();
}

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

    ble_hs_cfg.sync_cb = ble_app_on_sync;

    /* As the last thing, process events from default event queue. */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }
}

```

4.6.4 BLE Peripheral Project

Introduction

- *Overview*
- *Services, Characteristics, Descriptors*

Overview

bleprph is an example app included in the apache-mynewt-nimble repository. This app implements a simple BLE peripheral with the following properties:

- Supports three services: GAP, GATT, and alert notification service (ANS).
- Supports a single concurrent connection.
- Automatically advertises connectability when not connected to a central device.

This tutorial aims to provide a guided tour through the *bleprph* app source code. This document builds on some concepts described elsewhere in the Apache Mynewt documentation. Before proceeding with this tutorial, you might want to familiarize yourself with the following pages:

- *Create Your First Mynewt Project*
- *BLE Bare Bones Application Tutorial*
- *NimBLE Stack Initialization*

Services, Characteristics, Descriptors

A BLE peripheral interfaces with other BLE devices by exposing *services*, *characteristics*, and *descriptors*. All three of these entities are implemented at a lower layer via *attributes*. If you are not familiar with these concepts, you will probably want to check out [overview](#) from the Bluetooth Developer's site before proceeding.

Now let's dig in to some C code.

Service Registration

- *Attribute Set*
- *Registration function*
- *Descriptors and Included Services*

Attribute Set

The NimBLE host uses a table-based design for GATT server configuration. The set of supported attributes are expressed as a series of tables that resides in your C code. When possible, we recommend using a single monolithic table, as it results in code that is simpler and less error prone. Multiple tables can be used if it is impractical for the entire attribute set to live in one place in your code.

bleprph uses a single attribute table located in the *gatt_svr.c* file, so let's take a look at that now. The attribute table is called *gatt_svr_svcs*; here are the first several lines from this table:

```
static const struct ble_gatt_svc_def gatt_svr_svcs[] = {
{
    /*** Service: Security test. */
    .type          = BLE_GATT_SVC_TYPE_PRIMARY,
    .uuid          = &gatt_svr_svc_sec_test_uuid.u,
    .characteristics = (struct ble_gatt_chr_def[]) { {
        /*** Characteristic: Random number generator. */
        .uuid          = &gatt_svr_chr_sec_test_rand_uuid.u,
        .access_cb     = gatt_svr_chr_access_sec_test,
        .flags         = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_READ_ENC,
    }, {
        /*** Characteristic: Static value. */
        .uuid          = gatt_svr_chr_sec_test_static_uuid.u,
        .access_cb     = gatt_svr_chr_access_sec_test,
        .flags         = BLE_GATT_CHR_F_READ |
                          BLE_GATT_CHR_F_WRITE | BLE_GATT_CHR_F_WRITE_ENC,
    }, {
        /**
    // [...]
}
```

As you can see, the table is an array of service definitions (`struct ble_gatt_svc_def`). This code excerpt contains a small part of the *GAP service*. The GAP service exposes generic information about a device, such as its name and appearance. Support for the GAP service is mandatory for all BLE peripherals. Let's now consider the contents of this table in more detail.

A service definition consists of the following fields:

Field	Meaning	Notes
type	Specifies whether this is a primary or secondary service.	Secondary services are not very common. When in doubt, specify <code>BLE_GATT_SVC_TYPE_PRIMARY</code> for new services.
uuid128	The UUID of this service.	If the service has a 16-bit UUID, you can convert it to its corresponding 128-bit UUID with the <code>BLE_UU_ID16()</code> macro.
characteristics	The array of characteristics that belong to this service.	

A service is little more than a container of characteristics; the characteristics themselves are where the real action happens. A characteristic definition consists of the following fields:

Field	Meaning	Notes
uuid1	The UUID of this characteristic.	If the characteristic has a 16-bit UUID, you can convert it to its corresponding 128-bit UUID with the <code>BLE_UU_ID16()</code> macro.
access_cb	A callback function that gets executed whenever a peer device accesses this characteristic.	<i>For reads:</i> this function generates the value that gets sent back to the peer.* <i>For writes:</i> * this function receives the written value as an argument.
flags	Indicates which operations are permitted for this characteristic. The NimBLE stack responds negatively when a peer attempts an unsupported operation.	The full list of flags can be found under <code>ble_gatt_chr_flags</code> in <code>net/nimble/host/include/host/blueGatt.h</code> .

The access callback is what implements the characteristic's behavior. Access callbacks are described in detail in the next section: [BLE Peripheral - Characteristic Access](#).

The service definition array and each characteristic definition array is terminated with an empty entry, represented with a 0. The below code listing shows the last service in the array, including terminating zeros for the characteristic array and service array.

```
{
    /*** Service: Security test. */
    .type = BLE_GATT_SVC_TYPE_PRIMARY,
    .uuid = &gatt_svr_svc_sec_test_uuid.u,
    .characteristics = (struct ble_gatt_chr_def[]) { {
        /*** Characteristic: Random number generator. */
        .uuid = &gatt_svr_chr_sec_test_rand_uuid.u,
        .access_cb = gatt_svr_chr_access_sec_test,
        .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_READ_ENC,
    }, {
        /*** Characteristic: Static value. */
        .uuid = &gatt_svr_chr_sec_test_static_uuid.u,
        .access_cb = gatt_svr_chr_access_sec_test,
        .flags = BLE_GATT_CHR_F_READ |
                  BLE_GATT_CHR_F_WRITE | BLE_GATT_CHR_F_WRITE_ENC,
    }, {
        0, /* No more characteristics in this service. */
    } },
},
```

(continues on next page)

(continued from previous page)

```
0, /* No more services. */  
},
```

Registration function

After you have created your service table, your app needs to register it with the NimBLE stack. This is done by calling the following function:

```
int  
ble_gatts_add_svcs(const struct ble_gatt_svc_def *svcs)
```

where *svcs* is table of services to register.

The `ble_gatts_add_svcs()` function returns 0 on success, or a *BLE_HS_E[...]* error code on failure.

More detailed information about the registration callback function can be found in the [BLE User Guide](#).

The *bleprph* app registers its services as follows:

```
rc = ble_gatts_add_svcs(gatt_svr_svcs);  
assert(rc == 0);
```

which adds services to registration queue. On startup NimBLE host automatically calls `ble_gatts_start()` function which makes all registered services available to peers.

Descriptors and Included Services

Your peripheral can also expose descriptors and included services. These are less common, so they are not covered in this tutorial. For more information, see the [BLE User Guide](#).

Characteristic Access

- *Review*
- *Function signature*
- *Determine characteristic being accessed*
- *Read access*
- *Write access*

Review

A characteristic's access callback implements its behavior. Recall that services and characteristics are registered with NimBLE via attribute tables. Each characteristic definition in an attribute table contains an *access_cb* field. The *access_cb* field is an application callback that gets executed whenever a peer device attempts to read or write the characteristic.

Earlier in this tutorial, we looked at how *bleprph* implements the GAP service. Let's take another look at how *bleprph* specifies the first few characteristics in this service.

```
static const struct ble_gatt_svc_def gatt_svr_svcs[] = {
{
    /*** Service: Security test. */
    .type          = BLE_GATT_SVC_TYPE_PRIMARY,
    .uuid          = &gatt_svr_svc_sec_test_uuid.u,
    .characteristics = (struct ble_gatt_chr_def[]) { {
        /*** Characteristic: Random number generator. */
        .uuid          = &gatt_svr_chr_sec_test_rand_uuid.u,
        .access_cb     = gatt_svr_chr_access_sec_test,
        .flags         = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_READ_ENC,
    }, {
        /*** Characteristic: Static value. */
        .uuid          = gatt_svr_chr_sec_test_static_uuid.u,
        .access_cb     = gatt_svr_chr_access_sec_test,
        .flags         = BLE_GATT_CHR_F_READ |
                          BLE_GATT_CHR_F_WRITE | BLE_GATT_CHR_F_WRITE_ENC,
    }, {
        // [...]
    }
}
```

As you can see, *bleprph* uses the same *access_cb* function for all the GAP service characteristics, but the developer could have implemented separate functions for each characteristic if they preferred. Here is the *access_cb* function that the GAP service characteristics use:

```
static int
gatt_svr_chr_access_sec_test(uint16_t conn_handle, uint16_t attr_handle,
                           struct ble_gatt_access_ctxt *ctxt,
                           void *arg)
{
    const ble_uuid_t *uuid;
    int rand_num;
    int rc;

    uuid = ctxt->chr->uuid;

    /* Determine which characteristic is being accessed by examining its
     * 128-bit UUID.
     */

    if (ble_uuid_cmp(uuid, &gatt_svr_chr_sec_test_rand_uuid.u) == 0) {
        assert(ctxt->op == BLE_GATT_ACCESS_OP_READ_CHR);

        /* Respond with a 32-bit random number. */
        rand_num = rand();
        rc = os_mbuf_append(ctxt->om, &rand_num, sizeof rand_num);
    }
}
```

(continues on next page)

(continued from previous page)

```

    return rc == 0 ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;
}

if (ble_uuid_cmp(uuid, &gatt_svr_chr_sec_test_static_uuid.u) == 0) {
    switch (ctxt->op) {
        case BLE_GATT_ACCESS_OP_READ_CHR:
            rc = os_mbuf_append(ctxt->om, &gatt_svr_sec_test_static_val,
                                sizeof gatt_svr_sec_test_static_val);
            return rc == 0 ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;

        case BLE_GATT_ACCESS_OP_WRITE_CHR:
            rc = gatt_svr_chr_write(ctxt->om,
                                    sizeof gatt_svr_sec_test_static_val,
                                    sizeof gatt_svr_sec_test_static_val,
                                    &gatt_svr_sec_test_static_val, NULL);
            return rc;

        default:
            assert(0);
            return BLE_ATT_ERR_UNLIKELY;
    }
}

/* Unknown characteristic; the nimble stack should not have called this
 * function.
 */
assert(0);
return BLE_ATT_ERR_UNLIKELY;
}

```

After you've taken a moment to examine the structure of this function, let's explore some details.

Function signature

```

static int
gatt_svr_chr_access_sec_test(uint16_t conn_handle, uint16_t attr_handle,
                           struct ble_gatt_access_ctxt *ctxt, void *arg)

```

A characteristic access function always takes this same set of parameters and always returns an int. The parameters to this function type are documented below.

Parameter	Purpose	Notes
conn_h	Indicates which connection the characteristic access was sent over.	Use this value to determine which peer is accessing the characteristic.
attr_han	The low-level ATT handle of the characteristic value attribute.	Can be used to determine which characteristic is being accessed if you don't want to perform a UUID lookup.
ctxt	Contains the characteristic value pointer that the application needs to access.	For characteristic accesses, use the <i>ctxt->chr_access</i> member; for descriptor accesses, use the <i>ctxt->dsc_access</i> member.

The return value of the access function tells the NimBLE stack how to respond to the peer performing the operation. A value of 0 indicates success. For failures, the function returns the specific ATT error code that the NimBLE stack should respond with. The ATT error codes are defined in [net/nimble/host/include/host/ble_att.h](#).

Determine characteristic being accessed

```
ble_uuid_cmp(uuid, &gatt_svr_chr_sec_test_rand_uuid.u)
```

The function compares UUID with UUIDs of characteristic - if it fits, characteristic is being accessed. There are two alternative methods *bleprph* could have used to accomplish this task:

- Map characteristics to ATT handles during service registration; use the *attr_handle* parameter as a key into this table during characteristic access.
- Implement a dedicated function for each characteristic; each function inherently knows which characteristic it corresponds to.

Read access

```
case BLE_GATT_ACCESS_OP_READ_CHR:
    rc = os_mbuf_append(ctxt->om, &gatt_svr_sec_test_static_val,
                        sizeof gatt_svr_sec_test_static_val);
    return rc == 0 ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;
```

This code excerpt handles read accesses to the device characteristic. *ctxt->om* is chained memory buffer that for reads is being populated with characteristic data. Returned value is either 0 for success or *BLE_ATT_ERR_INSUFFICIENT_RES* if failed. The check makes sure the NimBLE stack is doing its job; this characteristic was registered as read-only, so the stack should have prevented write accesses.

Write access

```
static int
gatt_svr_chr_write(struct os_mbuf *om, uint16_t min_len, uint16_t max_len,
                    void *dst, uint16_t *len)
{
    uint16_t om_len;
    int rc;
```

(continues on next page)

(continued from previous page)

```

om_len = OS_MBUF_PKTLEN(om);
if (om_len < min_len || om_len > max_len) {
    return BLE_ATT_ERR_INVALID_ATTR_VALUE_LEN;
}

rc = ble_hs_mbuf_to_flat(om, dst, max_len, len);
if (rc != 0) {
    return BLE_ATT_ERR_UNLIKELY;
}

return 0;
}
// [...]
case BLE_GATT_ACCESS_OP_WRITE_CHR:
    rc = gatt_svr_chr_write(ctxt->om,
                           sizeof(gatt_svr_sec_test_static_val),
                           sizeof(gatt_svr_sec_test_static_val),
                           &gatt_svr_sec_test_static_val, NULL);
    return rc;

```

This code excerpt handles writes to the Static value characteristic. This characteristic was registered as read-write, so the *return rc* here is just a safety precaution to ensure the NimBLE stack is doing its job.

Data is written to the *ctxt->om* buffer from *gatt_svr_sec_test_static_val* by *ble_hs_mbuf_to_flat()* function. If length of written data greater or smaller than length of *gatt_svr_sec_test_static_val*, function return error.

Many characteristics have strict length requirements for write operations.

Advertising

- *Overview*
- *Setting advertisement data*
- *Begin advertising*

Overview

A peripheral announces its presence to the world by broadcasting advertisements. An advertisement typically contains additional information about the peripheral sending it, such as the device name and an abbreviated list of supported services. The presence of this information helps a listening central to determine whether it is interested in connecting to the peripheral. Advertisements are quite limited in the amount of information they can contain, so only the most important information should be included.

When a listening device receives an advertisement, it can choose to connect to the peripheral, or query the sender for more information. This second action is known as an *active scan*. A peripheral responds to an active scan with some extra information that it couldn't fit in its advertisement. This additional information is known as *scan response data*. *bleprph* does not configure any scan response data, so this feature is not discussed in the remainder of this tutorial.

bleprph constantly broadcasts advertisements until a central connects to it. When a connection is terminated, *bleprph* resumes advertising.

Let's take a look at *bleprph*'s advertisement code (*main.c*):

```
/***
 * Enables advertising with the following parameters:
 *   o General discoverable mode.
 *   o Undirected connectable mode.
 */
static void
bleprph_advertise(void)
{
    struct ble_hs_adv_fields fields;
    int rc;

    /* Set the advertisement data included in our advertisements. */
    memset(&fields, 0, sizeof fields);
    fields.name = (uint8_t *)bleprph_device_name;
    fields.name_len = strlen(bleprph_device_name);
    fields.name_is_complete = 1;
    rc = ble_gap_adv_set_fields(&fields);
    if (rc != 0) {
        BLEPRPH_LOG(ERROR, "error setting advertisement data; rc=%d\n", rc);
        return;
    }

    /* Begin advertising. */
    rc = ble_gap_adv_start(BLE_GAP_DISC_MODE_GEN, BLE_GAP_CONN_MODE_UND,
                           NULL, 0, NULL, bleprph_on_connect, NULL);
    if (rc != 0) {
        BLEPRPH_LOG(ERROR, "error enabling advertisement; rc=%d\n", rc);
        return;
    }
}
```

Now let's examine this code in detail.

Setting advertisement data

A NimBLE peripheral specifies what information to include in its advertisements with the following function:

```
int
ble_gap_adv_set_fields(struct ble_hs_adv_fields *adv_fields)
```

The *adv_fields* argument specifies the fields and their contents to include in subsequent advertisements. The Bluetooth Core Specification Supplement defines a set of standard fields that can be included in an advertisement; the member variables of the *struct ble_hs_adv_fields* type correspond to these standard fields. Information that doesn't fit neatly into a standard field should be put in the *manufacturing specific data* field.

As you can see in the above code listing, the *struct ble_hs_adv_fields* instance is allocated on the stack. It is OK to use the stack for this struct and the data it references, as the *ble_gap_adv_set_fields()* function makes a copy of all the advertisement data before it returns. *bleprph* doesn't take full advantage of this; it stores its device name in a static array.

The code sets three members of the *struct ble_hs_adv_fields* instance:

- name
- name_len
- name_is_complete

The first two fields are used to communicate the device's name and are quite straight-forward. The third field requires some explanation. Bluetooth specifies two name-related advertisement fields: *Shortened Local Name* and *Complete Local Name*. Setting the `name_is_complete` variable to 1 or 0 tells NimBLE which of these two fields to include in advertisements. Some other advertisement fields also correspond to multiple variables in the field struct, so it is a good idea to review the *ble_hs_adv_fields* reference to make sure you get the details right in your app.

Begin advertising

An app starts advertising with the following function:

```
int  
ble_gap_adv_start(uint8_t discoverable_mode, uint8_t connectable_mode,  
                  uint8_t *peer_addr, uint8_t peer_addr_type,  
                  struct hci_adv_params *adv_params,  
                  ble_gap_conn_fn *cb, void *cb_arg)
```

This function allows a lot of flexibility, and it might seem daunting at first glance. `bleprph` specifies a simple set of arguments that is appropriate for most peripherals. When getting started on a typical peripheral, we recommend you use the same arguments as `bleprph`, with the exception of the last two (`cb` and `cb_arg`). These last two arguments will be specific to your app, so let's talk about them.

`cb` is a callback function. It gets executed when a central connects to your peripheral after receiving an advertisement. The `cb_arg` argument gets passed to the `cb` callback. If your callback doesn't need the `cb_arg` parameter, you can do what `bleprph` does and pass `NULL`. Once a connection is established, the `cb` callback becomes permanently associated with the connection. All subsequent events related to the connection are communicated to your app via calls to this callback function. Connection callbacks are an important part of building a BLE app, and we examine `bleprph`'s connection callback in detail in the next section of this tutorial.

One final note: Your peripheral automatically stops advertising when a central connects to it. You can immediately resume advertising if you want to allow another central to connect, but you will need to do so explicitly by calling `ble_gap_adv_start()` again. Also, be aware NimBLE's default configuration only allows a single connection at a time. NimBLE supports multiple concurrent connections, but you must configure it to do so first.

GAP Event callbacks

- *Overview*
- *bleprph_gap_event()*
- *Guarantees*

Overview

Every BLE connection has a *GAP event callback* associated with it. A GAP event callback is a bit of application code which NimBLE uses to inform you of connection-related events. For example, if a connection is terminated, NimBLE lets you know about it with a call to that connection's callback.

In the *advertising section* of this tutorial, we saw how the application specifies a GAP event callback when it begins advertising. NimBLE uses this callback to notify the application that a central has connected to your peripheral after receiving an advertisement. Let's revisit how *bleprph* specifies its connection callback when advertising:

```
/* Begin advertising. */
memset(&adv_params, 0, sizeof adv_params);
adv_params.conn_mode = BLE_GAP_CONN_MODE_UND;
adv_params.disc_mode = BLE_GAP_DISC_MODE_GEN;
rc = ble_gap_adv_start(BLE_ADDR_TYPE_PUBLIC, 0, NULL, BLE_HS_FOREVER,
                      &adv_params, bleprph_gap_event, NULL);
if (rc != 0) {
    BLEPRPH_LOG(ERROR, "error enabling advertisement; rc=%d\n", rc);
    return;
}
```

bleprph_gap_event()

The *bleprph_gap_event()* function is *bleprph*'s GAP event callback; NimBLE calls this function when the advertising operation leads to connection establishment. Upon connection establishment, this callback becomes permanently associated with the connection; all subsequent events related to this connection are communicated through this callback.

Now let's look at the function that *bleprph* uses for all its connection callbacks: *bleprph_gap_event()*.

```
/**
 * The nimble host executes this callback when a GAP event occurs. The
 * application associates a GAP event callback with each connection that forms.
 * bleprph uses the same callback for all connections.
 *
 * @param event           The type of event being signalled.
 * @param ctxt            Various information pertaining to the event.
 * @param arg             Application-specified argument; unused by
 *                       bleprph.
 *
 * @return                0 if the application successfully handled the
 *                       event; nonzero on failure. The semantics
 *                       of the return code is specific to the
 *                       particular GAP event being signalled.
 */
static int
bleprph_gap_event(struct ble_gap_event *event, void *arg)
{
    struct ble_gap_conn_desc desc;
    int rc;

    switch (event->type) {
    case BLE_GAP_EVENT_CONNECT:
        /* A new connection was established or a connection attempt failed. */
    }
```

(continues on next page)

(continued from previous page)

```

BLEPRPH_LOG(INFO, "connection %s; status=%d ",
            event->connect.status == 0 ? "established" : "failed",
            event->connect.status);
if (event->connect.status == 0) {
    rc = ble_gap_conn_find(event->connect.conn_handle, &desc);
    assert(rc == 0);
    bleprph_print_conn_desc(&desc);
}
BLEPRPH_LOG(INFO, "\n");

if (event->connect.status != 0) {
    /* Connection failed; resume advertising. */
    bleprph_advertise();
}
return 0;

case BLE_GAP_EVENT_DISCONNECT:
    BLEPRPH_LOG(INFO, "disconnect; reason=%d ", event->disconnect.reason);
    bleprph_print_conn_desc(&event->disconnect.conn);
    BLEPRPH_LOG(INFO, "\n");

    /* Connection terminated; resume advertising. */
    bleprph_advertise();
    return 0;

case BLE_GAP_EVENT_CONN_UPDATE:
    /* The central has updated the connection parameters. */
    BLEPRPH_LOG(INFO, "connection updated; status=%d ",
                event->conn_update.status);
    rc = ble_gap_conn_find(event->connect.conn_handle, &desc);
    assert(rc == 0);
    bleprph_print_conn_desc(&desc);
    BLEPRPH_LOG(INFO, "\n");
    return 0;

case BLE_GAP_EVENT_ENC_CHANGE:
    /* Encryption has been enabled or disabled for this connection. */
    BLEPRPH_LOG(INFO, "encryption change event; status=%d ",
                event->enc_change.status);
    rc = ble_gap_conn_find(event->connect.conn_handle, &desc);
    assert(rc == 0);
    bleprph_print_conn_desc(&desc);
    BLEPRPH_LOG(INFO, "\n");
    return 0;

case BLE_GAP_EVENT_SUBSCRIBE:
    BLEPRPH_LOG(INFO, "subscribe event; conn_handle=%d attr_handle=%d "
                "reason=%d prevn=%d curn=%d previ=%d curi=%d\n",
                event->subscribe.conn_handle,
                event->subscribe.attr_handle,
                event->subscribe.reason,
                event->subscribe.prev_notify,

```

(continues on next page)

(continued from previous page)

```

        event->subscribe.cur_notify,
        event->subscribe.prev_indicate,
        event->subscribe.cur_indicate);
    return 0;
}

return 0;
}

```

Connection callbacks are used to communicate a variety of events related to a connection. An application determines the type of event that occurred by inspecting the value of the *event->type* parameter. The full list of event codes can be found on the [GAP page](#).

Guarantees

It is important to know what your application code is allowed to do from within a connection callback.

No restrictions on NimBLE operations

Your app is free to make calls into the NimBLE stack from within a connection callback. *bleprph* takes advantage of this freedom when it resumes advertising upon connection termination. All other NimBLE operations are also allowed (service discovery, pairing initiation, etc).

All context data is transient

Pointers in the context object point to data living on the stack. Your callback is free to read (or write, if appropriate) through these pointers, but you should not store these pointers for later use. If your application needs to retain some data from a context object, it needs to make a copy.

BLE Peripheral App

Overview

Now that we've gone through how BLE Apps are contructed, how they function, and how all the parts fit together let's try out a BLE Peripheral App to see how it all works.

Prerequisites

- You should have a BLE Central App of some sort to connect with. On Mac OS or iOS, you can use [LightBlue](#) which is a free app to browse and connect to BLE Peripheral devices.

Create a New Target

You can create a new project instead, but this tutorial will simply use the previously created btshell project and add a new target for the BLE Peripheral

```

$ newt target create myperiph
Target targets/myperiph successfully created
$ newt target set myperiph bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
Target targets/myperiph successfully set target.bsp to @apache-mynewt-core/hw/bsp/nordic_

```

(continues on next page)

(continued from previous page)

```

→pca10040
$ newt target set myperiph app=@apache-mynewt-nimble/apps/bleprph
Target targets/myperiph successfully set target.app to @apache-mynewt-nimble/apps/bleprph
$ newt target set myperiph build_profile=optimized
Target targets/myperiph successfully set target.build_profile to optimized
$ newt build myperiph
Building target targets/myperiph
...
Linking ~/dev/nrf52dk/bin/targets/myperiph/app/apps/bleprph/bleprph.elf
Target successfully built: targets/myperiph
$ newt create-image myperiph 1.0.0
App image successfully generated: ~/dev/nrf52dk/bin/targets/myperiph/app/apps/bleprph/
→bleprph.img
$ newt load myperiph
Loading app image into slot 1

```

Now if you reset the board, and fire up your BLE Central App, you should see a new peripheral device called ‘nimble-bleprph’.

Now that you can see the device, you can begin to interact with the advertised service.

Click on the device and you’ll establish a connection.

Now that you’re connected, you can see the Services that are being advertised.

Scroll to the bottom and you will see a Read Characteristic, and a Read/Write Characteristic.

Just click on the Read Write Characteristic and you will see the existing value.

Type in a new value.

And you will see the new value reflected.

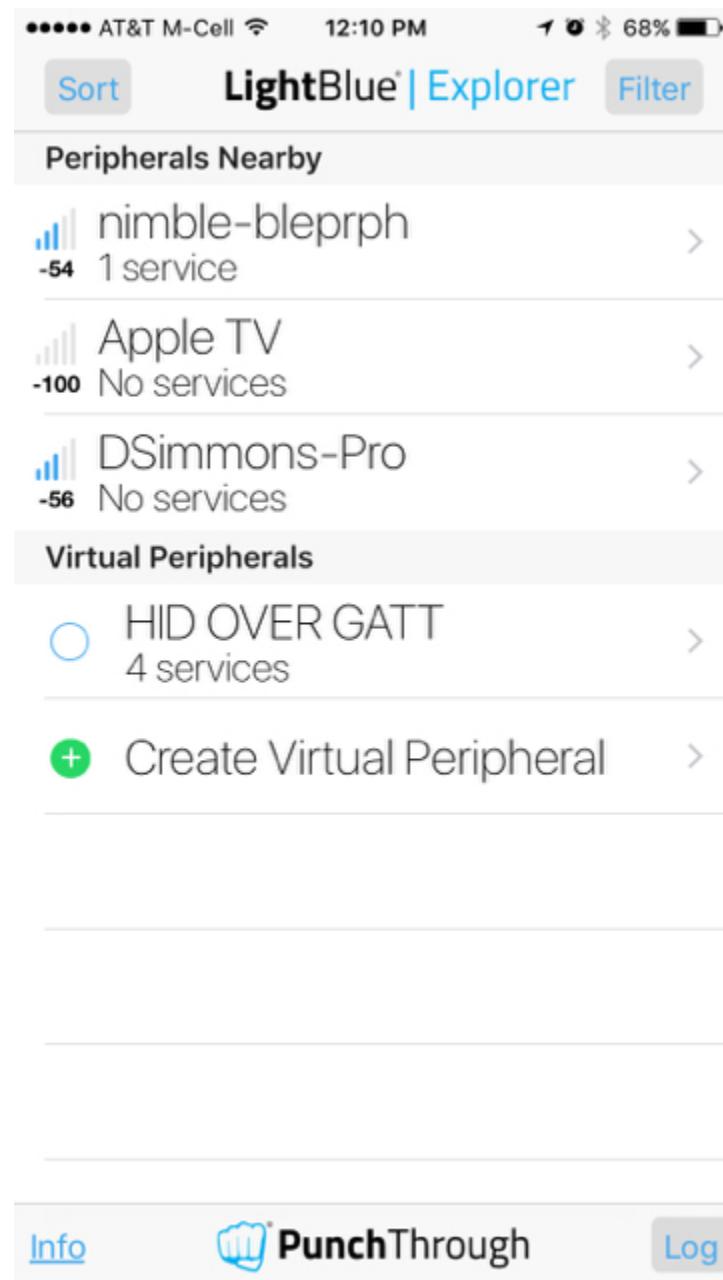
If you still have your console connected, you will be able to see the connection requests, and pairing, happen on the device as well.

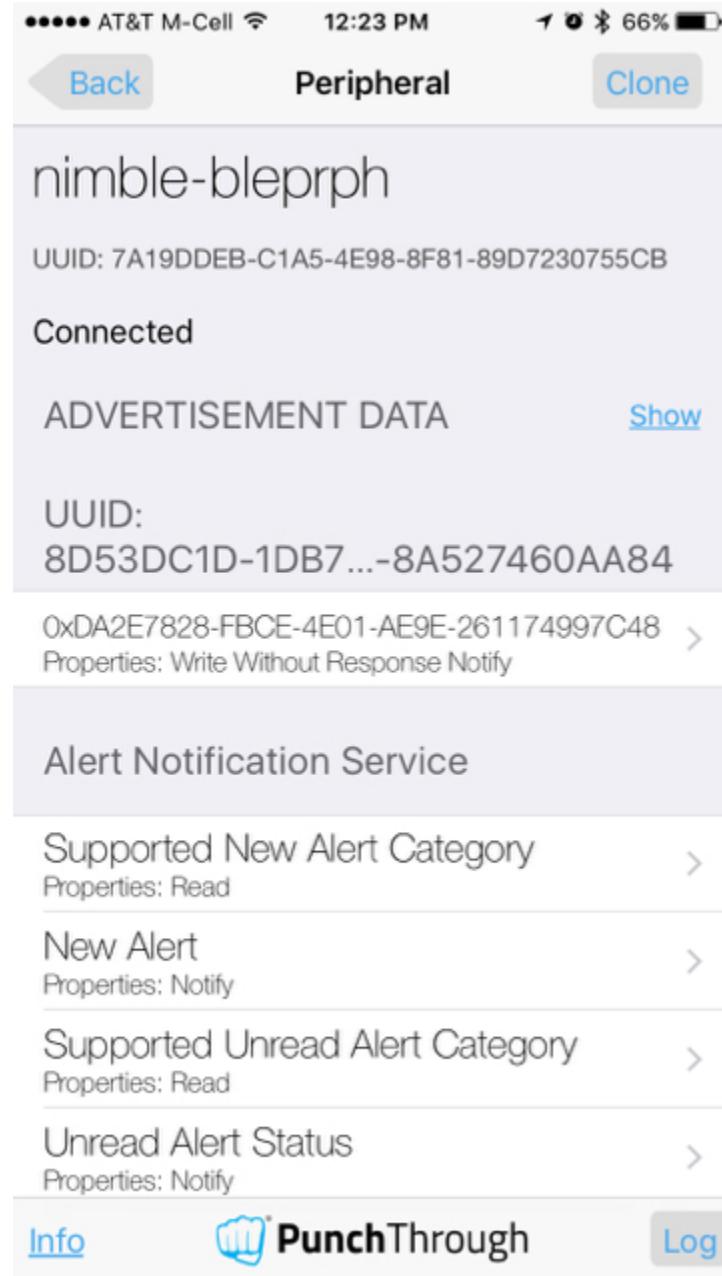
```

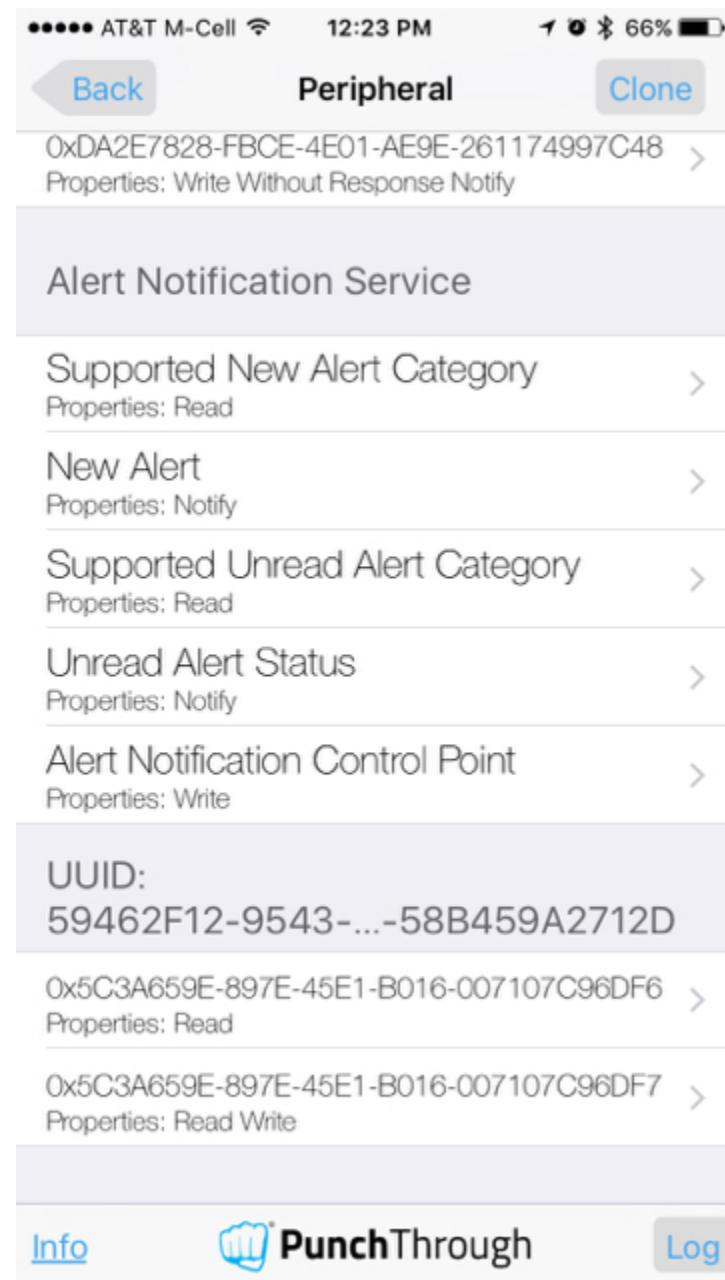
258894:[ts=2022609336ssb, mod=64 level=1] connection established; status=0 handle=1 our_
→ota_addr_type=0 our_ota_addr=0a:0a:0a:0a:0a:0a our_id_addr_type=0 our_id_
→addr=0a:0a:0a:0a:0a:0a peer_ota_addr_type=1 peer_ota_addr=7f:be:d4:44:c0:d4 peer_id_
→addr_type=1 peer_id_addr=7f:be:d4:44:c0:d4 conn_itvl=24 conn_latency=0 supervision_
→timeout=72 encrypted=0 authenticated=0 bonded=0
258904:[ts=2022687456ssb, mod=64 level=1]
258917:[ts=2022789012ssb, mod=64 level=1] mtu update event; conn_handle=1 cid=4 mtu=185
258925:[ts=2022851508ssb, mod=64 level=1] subscribe event; conn_handle=1 attr_handle=14_
→reason=1 prevn=0 currn=0 previ=0 curi=1
261486:[ts=2042859320ssb, mod=64 level=1] encryption change event; status=0 handle=1 our_
→ota_addr_type=0 our_ota_addr=0a:0a:0a:0a:0a:0a our_id_addr_type=0 our_id_
→addr=0a:0a:0a:0a:0a:0a peer_ota_addr_type=1 peer_ota_addr=7f:be:d4:44:c0:d4 peer_id_
→addr_type=1 peer_id_addr=7f:be:d4:44:c0:d4 conn_itvl=24 conn_latency=0 supervision_
→timeout=72 encrypted=1 authenticated=0 bonded=1
261496:[ts=2042937440ssb, mod=64 level=1]

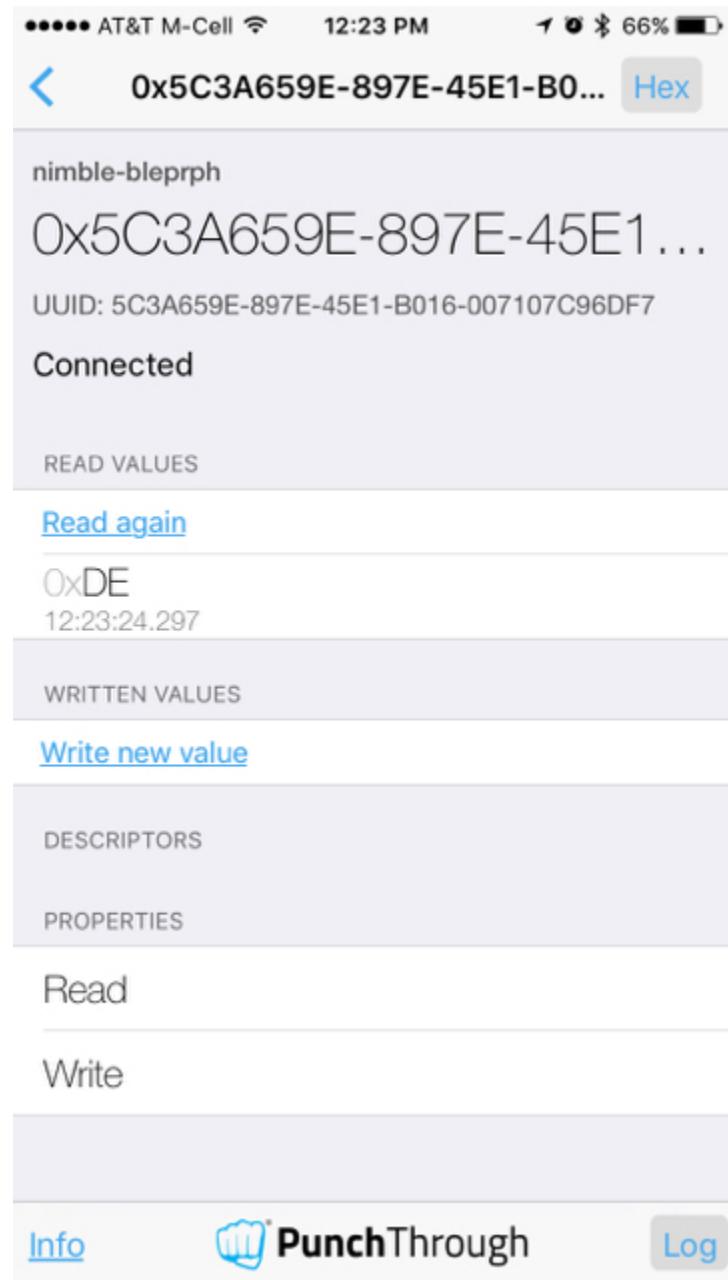
```

Congratulations! You’ve just built and connected your first BLE Peripheral device!









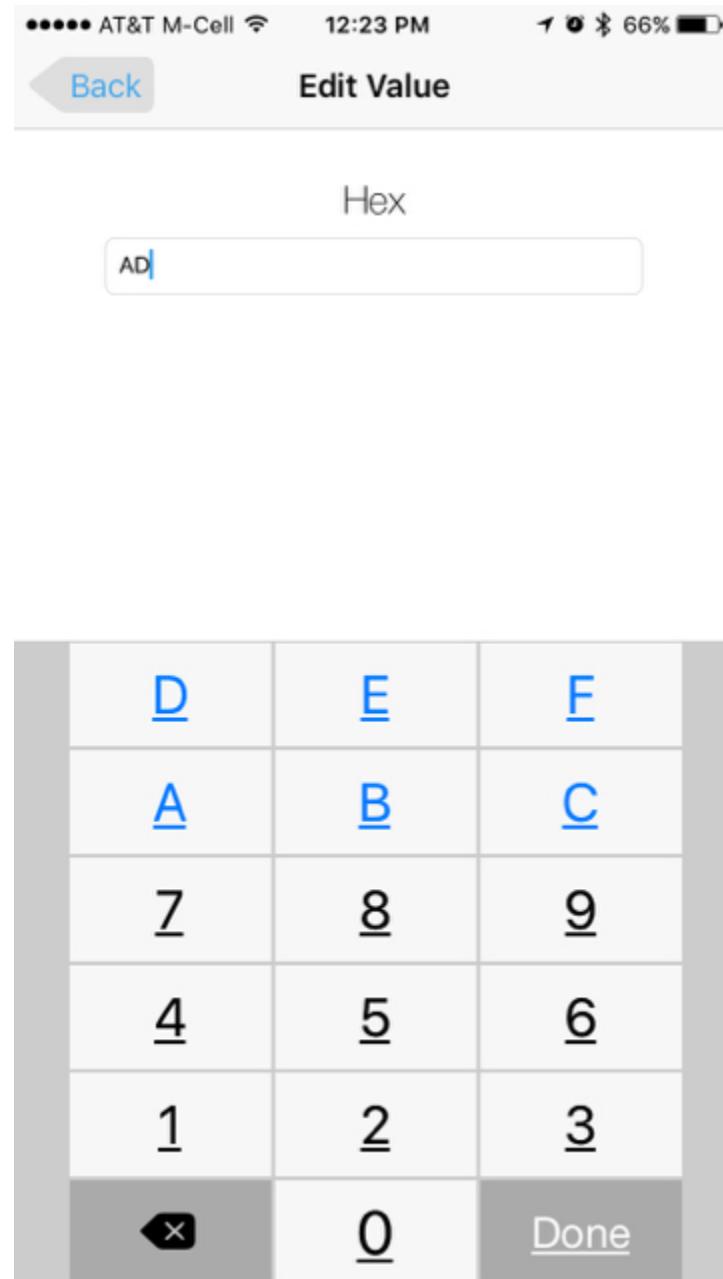
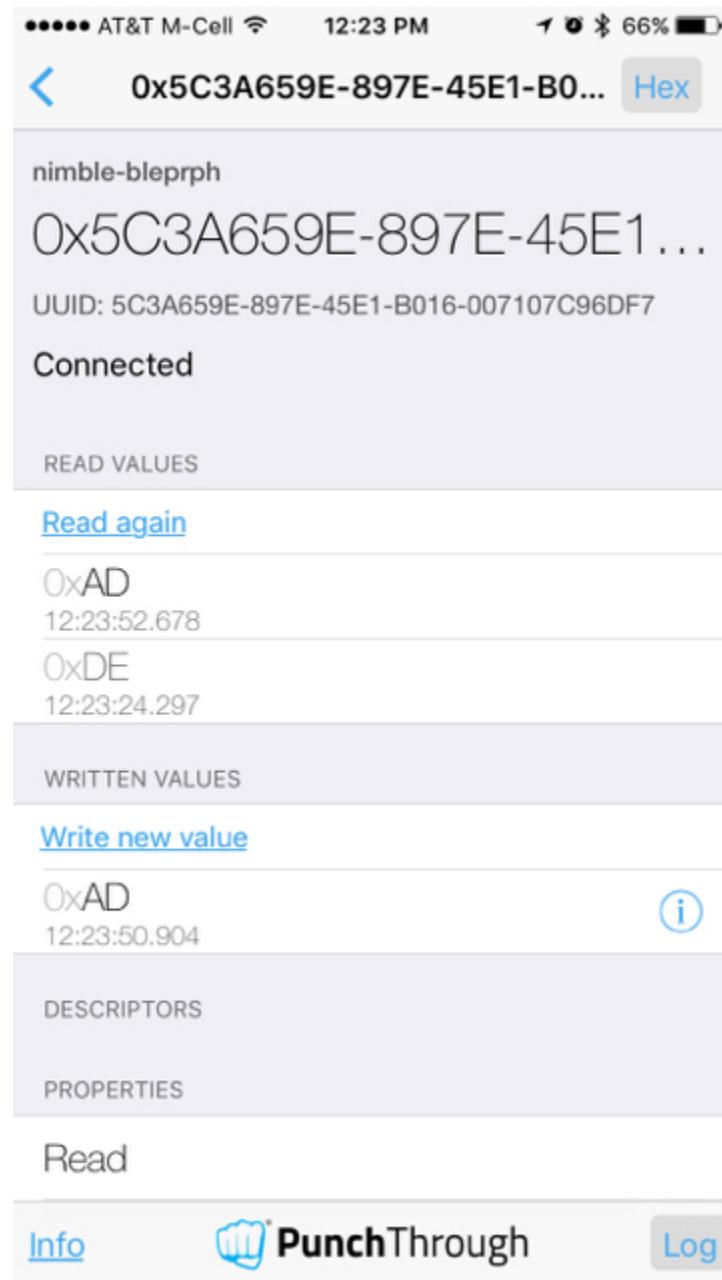


Fig. 3: LightBlue



4.6.5 Use HCI access to NimBLE controller

This tutorial explains how to use the example application `blehci` included in the NimBLE stack to talk to the Mynewt NimBLE controller via the Host Controller Interface. You may build the Mynewt image using a laptop running any OS of your choice - Mac, Linux, or Windows.

The host used in this specific example is the BlueZ Bluetooth stack. Since BlueZ is a Bluetooth stack for Linux kernel-based family of operating system, the tutorial expects a computer running Linux OS and with BlueZ installed to talk to the board with the Mynewt image.

- *Prerequisites*
- *Create a project*
- *Create targets*
- *Build targets*
- *Create the app image*
- *Load the bootloader and the application image*
- *Establish serial connection*
- *Open Bluetooth monitor btmon*
- *Attach the blehci device to BlueZ*
- *Start btmgmt to send commands*

Prerequisites

Ensure that you meet the following prerequisites before continuing with one of the tutorials.

- Have Internet connectivity to fetch remote Mynewt components.
- Have a board with BLE radio that is supported by Mynewt. We will use an nRF52 Dev board in this tutorial.
- Have a USB TTL Serial Cable that supports hardware flow control such as ones found at <http://www.ftdichip.com/Products/Cables/USBTTLSerial.htm> to establish a serial USB connection between the board and the laptop.
- Install the newt tool and toolchains (See [Basic Setup](#)).
- Install a BLE host such as BlueZ on a Linux machine to talk to the nRF52 board running Mynewt. Use `sudo apt-get install bluez` to install it on your Linux machine.

Create a project

Use the newt tool to create a new project directory containing a skeletal Mynewt framework. Change into the newly created directory.

```
$ newt new blehciproj
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in blehciproj ...
Project blehciproj successfully created.
$ cd mblehciproj
```

(continues on next page)

(continued from previous page)

```
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
apache-mynewt-core successfully upgraded to version 1.7.0
```

Create targets

You will create two targets - one for the bootloader, the other for the application. Then you will add the definitions for them. Note that you are using the example app blehci for the application target. Set the bsp to nordic_pca10040.

NOTE: The preview version, nRF52PDK, is no longer supported. If you do not see PCA100040 on the top of your board, you have a preview version of the board and will need to upgrade your developer board before continuing.

```
$ newt target create nrf52_boot
$ newt target set nrf52_boot app=@mcuboot/boot/mynewt
$ newt target set nrf52_boot bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
$ newt target set nrf52_boot build_profile=optimized
```

```
$ newt target create myble2
$ newt target set myble2 bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
$ newt target set myble2 app=@apache-mynewt-nimble/apps/blehci
$ newt target set myble2 build_profile=optimized
```

Check that the targets are defined correctly.

```
$ newt target show
targets/my_blinky_sim
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/native
    build_profile=debug
targets/myble2
    app=@apache-mynewt-nimble/apps/blehci
    bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
    build_profile=optimized
targets/nrf52_boot
    app=@mcuboot/boot/mynewt
    bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
    build_profile=optimized
```

Build targets

Then build the two targets.

```
$ newt build nrf52_boot
<snip>
Linking ~/dev/blehciproj/bin/targets/nrf52_boot/app/boot/mynewt/mynewt.elf
Target successfully built: targets/nrf52_boot

$ newt build myble2
<snip>
Linking ~/dev/blehciproj/bin/targets/myble2/app/apps/blehci/blehci.elf
```

(continues on next page)

(continued from previous page)

```
Target successfully built: targets/myble2
$
```

Create the app image

Generate a signed application image for the myble2 target. The version number is arbitrary.

```
$ newt create-image myble2 1.0.0
App image successfully generated: ~/dev/blehciproj/bin/targets/myble2/app/apps/blehci/
→blehci.img
```

Load the bootloader and the application image

Make sure the USB connector is in place and the power LED on the board is lit. Use the Power ON/OFF switch to reset the board after loading the image.

Load the bootloader:

```
$ newt load nrf52_boot
Loading bootloader
$
```

Load the application image:

```
$ newt load myble2
Loading app image into slot 1
$
```

Establish serial connection

Attach a serial port to your board by connecting the USB TTL Serial Cable. This should create /dev/ttyUSB0 (or similar) on your machine.

Note Certain Linux OS versions have been observed to detect the nrf52 board as a mass storage device and the console access doesn't work properly. In that case try powering the nrf52 board from your monitor or something other than your Linux computer/laptop when you set up the serial port for HCI communication.

Open Bluetooth monitor btmon

btmon is a BlueZ test tool to display all HCI commands and events in a human readable format. Start the btmon tool in a terminal window.

```
$ sudo btmon
[sudo] password for admin:
Bluetooth monitor ver 5.37
```

Attach the blehci device to BlueZ

In a different terminal, attach the blehci device to the BlueZ daemon (substitute the correct /dev filename for ttyUSB0).

```
$ sudo btattach -B /dev/ttyUSB0 -S 1000000
Attaching BR/EDR controller to /dev/ttyUSB0
Switched line discipline from 0 to 15
Device index 1 attached
```

The baud rate used to connect to the controller may be changed by overriding the default value of 1000000 in the net/nimble/transport/uart/syscfg.yml. Settings in the serial transport syscfg.yml file can be overridden by a higher priority package such as the application. So, for example, you may set the BLE_HCI_UART_BAUD to a different value in apps/blehci/syscfg.yml.

If there is no CTS/RTS lines present in the test environment, flow control should be turned off. This can be done with -N option for btattach. **Note:** -N option came with BlueZ ver 5.44. Also, modify the value of BLE_HCI_UART_FLOW_CTRL in the nimble/transport/uart/syscfg.yml to HAL_UART_FLOW_CTL_NONE.

Start btmgmt to send commands

In a third terminal, start btmgmt. This tool allows you to send commands to the blehci controller. Use the index number that shows up when you btattach in the previous step.

```
$ sudo btmgmt --index 1
[sudo] password for admin:
```

Set your device address (you can substitute any static random address here).

```
[hci1]# static-addr cc:11:11:11:11:11
Static address successfully set
```

Initialize the controller.

```
[hci1]# power on
hci1 Set Powered complete, settings: powered le static-addr
```

Begin scanning.

```
[hci1]# find -l
Discovery started
hci1 type 6 discovering on
hci1 dev_found: 58:EF:77:C8:8D:17 type LE Random rssi -78 flags 0x0000
AD flags 0x06
eir_len 23
<snip>
```

4.7 LoRaWAN App

4.7.1 Objective

The purpose of this tutorial is to demonstrate how to build the lora app shell application for either a class A or class C lora device and to perform basic functions such as joining and sending data packets to a lora gateway/server.

NOTE: This tutorial presumes that you have a running lora gateway and lora network server. No description of the gateway/server is provided. It is expected that the user understands how to configure and operate the gateway/server so that it can communicate with a class A or class C device.

4.7.2 Hardware needed

- Telenor EE02 module
- Segger J-Link or similar debugger
- LORA gateway
- Laptop running Mac OS
- It is assumed you have already installed newt tool.
- It is assumed you understand the basics of the mynewt OS
- 3-wire serial cable to connect telenor module to your laptop
- Some form of terminal emulation application running on your laptop.

4.7.3 Create a project.

Create a new project to hold your work. For a deeper understanding, you can read about project creation in [Get Started – Creating Your First Project](#) or just follow the commands below.

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new mylora
Downloading project skeleton from apache/mynewt-blinky...
Installing skeleton in mylora...
Project mylora successfully created.
$ cd mylora
```

4.7.4 Install Everything

Now that you have defined the needed repositories, it's time to install everything so that you can get started.

```
$ newt upgrade -v
Downloading repository mynewt-core (commit: master) ...
...
apache-mynewt-core successfully upgraded to version 1.7.0
...
```

4.7.5 Create the targets

Create two targets - one for the bootloader and one for the lora app shell application.

```
$ newt target create telee02_boot
$ newt target set telee02_boot bsp=@apache-mynewt-core/hw/bsp/telee02
$ newt target set telee02_boot app=@mcuboot/boot/mynewt
$ newt target set telee02_boot build_profile=optimized

$ newt target create lora_app_shell_telee02
$ newt target set lora_app_shell_telee02 bsp=@apache-mynewt-core/hw/bsp/telee02
$ newt target set lora_app_shell_telee02 app=@apache-mynewt-core/apps/lora_app_shell
$ newt target set lora_app_shell_telee02 build_profile=optimized
```

The lora app shell application requires a few additional system configuration variables. Create and edit a file called syscfg.yml in dev/mylora/targets/lora_app_shell. The file contents should be the following:

```
### Package: targets/lora_app_shell_telee02

syscfg.vals:
  SHELL_CMD_ARGC_MAX: "20"
  LORA_MAC_TIMER_NUM: "4"
  TIMER_4: "1"
```

You can now “display” the targets you created to make sure they are correct:

```
$ newt target show telee02_boot
targets/telee02_boot
  app=@mcuboot/boot/mynewt
  bsp=@apache-mynewt-core/hw/bsp/telee02
  build_profile=optimized

$ newt target show lora_app_shell_telee02
targets/lora_app_shell_telee02
  app=@apache-mynewt-core/apps/lora_app_shell
  bsp=@apache-mynewt-core/hw/bsp/telee02
  build_profile=optimized
  syscfg=LORA_MAC_TIMER_NUM=4:SHELL_CMD_ARGC_MAX=20:TIMER_4=1
```

Note: If you’ve already built and installed a bootloader for your ee02 module then you do not need to create a target for it here, or build and load it as below.

4.7.6 Build the target executables

```
$ newt clean telee02_boot
$ newt build telee02_boot
Building target targets/telee02_boot
Compiling repos/apache-mynewt-core/boot/bootutil/src/image_rsa.c
Compiling repos/apache-mynewt-core/boot/bootutil/src/image_ec.c
Compiling repos/apache-mynewt-core/boot/bootutil/src/image_ec256.c

. . .
```

(continues on next page)

(continued from previous page)

```

Archiving telee02_boot-sysinit-app.a
Archiving util_mem.a
Linking /Users/wes/dev/wes/bin/targets/telee02_boot/app/boot/mynewt/mynewt.elf
Target successfully built: targets/telee02_boot

$ newt clean lora_app_shell_telee02
$ newt build lora_app_shell_telee02
Building target targets/lora_app_shell_telee02
Assembling repos/apache-mynewt-core/hw/bsp/telee02/src/arch/cortex_m4/gcc_startup_nrf52_
↳split.s
Compiling repos/apache-mynewt-core/encoding/base64/src/hex.c
Compiling repos/apache-mynewt-core/encoding/base64/src/base64.c
. . .

Archiving util_mem.a
Archiving util_parse.a
Linking /Users/wes/dev/wes/bin/targets/lora_app_shell_telee02/app/apps/lora_app_shell/
↳lora_app_shell.elf
Target successfully built: targets/lora_app_shell_telee0

```

Note: The newt clean step is not necessary but shown here for good measure.

4.7.7 Sign and create the application image

You must sign and version your application image to download it using newt to the board. Use the newt create-image command to perform this action. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image lora_app_shell_telee02 0.0.0
App image successfully generated: /Users/wes/dev/wes/bin/targets/lora_app_shell_telee02/
↳app/apps/lora_app_shell/lora_app_shell.img
```

Note: Only the application image requires this step; the bootloader does not

4.7.8 Connect the board

Connect the evaluation board via micro-USB to your PC via USB cable. Connect the Segger J-link debugger to the 9-pin SWD connector. Connect the UART pins (RX, TX and GND) to the board. Terminal settings 115200, N, 8, 1.

4.7.9 Download bootloader and application

Note: If you want to erase the flash and load the image again, you can use JLinkExe to issue an `erase` command.

```
$ JLinkExe -device nRF52 -speed 4000 -if SWD
SEGGER J-Link Commander V5.12c (Compiled Apr 21 2016 16:05:51)
DLL version V5.12c, compiled Apr 21 2016 16:05:45

Connecting to J-Link via USB...O.K.
Firmware: J-Link OB-SAM3U128-V2-NordicSemi compiled Mar 15 2016 18:03:17
Hardware version: V1.00
```

(continues on next page)

(continued from previous page)

```
S/N: 682863966
VTref = 3.300V
```

```
Type "connect" to establish a target connection, '?' for help
J-Link>erase
Cortex-M4 identified.
Erasing device (0;?i?)...
Comparing flash [100%] Done.
Erasing flash [100%] Done.
Verifying flash [100%] Done.
J-Link: Flash download: Total time needed: 0.363s (Prepare: 0.093s, Compare: 0.000s, Erase: 0.262s, Program: 0.000s, Verify: 0.000s, Restore: 0.008s)
Erasing done.
J-Link>exit
$
```

```
$ newt load telee02_boot
Loading bootloader
$ newt load lora_app_shell_telee02
Loading app image into slot 1
```

Assuming you attached the serial port and have a terminal up you should set the following output on the terminal:

```
000002 lora_app_shell
### Shell Commands
```

There are a number of shell commands that will allow you to join and send both unconfirmed and confirmed data. If you type 'help' in your terminal you will see the various commands displayed. Here is a screen shot of the output of help

```
000002 lora_app_shell
help

032766 help
032766 stat
032767 tasks
032768 mpool
032769 date
032770 las_wr_mib
032771 las_rd_mib
032772 las_rd_dev_eui
032773 las_wr_dev_eui
032774 las_rd_app_eui
032775 las_wr_app_eui
032776 las_rd_app_key
032777 las_wr_app_key
032778 las_app_port
032779 las_app_tx
032780 las_join
032781 las_link_chk
```

(continues on next page)

(continued from previous page)

032782 compat>

The following table lists the commands and gives a brief description of the commands. The lora commands are described in more detail later in the tutorial as well as their syntax (syntax not shown in the table).

Command	Description
help	Display list of available shell commands
stat	Display statistics. Syntax: stat <statistics group>. ‘stat’ with no group displays avaialable groups
tasks	Display OS tasks
mpool	Displays OS memory pools and memory pool statistics
date	Displays current date/time
las_wr_mib	Write lora MIB
las_rd_mib	Read lora MIB
las_rd_dev_eui	Read lora device EUI
las_wr_dev_eui	Write lora device EUI
las_rd_app_eui	Read lora application EUI
las_wr_app_eui	Write lora application EUI
las_rd_app_key	Read lora application key
las_wr_app_key	Write lora application key
las_app_port	Open/close lora application port
las_app_tx	Transmit on lora application port
las_join	Perform a lora OTA join
las_link_chk	Perform a lora link check

4.7.10 OTA Join

Before sending any application data a lora end device must be joined to its lora network. To perform a lora OTA (over-the-air) join there are some commands that must be issued prior to attempting to join. The reason for these commands is that a lora end device must be configured with a device EUI, application EUI and application key prior to performing an OTA join.

```
598763 compat> las_wr_app_eui 0x00:0x11:0x22:0x01:0x01:0x00:0x10:10
623106 compat> las_wr_app_key 03:03:03:03:03:03:03:03:03:03:03:03:03:03:03:03
623758 compat> las_wr_dev_eui 0x00:0x11:0x22:0x02:0x02:0x00:0x00:0x00
630333 compat> las_join 1
630634 Attempting to join...
019802 compat> Join cb. status=0 attempts=1
```

If the join is successful the status returned should be 0. If it fails the status will be a non-zero lora status code (lora status error codes are described later in this tutorial).

A note about “endianness” in the device EUI commands. The first three bytes of the EUI are the OUI and the last 5 bytes are unique (for that OUI). The above example assumes an OUI of 001122. This is not the same order as the address over the air as device addresses are sent “least significant byte” first (little endian). The same convention also applies to keys: they are in big-endian order in the command but sent little endian over the air.

4.7.11 Opening/closing an application port

Another step that must be performed prior to sending application data is to open an application port. All data frames containing application data are sent to a specific port. Port numbers are in the range 1 - 223 as port 0 is reserved for MLME-related activities. Ports 224-255 are reserved for future standardized application extensions.

The lora app shell does not open any application ports by default.

To open and/or close an application port the following commands are used. Note that the application port which you are using to send data must be open if you want to send data (or receive it).

```
115647 compat> las_app_port open 1  
  
150958 Opened app port 1  
150958 compat> las_app_port close 1  
  
151882 Closed app port 1
```

4.7.12 Sending data

The lora app shell allows the user to send both unconfirmed and confirmed data. The command to send data is *las_app_tx <port> <len> <type>*

NOTE: the current usage for this command shows an optional data rate and retries for this command. That feature has not been implemented and the command will not be accepted if they are separated.

Where: port = port number on which to send len = size n bytes of app data type = 0 for unconfirmed, 1 for confirmed

To send a confirmed data transmission of size 5 bytes on port 10 the command would be: *las_app_tx 10 20 1*

Once the end device has sent the frame requested there should be a message which contains some additional information. Here is a screen shot using the above example. Note that there will be some delay between seeing the “Packet sent on port 10” message and the additional information as the additional information is the “confirmation” that the lora stack provides and the confirmation will not be returned until the lora stack is finished transmitting the frame and has received an acknowledgement or has finished waiting for all the receive windows.

```
449751 compat> las_app_tx 10 5 1  
  
452144 Packet sent on port 10  
452144 compat> Txd on port 10 type=conf status=0 len=5  
452325 dr:0  
452325 txpower:5  
452325 tries:1  
452326 ack_rxd:1  
452326 tx_time_on_air:330  
452327 uplink_cntr:0  
452327 uplink_freq:903500000
```

The information contained in the confirmation is the following:

dr: The data rate on which the frame was sent.

txpower: Transmit power level of the device.

tries: # of attempts made to transmit the frame successfully.

ack_rxd: Was an acknowledgement received (0 no 1 yes).

tx_time_on_air: The on-air length of the frame (in milliseconds).

uplink_cntr: The frame uplink counter that this frame used.

uplink_freq: The frequency (logical) on which the frame was sent (in Hz).

4.8 OS Fundamentals

4.8.1 Events and Event Queues

How to Use Event Queues to Manage Multiple Events

- *Introduction*
- *Prerequisites*
- *Example Application*
 - *Create the Project*
 - *Create the Application*
 - *Application Task Generated Events*
 - *OS Callout Timer Events*
 - *Interrupt Events*
 - *Putting It All Together*

Introduction

The event queue mechanism allows you to serialize incoming events for your task. You can use it to get information about hardware interrupts, callout expirations, and messages from other tasks.

The benefit of using events for inter-task communication is that it can reduce the number of resources that need to be shared and locked.

The benefit of processing interrupts in a task context instead of the interrupt context is that other interrupts and high priority tasks are not blocked waiting for the interrupt handler to complete processing. A task can also access other OS facilities and sleep.

This tutorial assumes that you have read about *Event Queues*, the *Hardware Abstraction Layer*, and *OS Callouts* in the OS User's Guide.

This tutorial shows you how to create an application that uses events for:

- Inter-task communication
- OS callouts for timer expiration
- GPIO interrupts

It also shows you how to:

- Use the Mynewt default event queue and application main task to process your events.
- Create a dedicated event queue and task for your events.

To reduce an application's memory requirement, we recommend that you use the Mynewt default event queue if your application or package does not have real-time timing requirements.

Prerequisites

Ensure that you have met the following prerequisites before continuing with this tutorial:

- Install the newt tool.
- Install the newtmgr tool.
- Have Internet connectivity to fetch remote Mynewt components.
- Install the compiler tools to support native compiling to build the project this tutorial creates.
- Have a cable to establish a serial USB connection between the board and the laptop.

Example Application

In this example, you will write an application, for the Nordic nRF52 board, that uses events from three input sources to toggle three GPIO outputs and light up the LEDs. If you are using a different board, you will need to adjust the GPIO pin numbers in the code example.

The application handles events from three sources on two event queues:

- Events generated by an application task at periodic intervals are added to the Mynewt default event queue.
- OS callouts for timer events are added to the `my_timer_interrupt_eventq` event queue.
- GPIO interrupt events are added to the `my_timer_interrupt_eventq` event queue.

Create the Project

Follow the instructions in the [nRF52 tutorial for Blinky](#) to create a project.

Create the Application

Create the `pkg.yml` file for the application:

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

pkg.deps:
  - kernel/os
  - hw/hal
  - sys/console/stub
```

Application Task Generated Events

The application creates a task that generates events, at periodic intervals, to toggle the LED at pin TASK_LED. The event is queued on the Mynewt default event queue and is processed in the context of the application main task.

Declare and initialize the gen_task_ev event with the my_ev_cb() callback function to process the event:

```
/* Callback function for application task event */
static void my_ev_cb(struct os_event *);

/* Initialize the event with the callback function */
static struct os_event gen_task_ev = {
    .ev_cb = my_ev_cb,
};
```

Implement the my_ev_cb() callback function to process a task generated event and toggle the LED at pin TASK_LED:

```
/* LED 1 (P0.17 on the board) */
#define TASK_LED      17

/*
 * Event callback function for events generated by gen_task. It toggles
 * the LED at pin TASK_LED.
 */
static void my_ev_cb(struct os_event *ev)
{
    assert(ev);
    hal_gpio_toggle(TASK_LED);
    return;
}
```

Create a task that generates an event at periodic intervals and adds, using the os_eventq_put() function, the event to the Mynewt default event queue:

```
#define GEN_TASK_PRIO      3
#define GEN_TASK_STACK_SZ   512

static os_stack_t gen_task_stack[GEN_TASK_STACK_SZ];
static struct os_task gen_task_str;

/*
 * Task handler to generate an event to toggle the LED at pin TASK_LED.
 * The event is added to the Mynewt default event queue.
 */
static void
gen_task(void *arg)
{
    while (1) {
        os_time_delay(OS_TICKS_PER_SEC / 4);
        os_eventq_put(os_eventq_dflt_get(), &gen_task_ev);
    }
}

static void
```

(continues on next page)

(continued from previous page)

```
init_tasks(void)
{
    /* Create a task to generate events to toggle the LED at pin TASK_LED */

    os_task_init(&gen_task_str, "gen_task", gen_task, NULL, GEN_TASK_PRIO,
                 OS_WAIT_FOREVER, gen_task_stack, GEN_TASK_STACK_SZ);

    ...
}
```

Implement the application `main()` function to call the `os_eventq_run()` function to dequeue an event from the Mynewt default event queue and call the callback function to process the event.

```
int
main(int argc, char **argv)
{
    sysinit();

    init_tasks();

    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }
    assert(0);
}
```

OS Callout Timer Events

Set up OS callout timer events. For this example, we use a dedicated event queue for timer events to show you how to create a dedicated event queue and a task to process the events.

Implement the `my_timer_ev_cb()` callback function to process a timer event and toggle the LED at pin `CALLOUT_LED`:

```
/* LED 2 (P0.18 on the board) */
#define CALLOUT_LED      18

/* The timer callout */
static struct os_callout my_callout;

/*
 * Event callback function for timer events. It toggles the LED at pin CALLOUT_LED.
 */
static void my_timer_ev_cb(struct os_event *ev)
{
    assert(ev != NULL);

    hal_gpio_toggle(CALLOUT_LED);

    os_callout_reset(&my_callout, OS_TICKS_PER_SEC / 2);
}
```

In the `init_tasks()` function, initialize the `my_timer_interrupt_eventq` event queue, create a task to process events from the queue, and initialize the OS callout for the timer:

```
#define MY_TIMER_INTERRUPT_TASK_PRIO 4
#define MY_TIMER_INTERRUPT_TASK_STACK_SZ 512

static os_stack_t my_timer_interrupt_task_stack[MY_TIMER_INTERRUPT_TASK_STACK_SZ];
static struct os_task my_timer_interrupt_task_str;

static void
init_tasks(void)
{
    /* Use a dedicate event queue for timer and interrupt events */

    os_eventq_init(&my_timer_interrupt_eventq);

    /*
     * Create the task to process timer and interrupt events from the
     * my_timer_interrupt_eventq event queue.
     */
    os_task_init(&my_timer_interrupt_task_str, "timer_interrupt_task",
                my_timer_interrupt_task, NULL,
                MY_TIMER_INTERRUPT_TASK_PRIO, OS_WAIT_FOREVER,
                my_timer_interrupt_task_stack,
                MY_TIMER_INTERRUPT_TASK_STACK_SZ);

    /*
     * Initialize the callout for a timer event.
     * The my_timer_ev_cb callback function processes the timer events.
     */
    os_callout_init(&my_callout, &my_timer_interrupt_eventq,
                    my_timer_ev_cb, NULL);

    os_callout_reset(&my_callout, OS_TICKS_PER_SEC);
}
```

Implement the `my_timer_interrupt_task()` task handler to dispatch events from the `my_timer_interrupt_eventq` event queue:

```
static void
my_timer_interrupt_task(void *arg)
{
    while (1) {
        os_eventq_run(&my_timer_interrupt_eventq);
    }
}
```

Interrupt Events

The application toggles the LED each time button 1 on the board is pressed. The interrupt handler generates an event when the GPIO for button 1 (P0.13) changes state. The events are added to the `my_timer_interrupt_eventq` event queue, the same queue as the timer events.

Declare and initialize the `gpio_ev` event with the `my_interrupt_ev_cb()` callback function to process the event:

```
static struct os_event gpio_ev {
    .ev_cb = my_interrupt_ev_cb,
};
```

Implement the `my_interrupt_ev_cb()` callback function to process an interrupt event and toggle the LED at pin `GPIO_LED`:

```
/* LED 3 (P0.19 on the board) */
#define GPIO_LED      19

/*
 * Event callback function for interrupt events. It toggles the LED at pin GPIO_LED.
 */
static void my_interrupt_ev_cb(struct os_event *ev)
{
    assert(ev != NULL);

    hal_gpio_toggle(GPIO_LED);
}
```

Implement the `my_gpio_irq()` handler to post an interrupt event to the `my_timer_interrupt_eventq` event queue:

```
static void
my_gpio_irq(void *arg)
{
    os_eventq_put(&my_timer_interrupt_eventq, &gpio_ev);
}
```

In the `init_tasks()` function, add the code to set up and enable the GPIO input pin for the button and initialize the GPIO output pins for the LEDs:

```
/* LED 1 (P0.17 on the board) */
#define TASK_LED      17

/* 2 (P0.18 on the board) */
#define CALLOUT_LED   18

/* LED 3 (P0.19 on the board) */
#define GPIO_LED      19

/* Button 1 (P0.13 on the board) */
#define BUTTON1_PIN    13

void
init_tasks()
```

(continues on next page)

(continued from previous page)

```

/* Initialize OS callout for timer events. */

....
```

- /*
- * Initialize and enable interrupts for the pin for button 1 and
- * configure the button with pull up resistor on the nrf52dk.
- */
- hal_gpio_irq_init(BUTTON1_PIN, my_gpio_irq, NULL, HAL_GPIO_TRIG_RISING, HAL_GPIO_
- PULL_UP);

```

hal_gpio_irq_enable(BUTTON1_PIN);

/* Initialize the GPIO output pins. Value 1 is off for these LEDs. */

hal_gpio_init_out(TASK_LED, 1);
hal_gpio_init_out(CALLOUT_LED, 1);
hal_gpio_init_out(GPIO_LED, 1);
}
```

Putting It All Together

Here is the complete `main.c` source for your application. Build the application and load it on your board. The task LED (LED1) blinks at an interval of 250ms, the callout LED (LED2) blinks at an interval of 500ms, and the GPIO LED (LED3) toggles on or off each time you press Button 1.

```

#include <os/os.h>
#include <bsp/bsp.h>
#include <hal/hal_gpio.h>
#include <assert.h>
#include <sysinit/sysinit.h>

#define MY_TIMER_INTERRUPT_TASK_PRIO 4
#define MY_TIMER_INTERRUPT_TASK_STACK_SZ 512

#define GEN_TASK_PRIO 3
#define GEN_TASK_STACK_SZ 512

/* LED 1 (P0.17 on the board) */
#define TASK_LED 17

/* LED 2 (P0.18 on the board) */
#define CALLOUT_LED 18

/* LED 3 (P0.19 on the board) */
#define GPIO_LED 19

/* Button 1 (P0.13 on the board) */
#define BUTTON1_PIN 13
```

(continues on next page)

(continued from previous page)

```
static void my_ev_cb(struct os_event *);  
static void my_timer_ev_cb(struct os_event *);  
static void my_interrupt_ev_cb(struct os_event *);  
  
static struct os_eventq my_timer_interrupt_eventq;  
  
static os_stack_t my_timer_interrupt_task_stack[MY_TIMER_INTERRUPT_TASK_STACK_SZ];  
static struct os_task my_timer_interrupt_task_str;  
  
static os_stack_t gen_task_stack[GEN_TASK_STACK_SZ];  
static struct os_task gen_task_str;  
  
static struct os_event gen_task_ev = {  
    .ev_cb = my_ev_cb,  
};  
  
static struct os_event gpio_ev = {  
    .ev_cb = my_interrupt_ev_cb,  
};  
  
static struct os_callout my_callout;  
  
/*  
 * Task handler to generate an event to toggle the LED at pin TASK_LED.  
 * The event is added to the Mynewt default event queue.  
 */  
  
static void  
gen_task(void *arg)  
{  
    while (1) {  
        os_time_delay(OS_TICKS_PER_SEC / 4);  
        os_eventq_put(os_eventq_dflt_get(), &gen_task_ev);  
    }  
}  
  
/*  
 * Event callback function for events generated by gen_task. It toggles the LED at pin  
 * TASK_LED.  
 */  
static void my_ev_cb(struct os_event *ev)  
{  
    assert(ev);  
    hal_gpio_toggle(TASK_LED);  
    return;  
}  
  
/*  
 * Event callback function for timer events. It toggles the LED at pin CALLOUT_LED.  
 */
```

(continues on next page)

(continued from previous page)

```

static void my_timer_ev_cb(struct os_event *ev)
{
    assert(ev != NULL);

    hal_gpio_toggle(CALLOUT_LED);
    os_callout_reset(&my_callout, OS_TICKS_PER_SEC / 2);
}

/*
 * Event callback function for interrupt events. It toggles the LED at pin GPIO_LED.
 */
static void my_interrupt_ev_cb(struct os_event *ev)
{
    assert(ev != NULL);

    hal_gpio_toggle(GPIO_LED);
}

static void
my_gpio_irq(void *arg)
{
    os_eventq_put(&my_timer_interrupt_eventq, &gpio_ev);
}

static void
my_timer_interrupt_task(void *arg)
{
    while (1) {
        os_eventq_run(&my_timer_interrupt_eventq);
    }
}

void
init_tasks(void)
{
    /* Create a task to generate events to toggle the LED at pin TASK_LED */

    os_task_init(&gen_task_str, "gen_task", gen_task, NULL, GEN_TASK_PRIO,
                OS_WAIT_FOREVER, gen_task_stack, GEN_TASK_STACK_SZ);

    /* Use a dedicate event queue for timer and interrupt events */
    os_eventq_init(&my_timer_interrupt_eventq);

    /*
     * Create the task to process timer and interrupt events from the
     * my_timer_interrupt_eventq event queue.
     */
    os_task_init(&my_timer_interrupt_task_str, "timer_interrupt_task",

```

(continues on next page)

(continued from previous page)

```

my_timer_interrupt_task, NULL,
MY_TIMER_INTERRUPT_TASK_PRIO, OS_WAIT_FOREVER,
my_timer_interrupt_task_stack,
MY_TIMER_INTERRUPT_TASK_STACK_SZ);

/*
 * Initialize the callout for a timer event.
 * The my_timer_ev_cb callback function processes the timer event.
 */
os_callout_init(&my_callout, &my_timer_interrupt_eventq,
                my_timer_ev_cb, NULL);

os_callout_reset(&my_callout, OS TICKS_PER_SEC);

/*
 * Initialize and enable interrupt for the pin for button 1 and
 * configure the button with pull up resistor on the nrf52dk.
 */
hal_gpio_irq_init(BUTTON1_PIN, my_gpio_irq, NULL, HAL_GPIO_TRIG_RISING, HAL_GPIO_
PULL_UP);

hal_gpio_irq_enable(BUTTON1_PIN);

hal_gpio_init_out(TASK_LED, 1);
hal_gpio_init_out(CALLOUT_LED, 1);
hal_gpio_init_out(GPIO_LED, 1);
}

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

    init_tasks();

    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }
    assert(0);
}

```

4.8.2 Tasks and Priority Management

Target Platform: Arduino M0 Pro (or legacy Arduino Zero or Zero Pro, but not Arduino M0)

This lesson is designed to teach core OS concepts and strategies encountered when building applications using Mynewt. Specifically, this lesson will cover tasks, simple multitasking, and priority management running on an Arduino M0 Pro.

- *Prerequisites*
- *Equipment*
- *Build Your Application*
- *Default Main Task*
- *Create a New Task*
 - *Task Stack*
 - *Task Function*
 - *Task Priority*
 - *Initialization*
- *Task Priority, Preempting, and Context Switching*
 - *Priority Management Considerations*

Prerequisites

Before starting, you should read about Mynewt in the [Introduction](#) section and complete the [QuickStart](#) guide and the [Blinky](#) tutorial. Furthermore, it may be helpful to take a peek at the [task documentation](#) for additional insights.

Equipment

You will need the following equipment:

- Arduino M0 Pro (or legacy Arduino Zero or Zero Pro, but not Arduino M0)
- Computer with Mynewt installed
- USB to Micro USB Cable

Build Your Application

To save time, we will simply modify the Blinky application. We'll add the Task Management code to the Blinky application. Follow the [Arduino Zero Blinky tutorial](#) to create a new project and build your bootloader and application. Finally, build and load the application to your Arduino to verify that everything is in order. Now let's get started!

Default Main Task

During Mynewt system startup, Mynewt creates a default main task and executes the application `main()` function in the context of this task. The main task priority defaults to 127 and can be configured with the `OS_MAIN_TASK_PRIO` system configuration setting.

The blinky application only has the `main` task. The `main()` function executes an infinite loop that toggles the led and sleeps for one second.

Create a New Task

The purpose of this section is to give an introduction to the important aspects of tasks and how to properly initialize them. First, let's define a second task called `work_task` in `main.c` (located in `apps/blinky/src`):

```
struct os_task work_task;
```

A task is represented by the `os_task` struct which will hold the task's information (name, state, priority, etc.). A task is made up of two main elements, a task function (also known as a task handler) and a task stack.

Next, let's take a look at what is required to initialize our new task.

Task Stack

The task stack is an array of type `os_stack_t` which holds the program stack frames. Mynewt gives us the ability to set the stack size for a task giving the application developer room to optimize memory usage. Since we're not short on memory, our `work_stack` is plenty large for the purpose of this lesson. Notice that the elements in our task stack are of type `os_stack_t` which are generally 32 bits, making our entire stack 1024 Bytes.

```
#define WORK_STACK_SIZE OS_STACK_ALIGN(256)
```

Note: The `OS_STACK_ALIGN` macro is used to align the stack based on the hardware architecture.

Task Function

A task function is essentially an infinite loop that waits for some “event” to wake it up. In general, the task function is where the majority of work is done by a task. Let's write a task function for `work_task` called `work_task_handler()`:

```
void
work_task_handler(void *arg)
{
    struct os_task *t;

    g_led_pin = LED_BLINK_PIN;
    hal_gpio_init_out(g_led_pin, 1);

    while (1) {
        t = os_sched_get_current_task();
        assert(t->t_func == work_task_handler);
        /* Do work... */
    }
}
```

The task function is called when the task is initially put into the *running* state by the scheduler. We use an infinite loop to ensure that the task function never returns. Our assertion that the current task's handler is the same as our task handler is for illustration purposes only and does not need to be in most task functions.

Task Priority

As a preemptive, multitasking RTOS, Mynewt decides which tasks to run based on which has a higher priority; the highest priority being 0 and the lowest 255. Thus, before initializing our task, we must choose a priority defined as a macro variable.

Let's set the priority of `work_task` to 0, because everyone knows that work is more important than blinking.

```
#define WORK_TASK_PRIO (0)
```

Initialization

To initialize a new task we use `os_task_init()` which takes a number of arguments including our new task function, stack, and priority.

Add the `init_tasks()` function to initialize `work_task` to keep our main function clean.

```
int
init_tasks(void)
{
    /* ... */
    os_stack_t *work_stack;
    work_stack = malloc(sizeof(os_stack_t)*WORK_STACK_SIZE);

    assert(work_stack);
    os_task_init(&work_task, "work", work_task_handler, NULL,
                WORK_TASK_PRIO, OS_WAIT_FOREVER, work_stack,
                WORK_STACK_SIZE);

    return 0;
}
```

Add the call to `init_tasks()` in `main()` before the `while` loop:

```
int
main(int argc, char **argv)
{
    ...

    /* Initialize the work task */
    init_tasks();

    while (1) {
        ...
    }
}
```

And that's it! Now run your application using the `newt run` command.

```
$ newt run arduino_blinky 0.0.0
```

When GDB appears press C then Enter to continue and ... *wait, why doesn't our LED blink anymore?*

Review

Before we run our new app, let's review what we need in order to create a task. This is a general case for a new task called mytask:

- 1) Define a new task, task stack, and priority:

```
/* My Task */
struct os_task mytask
/* My Task Stack */
#define MYTASK_STACK_SIZE OS_STACK_ALIGN(256)
os_stack_t mytask_stack[MYTASK_STACK_SIZE];
/* My Task Priority */
#define MYTASK_PRIO (0)
```

- 2) Define task function:

```
void
mytask_handler(void *arg)
{
    while (1) {
        /* ... */
    }
}
```

- 3) Initialize the task:

```
os_task_init(&mytask, "mytask", mytask_handler, NULL,
            MYTASK_PRIO, OS_WAIT_FOREVER, mytask_stack,
            MYTASK_STACK_SIZE);
```

Task Priority, Preempting, and Context Switching

A preemptive RTOS is one in which a higher priority task that is *ready to run* will preempt (i.e. take the place of) the lower priority task which is *running*. When a lower priority task is preempted by a higher priority task, the lower priority task's context data (stack pointer, registers, etc.) is saved and the new task is switched in.

In our example, `work_task` (priority 0) has a higher priority than the `main` task (priority 127). Since `work_task` is never put into a `sleep` state, it holds the processor focus on its context.

Let's give `work_task` a delay and some simulated work to keep it busy. The delay is measured in os ticks and the actual number of ticks per second is dependent on the board. We multiply `OS_TICKS_PER_SEC`, which is defined in the MCU, by the number of seconds we wish to delay.

```
void
work_task_handler(void *arg)
{
    struct os_task *t;

    g_led_pin = LED_BLINK_PIN;
    hal_gpio_init_out(g_led_pin, 1);

    while (1) {
        t = os_sched_get_current_t:ask();
```

(continues on next page)

(continued from previous page)

```

assert(t->t_func == work_task_handler);
/* Do work... */
int i;
for(i = 0; i < 1000000; ++i) {
    /* Simulate doing a noticeable amount of work */
    hal_gpio_write(g_led_pin, 1);
}
os_time_delay(3 * OS_TICKS_PER_SEC);
}
}

```

In order to notice the LED changing, modify the time delay in `main()` to blink at a higher frequency.

```
os_time_delay(OS_TICKS_PER_SEC/10);
```

Before we run the app, let's predict the behavior. With the newest additions to `work_task_handler()`, our first action will be to sleep for three seconds. This allows the main task, running `main()`, to take over the CPU and blink to its heart's content. After three seconds, `work_task` will wake up and be made *ready to run*. This causes it to preempt the `main` task. The LED will then remain lit for a short period while `work_task` loops, then blink again for another three seconds while `work_task` sleeps.

You should see that our prediction was correct!

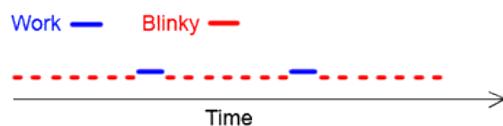
Priority Management Considerations

When projects grow in scope, from blinking LEDs into more sophisticated applications, the number of tasks needed increases alongside complexity. It remains important, then, that each of our tasks is capable of doing its work within a reasonable amount of time.

Some tasks, such as the Shell task, execute quickly and require almost instantaneous response. Therefore, the Shell task should be given a high priority. On the other hand, tasks which may be communicating over a network, or processing data, should be given a low priority in order to not hog the CPU.

The diagram below shows the different scheduling patterns we would expect when we set the `work_task` priority higher and lower than the `main` task priority.

Work Task Priority = 0, Blinky (Main Task) Priority = 127



Work Task Priority = 128, Blinky (Main Task) Priority = 127

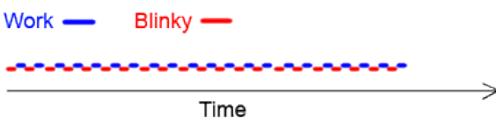


Fig. 4: Task Scheduling

In the second case where the `main` task has a higher priority, `work_task` runs and executes “work” when the `main` task sleeps, saving us idle time compared to the first case.

Note: Defining the same priority for two tasks fires an assert in `os_task_init()` and must be avoided. Priority 127 is reserved for main task, 255 for idle task.

4.9 Remote Device Management

4.9.1 Enabling Newt Manager in Your Application

In order for your application to communicate with the `newtmgr` tool and process Newt Manager commands, you must enable Newt Manager device management and the support to process Newt Manager commands in your application. This tutorial explains how to add the support to your application.

This tutorial assumes that you have read the *Device Management with Newt Manager* guide and are familiar with the `newtmgr` and `oicmgr` frameworks and all the options that are available to customize your application.

This tutorial shows you how to configure your application to:

- Use the `newtmgr` framework.
- Use serial transport to communicate with the `newtmgr` tool.
- Support all Newt Manager commands.

See *Other Configuration Options* on how to customize your application.

- *Prerequisites*
- *Use an Existing Project*
- *Modify Package Dependencies and Configurations*
- *Modify the Source*
- *Build the Targets*
- *Create the Application Image*
- *Load the Image*
- *Set Up a Connection Profile*
- *Communicate with Your Application*
- *Other Configuration Options*
 - *Newtmgr Framework Transport Protocol Options*
 - *Oicmgr Framework Options*
 - *Customize the Newt Manager Commands that Your Application Supports*

Prerequisites

Ensure that you have met the following prerequisites before continuing with this tutorial:

- Have Internet connectivity to fetch remote Mynewt components.
- Have a cable to establish a serial USB connection between the board and the laptop.
- Install the newt tool and toolchains (See [Basic Setup](#)).
- Install the [newtmgr tool](#).

Use an Existing Project

We assume that you have worked through at least some of the other tutorials and have an existing project. In this example, we modify the btshell app to enable Newt Manager support. We call our target `myble`. You can create the target using any name you choose.

Modify Package Dependencies and Configurations

Add the following packages to the `pkg.deps` parameter in your target or application `pkg.yml` file:

```
pkg.deps:
  - "@apache-mynewt-core/mgmt/mgmt"
  - "@apache-mynewt-core/mgmt/smp/transport/smp_shell"
  - "@apache-mynewt-core/mgmt/imgmgr"
  - "@apache-mynewt-core/sys/console/full"
  - "@apache-mynewt-core/sys/log/full"
  - "@apache-mynewt-core/sys/stats/full"
  - "@apache-mynewt-core/sys/config"
  - "@apache-mynewt-core/test/crash_test"
  - "@apache-mynewt-core/test/runtest"
```

Each package provides the following Newt Manager functionality:

- `mgmt/mgmt`: Supports the `newtmgr` framework and the Newt Manager `echo`, `taskstat` `mpstat`, `datetime`, and `reset` commands.
- `mgmt/newtmgr/smp/transport/nmgr_shell`: Supports serial transport.
- `mgmt/imgmgr`: Supports the `newtmgr image` command.
- `sys/console/full`: Supports a text-based IO interface with completion.
- `sys/log/full`: Supports the `newtmgr log` command.
- `sys/stats/full`: Supports the `newtmgr stat` command.
- `sys/config`: Supports the `newtmgr config` command.
- `test/crash_test`: Supports the `newtmgr crash` command.
- `test/runtest`: Supports the `newt run` command.

Add the following configuration setting values to the `syscfg.vals` parameter in the target or application `syscfg.yml` file:

```
syscfg.vals:  
    LOG_NEWTMGR: 1  
    STATS_NEWTMGR: 1  
    CONFIG_NEWTMGR: 1  
    CRASH_TEST_NEWTMGR: 1  
    RUNTEST_NEWTMGR: 1  
    SHELL_TASK: 1  
    SHELL_NEWTMGR: 1
```

The first five configuration settings enable support for the Newt Manager `log`, `stat`, `config`, `crash`, and `run` commands. The `SHELL_TASK` setting enables the shell for serial transport. The `SHELL_NEWTMGR` setting enables newtmgr support in the shell.

Note that you may need to override additional configuration settings that are specific to each package to customize the package functionality.

Modify the Source

By default, the `mgmt` package uses the Mynewt default event queue to receive request events from the `newtmgr` tool. These events are processed in the context of the application main task.

You can specify a different event queue for the package to use. If you choose to use a dedicated event queue, you must create a task to process events from this event queue. The `mgmt` package executes and handles `newtmgr` request events in the context of this task. The `mgmt` package exports the `mgmt_evq_set()` function that allows you to specify an event queue.

This example uses the Mynewt default event queue and you do not need to modify your application source.

If you choose to use a different event queue, see [Events and Event Queues](#) for details on how to initialize an event queue and create a task to process the events. You will also need to modify your `main.c` to add the call to the `mgmt_evq_set()` function as follows:

Add the `mgmt/mgmt.h` header file:

```
#include <mgmt/mgmt.h>
```

Add the call to specify the event queue. In the `main()` function, scroll down to the `while (1)` loop and add the following statement above the loop:

```
mgmt_evq_set(&my_eventq)
```

where `my_eventq` is an event queue that you have initialized.

Build the Targets

Build the two targets as follows:

```
$ newt build nrf52_boot  
<snip>  
App successfully built: ./bin/nrf52_boot/boot/mynewt/mynewt.elf  
$ newt build myble  
Compiling hci_common.c  
Compiling util.c  
Archiving nimble.a
```

(continues on next page)

(continued from previous page)

Compiling os.c <snip>

Create the Application Image

Generate an application image for the `myble` target. You can use any version number you choose.

\$ newt create-image myble 1.0.0 App image successfully generated: ./bin/makerbeacon/apps/btshell/btshell.img Build manifest: ./bin/makerbeacon/apps/btshell/manifest.json
--

Load the Image

Ensure the USB connector is in place and the power LED on the board is lit. Turn the power switch on your board off, then back on to reset the board after loading the image.

\$ newt load nrf52_boot \$ newt load myble

Set Up a Connection Profile

The `newtmgr` tool requires a connection profile in order to connect to your board. If you have not done so, follow the [instructions](#) for setting up your connection profile.

Communicate with Your Application

Once you have a connection profile set up, you can connect to your device with `newtmgr -c myconn <command>` to run commands in your application.

Issue the `echo` command to ensure that your application is communicating with the `newtmgr` tool:

\$ newtmgr -c myconn echo hello hello
--

Test your application to ensure that it can process a Newt Manager command that is supported by a different package. Issue the `stat` command to see the BLE stats.

stat group: ble_att 0 error_rsp_rx 0 error_rsp_tx 0 exec_write_req_rx 0 exec_write_req_tx 0 exec_write_rsp_rx 0 exec_write_rsp_tx 0 find_info_req_rx 0 find_info_req_tx 0 find_info_rsp_rx 0 find_info_rsp_tx 0 find_type_value_req_rx

(continues on next page)

(continued from previous page)

```
    ...
    0 read_type_req_tx
    0 read_type_rsp_rx
    0 read_type_rsp_tx
    0 write_cmd_rx
    0 write_cmd_tx
    0 write_req_rx
    0 write_req_tx
    0 write_rsp_rx
    0 write_rsp_tx
```

Your application is now able to communicate with the newtmgr tool.

Other Configuration Options

This section explains how to customize your application to use other Newt Manager protocol options.

Newtmgr Framework Transport Protocol Options

The newtmgr framework currently supports BLE and serial transport protocols. To configure the transport protocols that are supported, modify the `pkg.yml` and `syscfg.yml` files as follows:

- Add the `mgmt/newtmgr/transport/ble` package to the `pkg.deps` parameter to enable BLE transport.
- Add the `mgmt/newtmgr/transport/nmgr_shell` package to the `pkg.deps` parameter, and add `SHELL_TASK: 1` and `SHELL_NEWTMGR` to the `syscfg.vals` parameter to enable serial transport when your application also uses the *Shell*.
- Add the `mgmt/newtmgr/transport/nmgr_uart` package to the `pkg.deps` parameter to enable serial transport over a UART port. You can use this package instead of the `nmgr_shell` package when your application does not use the *Shell* or you want to use a dedicated UART port to communicate with newtmgr. You can change the `NMGR_UART` and `NMGR_URART_SPEED` sysconfig values to specify a different port.

Oicmgr Framework Options

To use the oicmgr framework instead of the newtmgr framework, modify the `pkg.yml` and `syscfg.yml` files as follows:

- Add the `mgmt/oicmgr` package (instead of the `mgmt/newtmgr` and `mgmt/newtmgr/transport` packages as described previously) to the `pkg.deps` parameter.
- Add `OC_SERVER: 1` to the `syscfg.vals` parameter.

Oicmgr supports the IP, serial, and BLE transport protocols. To configure the transport protocols that are supported, set the configuration setting values in the `syscfg.vals` parameter as follows:

- Add `OC_TRANSPORT_IP: 1` to enable IP transport.
- Add `OC_TRANSPORT_GATT: 1` to enable BLE transport.
- Add `OC_TRANSPORT_SERIAL: 1, SHELL_TASK: 1, SHELL_NEWTMGR: 1` to enable serial transport.

Customize the Newt Manager Commands that Your Application Supports

We recommend that you only enable support for the Newt Manager commands that your application uses to reduce your application code size. To configure the commands that are supported, set the configuration setting values in the `syscfg.vals` parameter as follows:

- Add `LOG_NEWTMGR`: 1 to enable support for the `newtmgr log` command.
- Add `STATS_NEWTMGR`: 1 to enable support for the `newtmgr stat` command.
- Add `CONFIG_NEWTMGR`: 1 to enable support for the `newtmgr config` command.
- Add `CRASH_TEST_NEWTMGR`: 1 to enable support for the `newtmgr crash` command.
- Add `RUNTEST_NEWTMGR`: 1 to enable support for the `newtmgr crash` command.

Notes:

- When you enable Newt Manager support, using either the `newtmgr` or `oicmgr` framework, your application automatically supports the Newt Manager `echo`, `taskstat`, `mpstat`, `datetime`, and `reset` commands. These commands cannot be configured individually.
- The `mgmt/imgmgr` package does not provide a configuration setting to enable or disable support for the `newtmgr image` command. Do not specify the package in the `pkg.deps` parameter if your device has limited flash memory and cannot support Over-The-Air (OTA) firmware upgrades.

4.9.2 Over-the-Air Image Upgrade

Mynewt OS supports over-the-air image upgrades. This tutorial shows you how to use the `newtmgr` tool to upgrade an image on a device over BLE communication.

To support over-the-air image upgrade over BLE, a device must be running a Mynewt application that has `newtmgr` image management over BLE transport enabled. For this tutorial, we use the `bleprph` application, which includes image management over BLE functionality, on an nRF52-DK board. If you prefer to use a different BLE application, see [Enable Newt Manager in any app](#) to enable `newtmgr` image management over BLE transport support in your application.

Note: Over-the-air upgrade via `newtmgr` BLE transport is supported on Mac OS and Linux. It is not supported on Windows platforms.

- *Prerequisites*
- *Reducing the Log Level*
- *Upgrading an Image on a Device*
- *Step 1: Creating a Newtmgr Connection Profile*
- *Step 2: Uploading an Image to the Device*
- *Step 3: Testing the Image*
- *Step 4: Confirming the Image*

Prerequisites

Ensure that you meet the following prerequisites:

- Have Internet connectivity to fetch remote Mynewt components.
- Have a computer that supports Bluetooth to communicate with the board and to build a Mynewt application.
- Have a Micro-USB cable to connect the board and the computer.
- Have a Nordic nRF52-DK Development Kit - PCA 10040
- Install the [Segger JLINK](#) software and documentation pack.
- Install the newt tool and toolchains (See [Setup & Get Started](#)).
- Read the Mynewt OS [Concepts](#) section.
- Read the [Secure Bootloader](#) section and understand the Mynewt bootloader concepts.
- Build and load the **bleprph** application on to an nRF52-DK board via a serial connection. See [BLE Peripheral App](#).

Reducing the Log Level

You need to build your application with log level set to INFO or lower. The default log level for the **bleprph** app is set to DEBUG. The extra logging causes the communication to timeout. Perform the following to reduce the log level to INFO, build, and load the application.

```
$ newt target amend myperiph syscfg="LOG_LEVEL=1"  
$ newt build myperiph  
$ newt create-image myperiph 1.0.0  
$ newt load myperiph
```

Upgrading an Image on a Device

Once you have an application with newtmgr image management with BLE transport support running on a device, you can use the newtmgr tool to upgrade an image over-the-air.

You must perform the following steps to upgrade an image:

Step 1: Create a newtmgr connection profile to communicate with the device over BLE. Step 2: Upload the image to the secondary slot (slot 1) on the device. Step 3: Test the image. Step 4: Confirm and make the image permanent.

See the [Secure Bootloader](#) section for more information on the bootloader, image slots, and boot states.

Step 1: Creating a Newtmgr Connection Profile

The **bleprph** application sets and advertises nimble-bleprph as its bluetooth device address. Run the `newtmgr conn add` command to create a newtmgr connection profile that uses this peer address to communicate with the device over BLE:

```
$ newtmgr conn add mybleprph type=ble connstring="peer_name=nimble-bleprph"  
Connection profile mybleprph successfully added
```

Verify that the newtmgr tool can communicate with the device and check the image status on the device:

```
$ newtmgr image list -c mybleprph
Images:
slot=0
  version: 1.0.0
  bootable: true
  flags: active confirmed
  hash: b8d17c77a03b37603cd9f89fdcce0ba726f8ddff6eac63011dee2e959cc316c2
Split status: N/A (0)
```

The device only has an image loaded on the primary slot (slot 0). It does not have an image loaded on the secondary slot (slot 1).

Step 2: Uploading an Image to the Device

We create an image with version 2.0.0 for the bleprph application from the myperiph target and upload the new image. You can upload a different image.

```
$ newt create-image myperiph 2.0.0
App image successfully generated: ~/dev/myproj/bin/targets/myperiph/app/apps/bleprph/
└─bleprph.img
```

Run the `newtmgr image upload` command to upload the image:

```
$ newtmgr image upload -c mybleprph ~/dev/myproj/bin/targets/myperiph/app/apps/bleprph/
└─bleprph.img
215
429
642
855
1068
1281

...
125953
126164
126375
126586
126704
Done
```

The numbers indicate the number of bytes that the `newtmgr` tool has uploaded.

Verify that the image uploaded to the secondary slot on the device successfully:

```
$ newtmgr image list -c mybleprph
Images:
slot=0
  version: 1.0.0
  bootable: true
  flags: active confirmed
  hash: b8d17c77a03b37603cd9f89fdcce0ba726f8ddff6eac63011dee2e959cc316c2
slot=1
```

(continues on next page)

(continued from previous page)

```
version: 2.0.0
bootable: true
flags:
hash: 291ebc02a8c345911c96fdf4e7b9015a843697658fd6b5faa0eb257a23e93682
Split status: N/A (0)
```

The device now has the uploaded image in the secondary slot (slot 1).

Step 3: Testing the Image

The image is uploaded to the secondary slot but is not yet active. You must run the `newtmgr image test` command to set the image status to **pending** and reboot the device. When the device reboots, the bootloader copies this image to the primary slot and runs the image.

```
$ newtmgr image test -c mybleprph
→291ebc02a8c345911c96fdf4e7b9015a843697658fd6b5faa0eb257a23e93682
Images:
slot=0
    version: 1.0.0
    bootable: true
    flags: active confirmed
    hash: b8d17c77a03b37603cd9f89fdcfe0ba726f8dff6eac63011dee2e959cc316c2
slot=1
    version: 2.0.0
    bootable: true
    flags: pending
    hash: 291ebc02a8c345911c96fdf4e7b9015a843697658fd6b5faa0eb257a23e93682
Split status: N/A (0)
```

The status of the image in the secondary slot is now set to **pending**.

Power the device OFF and ON and run the `newtmgr image list` command to check the image status on the device after the reboot:

```
$ newtmgr image list -c mybleprph
Images:
slot=0
    version: 2.0.0
    bootable: true
    flags: active
    hash: 291ebc02a8c345911c96fdf4e7b9015a843697658fd6b5faa0eb257a23e93682
slot=1
    version: 1.0.0
    bootable: true
    flags: confirmed
    hash: b8d17c77a03b37603cd9f89fdcfe0ba726f8dff6eac63011dee2e959cc316c2
Split status: N/A (0)
```

The uploaded image is now active and running in the primary slot. The image, however, is not confirmed. The confirmed image is in the secondary slot. On the next reboot, the bootloader reverts to using the confirmed image. It copies the confirmed image to the primary slot and runs the image when the device reboots. You need to confirm and make the uploaded image in the primary slot permanent.

Step 4: Confirming the Image

Run the `newtmgr image confirm` command to confirm and make the uploaded image permanent. Since the uploaded image is currently the active image, you can confirm the image setup without specifying the image hash value in the command:

```
$ newtmgr image confirm -c mybleprph
Images:
slot=0
    version: 2.0.0
    bootable: true
    flags: active confirmed
    hash: 291ebc02a8c345911c96fdf4e7b9015a843697658fd6b5faa0eb257a23e93682
slot=1
    version: 1.0.0
    bootable: true
    flags:
    hash: b8d17c77a03b37603cd9f89fdcfe0ba726f8dff6eac63011dee2e959cc316c2
Split status: N/A (0)
```

The uploaded image is now the active and confirmed image. You have successfully upgraded an image over-the-air.

4.10 Sensors

4.10.1 Sensor Tutorials Overview

This set of sensor tutorials allows you to explore the Mynewt Sensor Framework features and learn how to develop sensor-enabled Mynewt applications.

The Mynewt Sensor framework supports:

- Onboard and off-board sensors.
- Retrieving sensor data and controlling sensor devices via the Mynewt OS Shell.
- Retrieving sensor data over the OIC protocol and BLE transport.

4.10.2 Available Tutorials

The tutorials are:

- [*Enabling an Off-Board Sensor in an Existing Application*](#) - This is an introductory tutorial that shows you how to quickly bring up a sensor enabled application that retrieves data from a sensor device. We recommend that you work through this tutorial before trying one of the other tutorials.
- [*Changing the Default Configuration for a Sensor*](#) - This tutorial shows you how to change the default configuration values for a sensor.
- [*Developing an Application for an Onboard Sensor*](#) - This tutorial shows you how to develop a simple application for a device with an onboard sensor.
- [*Enabling OIC Sensor Data Monitoring*](#) - This tutorial shows you how to enable support for sensor data monitoring via OIC in an existing application.

4.10.3 Mynewt Smart Device Controller OIC App

We use the Mynewt Sensor Monitor App on iOS or Android to retrieve and display sensor data from the Mynewt OS OIC sensor applications described in the OIC Sensor Data Monitoring tutorials. You can download the app from either the Apple Store or Google Play Store.

Note: At the time of writing this tutorial, the iOS app was still in the queue waiting to be placed in the App Store. You can build the iOS app from source as indicated below.

If you would like to contribute or modify the Mynewt Smart Device Controller App, see the [Android Sensor source](#) and [iOS Sensor source](#) on github.

4.10.4 Prerequisites

Ensure that you meet the following prerequisites before continuing with one of the tutorials.

- Have Internet connectivity to fetch remote Mynewt components.
- Have a computer to build a Mynewt application and connect to the board over USB.
- Have a Micro-USB cable to connect the board and the computer.
- Install the newt tool and toolchains (See [Basic Setup](#)).
- Read the Mynewt OS [Concepts](#) section.
- Create a project space (directory structure) and populate it with the core code repository (apache-mynewt-core) explained in [Creating Your First Project](#)
- Work through one of the [Blinky Tutorials](#)

4.10.5 Sensor Framework

Enabling an Off-Board Sensor in an Existing Application

This tutorial shows you how to enable an existing application to run on a device with an off-board sensor device connected to it. It allows you to quickly bring up and run a Mynewt application on a device to view sensor data from a sensor device.

We use the [sensors_test](#) application running on an nRF52-DK board to communicate, via the I2C interface, with the [Adafruit BNO055](#) sensor. The sensors_test application is a sample application that demonstrates all the features of the Mynewt sensor framework. The application includes the sensor framework sensor shell command that allows you to view the sensors and sensor data managed by the sensor framework, and the bno055 shell command that allows you to control and query the BNO055 device and to view the sensor data.

This tutorial shows you how to:

- Create and build the application and bootloader targets.
- Connect a BNO055 sensor device to an nRF52-DK board.
- Run sensor and bno055 shell commands to view the sensor data and control the bno055 sensor device.

- [Prerequisites](#)
- [Description of the Packages Needed for the Sample Application](#)
- [Step 1: Creating the Application Target](#)

- *Step 2: Creating the Bootloader Target*
- *Step 3: Building the Bootloader and Application Image*
- *Step 4: Creating an Application Image*
- *Step 5: Connecting the BNO055 Sensor to the nRF52-DK Board*
- *Step 6: Connecting the nRF52-DK Board to your Computer*
- *Step 7: Loading the Bootloader and the Application Image*
- *Step 8: Using a Terminal Emulator to Connect to the Application Console*
- *Step 9: Viewing the Registered Sensors and Sensor Data*
 - *Listing the Registered Sensors*
 - *Listing the Types that a Sensor Supports*
 - *Viewing Sensor Data Samples*
- *Step 10: Controlling and Viewing Sensor Device Hardware and Sensor Data*
- *Next Steps*

Prerequisites

- Meet the prerequisites listed in [Sensor Tutorials](#)
- Have a Nordic nRF52-DK board.
- Have an [Adafruit BNO055](#) sensor.
- Have a [serial port setup](#)
- Install the [Segger JLINK](#) software and documentation pack.

Description of the Packages Needed for the Sample Application

The sensors_test application includes all the packages, and sets the syscfg settings to values, that are required to enable the full set of sensor framework features. This tutorial uses a subset of the sensors_test application functionality because the objective of the tutorial is to show you how to quickly bring up the sensors_test application and use the `sensor` and `bno055` shell commands to view the sensor data from the BNO055 sensor. The instructions in this tutorial show the syscfg settings that must be enabled in the sensors_test application to demonstrate the examples shown. The instructions do not explicitly exclude the packages or change the syscfg setting values to disable the functionality that is not used in the sensors_test application.

For your reference, we describe the packages and the setting values that enable the application functionality that this tutorial demonstrates:

- **hw/sensor:** The sensor framework package. This package defines the `SENSOR_CLI` setting that specifies whether the `sensor` shell command is enabled. This setting is enabled by default.
- **hw/sensor/creator:** The sensor creator package. This package supports off-board sensor devices. This package creates the os devices in the kernel for the sensors and configures the sensor devices with default values. It defines a syscfg setting for each sensor device and uses the naming convention `<SENSORNAME>_OFB`. For example, the syscfg setting for the BNO055 sensor is `BNO055_OFB`. The `<SENSORNAME>_OFB` setting specifies whether the sensor named `SENSORNAME` is enabled. The setting is disabled by default. This package includes the

sensor device driver package `hw/drivers/sensors/<sensornname>` and creates and configures a sensor named `SENSORNAME` when the `SENSORNAME_OFB` setting is enabled by the application.

- **hw/drivers/sensors/bno055:** The driver package for the BNO055 sensor. The creator package adds this package as a package dependency when the `BNO055_OFB` setting is enabled. The driver package defines the `BNO055_CLI` setting that specifies whether the `bno055` shell command is enabled. This setting is disabled by default and is enabled by the application. The package also exports the `bno055_shell_init()` function that an application calls to initialize the driver shell support.

Note: All sensor driver packages that support a sensor shell command define a syscfg setting to specify whether the shell command is enabled. They also export a shell initialization function that an application must call. The naming convention is `<SENSORNAME>_CLI` for the syscfg setting and `<sensornname>_shell_init()` for the initialization function.

- **sys/shell** and **sys/console/full:** The shell and console packages for shell support over the console. The `SHELL_TASK` setting needs to be set to enable the shell support in the package. The `sensors_test` application enables this setting by default.

Step 1: Creating the Application Target

In this step, you create a target for the `sensors_test` application that enables the BNO055 off-board sensor.

To add the BNO055 sensor support, you create the application target with the following syscfg settings enabled:

- `I2C_0`: Enables the I2C interface 0 in the nRF52 BSP HAL setting.
- `BNO055_OFB`: Enables support for the BNO055 sensor in the sensor creator package (`hw/sensor/creator`). When this setting is enabled, the creator package performs the following:
 - Includes the BNO055 driver package (`hw/drivers/sensors/bno055`) as a package dependency.
 - Creates an os device for the sensor in the Mynewt kernel.
 - Configures the sensor device with default values.
- `BNO055_CLI`: Enables the `bno055` shell command in the `bno055` device driver package. The `sensors_test` application also uses this setting to conditionally include the call to the `bno055_shell_init()` function to initialize the shell support in the driver.

Note: This tutorial uses the `sensor` and the `bno055` shell commands. The `SENSOR_CLI` setting, that specifies whether the `sensor` shell command is enabled, is enabled by default.

1. Run the `newt target create` command, from your project base directory, to create the target. We name the target `nrf52_bno055_test`:

```
$ newt target create nrf52_bno055_test
Target targets/nrf52_bno055_test successfully created
$
```

2. Run the `newt target set` command to set the app, bsp, and build_profile variables for the target:

```
$ newt target set nrf52_bno055_test app=@apache-mynewt-core/apps/sensors_test_
→bsp=@apache-mynewt-core/hw/bsp/nrf52dk build_profile=debug
Target targets/nrf52_bno055_test successfully set target.app to @apache-mynewt-core/apps/
→sensors_test
Target targets/nrf52_bno055_test successfully set target.bsp to @apache-mynewt-core/hw/
→bsp/nrf52dk
Target targets/nrf52_bno055_test successfully set target.build_profile to debug
```

(continues on next page)

(continued from previous page)

\$

- Run the `newt target set` command to enable the I2C_0, BNO055_OFB, and BBN0055_CLI syscfg settings:

```
$ newt target set nrf52_bno055_test syscfg=BN0055_OFB=1:I2C_0=1:BN0055_CLI=1
Target targets/nrf52_bno055_test successfully set target.syscfg to BN0055_OFB=1:I2C_
↪_0=1:BN0055_CLI=1
$
```

Step 2: Creating the Bootloader Target

Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `nrf52_boot`:

```
$ newt target create nrf52_boot
Target targets/nrf52_boot successfully created
$ newt target set nrf52_boot app=@mcuboot/boot/mynewt bsp=@apache-mynewt-core/hw/bsp/
↪_nrf52dk build_profile=optimized
Target targets/nrf52_boot successfully set target.app to @mcuboot/boot/mynewt
Target targets/nrf52_boot successfully set target.bsp to @apache-mynewt-core/hw/bsp/
↪_nrf52dk
Target targets/nrf52_boot successfully set target.build_profile to optimized
$
```

Step 3: Building the Bootloader and Application Image

- Run the `newt build nrf52_boot` command to build the bootloader:

```
$ newt build nrf52_boot
Building target targets/nrf52_boot
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/loader.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/mcuboot/boot/mynewt/src/main.c

...
Archiving sys_mfg.a
Archiving sys_sysinit.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/nrf52_boot/app/boot/mynewt/mynewt.elf
Target successfully built: targets/nrf52_boot
```

- Run the `newt build nrf52_bno055_test` command to build the sensors_test application:

```
$ newt build nrf52_bno055_test
Building target targets/nrf52_bno055_test
```

(continues on next page)

(continued from previous page)

```

Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/mcuboot/boot/bootutil/src/image_validate.c
Compiling repos/mcuboot/boot/bootutil/src/bootutil_misc.c
Compiling repos/apache-mynewt-core/apps/sensors_test/src/misc.c
Compiling repos/apache-mynewt-core/apps/sensors_test/src/gatt_svr.c
Compiling repos/apache-mynewt-core/apps/sensors_test/src/main.c

...
Compiling repos/apache-mynewt-core/hw/drivers/sensors/bno055/src/bno055.c
Compiling repos/apache-mynewt-core/hw/drivers/sensors/bno055/src/bno055_shell.c

...
Compiling repos/apache-mynewt-core/hw/sensor/src/sensor.c
Compiling repos/apache-mynewt-core/hw/sensor/src/sensor_oic.c
Compiling repos/apache-mynewt-core/hw/sensor/src/sensor_shell.c
Compiling repos/apache-mynewt-core/hw/sensor/creator/src/sensor_creator.c

...
Archiving util_mem.a
Archiving util_parse.a
Linking ~/dev/myproj/bin/targets/nrf52_bno055_test/app/apps/sensors_test/sensors_test.elf
Target successfully built: targets/nrf52_bno055_test

```

Step 4: Creating an Application Image

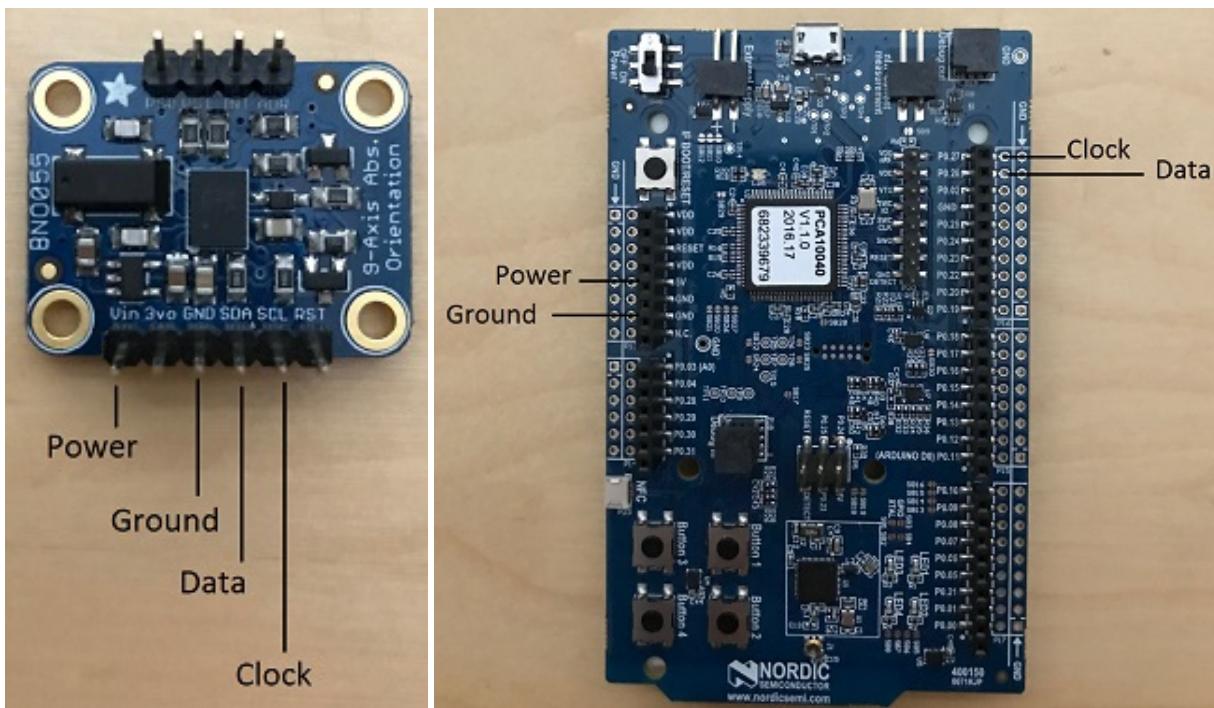
Run the `newt create-image` command to create an image file. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image nrf52_bno055_test 1.0.0
App image successfully generated: ~/dev/myproj/bin/targets/nrf52_bno055_test/app/apps/
→sensors_test/sensors_test.img
```

Step 5: Connecting the BNO055 Sensor to the nRF52-DK Board

Connect the pins from the BNO055 sensor to the nRF52-DK board as specified in the following table:

Lines	BNO055 Pin	nRF52-DK Pin
Power	Vin	5V
Clock	SCL	P0.27
Data	SDA	P0.26
Ground	GND	GND



Step 6: Connecting the nRF52-DK Board to your Computer

1. Set up two connections between your computer and the nRF52-DK board:
 - A serial connection to communicate with the sensors_test application and view the sensor data and hardware information via the Mynewt shell.

You can reference the [Serial Port Setup](#) tutorial for more information on setting up a serial communication.

 - A connection from your computer to the micro-USB port on the nRF52-DK board to power the board and to load the bootloader and application image.
2. Turn the power on the board to ON. You should see the green LED light up on the board.

Step 7: Loading the Bootloader and the Application Image

1. Run the `newt load nrf52_boot` command to load the bootloader onto the board:

```
$ newt load nrf52_boot
Loading bootloader
$
```

2. Run the `newt load nrf52_bno055_test` command to load the application image on to the board:

```
$ newt load nrf52_bno055_test
Loading app image into slot 1
$
```

3. Power the nRF52-DK board OFF and ON.

Step 8: Using a Terminal Emulator to Connect to the Application Console

Start up a terminal emulator to connect the sensors_test application console. You can use one of the terminal emulators listed below or one of your choice:

- On Mac OS and Linux platforms, you can run `minicom -D /dev/tty.usbserial-<port> -b 115200` to connect to the console of your app. Note that on Linux, the format of the port name is `/dev/ttyUSB<N>`, where N is a number.
- On Windows, you can use a terminal application such as PuTTY to connect to the device.

If you located your port from a MinGW terminal, the port name format is `/dev/ttyS<N>`, where N is a number. You must map the port name to a Windows COM port: `/dev/ttyS<N>` maps to `COM<N+1>`. For example, `/dev/ttyS2` maps to `COM3`.

You can also use the Windows Device Manager to locate the COM port.

We use minicom for this tutorial. After minicom connects, enter <return> to ensure the shell is running. You should see the `compat>` prompt:

```
Welcome to minicom 2.7.1

OPTIONS:
Compiled on May 17 2017, 15:29:14.
Port /dev/tty.usbserial, 13:55:21

Press Meta-Z for help on special keys

010674 compat>
```

Step 9: Viewing the Registered Sensors and Sensor Data

The sensor framework package implements the `sensor` shell command. This command allows you to:

- List all the registered sensor devices.
- View the sensor types that a registered sensor device supports.
- Read sensor data samples.

To view the command syntax, enter `sensor`

```
002340 Possible commands for sensor are:
002341   list
002341     list of sensors registered
002342   read <sensor_name> <type> [-n nsamples] [-i poll_itvl(ms)] [-d poll_du]
002344     read <no_of_samples> from sensor<sensor_name> of type:<type> at pr
002347     at <poll_interval> rate for <poll_duration>
002348   type <sensor_name>
002349     types supported by registered sensor
002350 compat>
```

Listing the Registered Sensors

You use the `sensor list` command to list all the registered sensor devices:

```
031798 compat> sensor list
129441 sensor dev = bno055_0, configured type = 0x1 0x2 0x4 0x200 0x1000 0x2000
129444 compat>
```

The output shows one sensor, **bno055_0**, registered, and the configured types for the sensor. A configure type is a subset of the types that a sensor supports.

Listing the Types that a Sensor Supports

You use the `sensor type` command to list the types that a sensor supports:

```
031822 compat> sensor type bno055_0
033156 sensor dev = bno055_0,
type =
033157     accelerometer: 0x1
033157     magnetic field: 0x2
033158     gyroscope: 0x4
033159     temperature: 0x10
033160     vector: 0x200
033160     accel: 0x1000
033161     gravity: 0x2000
033162     euler: 0x4000
```

Viewing Sensor Data Samples

You use the `sensor read` command to read data samples for a configured type. You can specify the number of samples to read, a poll interval, and a poll duration. You can only view sensor data for the sensor types that a sensor device is configured for.

Example 1: Read 5 samples of accelerometer data from the **bno055_0** sensor:

```
033163 compat> sensor read bno055_0 0x1 -n 5
042974 ts: [ secs: 335 usecs: 745441 cputime: 336218225 ]
042976 x = -0.519999968 y = -7.289999968 z = 6.489999776
042978 ts: [ secs: 335 usecs: 771216 cputime: 336244000 ]
042979 x = -0.529999968 y = -7.360000128 z = 6.559999936
042981 ts: [ secs: 335 usecs: 794640 cputime: 336267424 ]
042982 x = -0.529999968 y = -7.340000160 z = 6.480000032
042983 ts: [ secs: 335 usecs: 810795 cputime: 336283579 ]
042984 x = -0.519999968 y = -7.300000192 z = 6.530000224
042986 ts: [ secs: 335 usecs: 833703 cputime: 336306487 ]
042987 x = -0.510000000 y = -7.309999936 z = 6.380000128
```

Each sample contains two lines of output. The first line is the time when the sample is read. The second line is the sample data. For the example output:

These two lines are for the first sample:

```
042974 ts: [ secs: 335 usecs: 745441 cputime: 336218225 ]  
042976 x = -0.519999968 y = -7.289999968 z = 6.489999776
```

These two lines are for the last sample:

```
042986 ts: [ secs: 335 usecs: 833703 cputime: 336306487 ]  
042987 x = -0.510000000 y = -7.309999936 z = 6.380000128
```

Example 2: Read the vector data at 20 ms poll interval. You can enter **ctrl-c**, **q <return>**, or **Q <return>** to stop the polling.

```
002350 compat> sensor read bno055_0 0x200 -i 20  
019271 ts: [ secs: 150 usecs: 560056 cputime: 151019584 ]  
019272 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019274 ts: [ secs: 150 usecs: 580598 cputime: 151040126 ]  
019275 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019277 ts: [ secs: 150 usecs: 604036 cputime: 151063564 ]  
019278 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019280 ts: [ secs: 150 usecs: 627474 cputime: 151087002 ]  
019281 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019283 ts: [ secs: 150 usecs: 650912 cputime: 151110440 ]  
019284 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019286 ts: [ secs: 150 usecs: 674350 cputime: 151133878 ]  
019287 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019289 ts: [ secs: 150 usecs: 697788 cputime: 151157316 ]  
019290 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019292 ts: [ secs: 150 usecs: 721225 cputime: 151180753 ]  
019293 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019295 ts: [ secs: 150 usecs: 744663 cputime: 151204191 ]  
019296 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019298 ts: [ secs: 150 usecs: 768101 cputime: 151227629 ]  
019299 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984  
019301 ts: [ secs: 150 usecs: 791539 cputime: 151251067 ]  
019302 x = 3.442626944 y = 0.026977540 z = 3.993286144 w = 0.829833984
```

Step 10: Controlling and Viewing Sensor Device Hardware and Sensor Data

The BNO055 device driver implements the **bno055** shell command that allows you to:

- Read sensor data samples for all the sensor types that the device supports.
- Note:** The **sensor** shell command discussed previously only reads sensor data for configured sensor types.
- Query the chip id, sensor revisions, content of registers, sensor offsets.
 - Reset the device.
 - Change the power mode.
 - Change the operation mode.

Enter **bno055** to see the command syntax:

```
711258 bno055 cmd [flags...]  
711259 cmd:
```

(continues on next page)

(continued from previous page)

```

711259 r      [n_samples] [ 0-acc          | 1 -mag       | 2 -gyro     | 4 -tem|
                  9-quat        | 26-linearacc | 27-gravity | 28-eul]

711264 mode   [0-config    | 1-acc          | 2 -mag       | 3 -gyro     | 4 -acc|
                  5-acccgyro  | 6-maggyro    | 7 -amg      | 8 -imuplus  | 9 -com|
                  9-m4g        | 11-NDOF_FMC_OFF | 12-NDOF   ]

711269 chip_id
711270 rev
711270 reset
711270 pmode [0-normal   | 1-lowpower    | 2-suspend]
711272 sensor_offsets
711272 dumpreg [addr]

```

*** Example 3: *** Query the device chip id:

```

711273 compat> bno055 chip_id
769056 0xA0

```

Example 4: View the sensor revisions:

```

827472 compat> bno055 rev
862354 accel_rev:0xFB
mag_rev:0x32
gyro_rev:0x0F
sw_rev:0x311
bl_rev:0x15

```

Next Steps

Now that you have successfully enabled an application to communicate with a sensor, We recommend that you:

- Experiment with other sensor and bno055 shell commands in this tutorial to view other types of sensor data.
- Change the default configuration values for the sensor. See the [Changing the Default Configuration for a Sensor tutorial](#)
- Try a different off-board sensor. You can follow most of the procedures in this tutorial to enable other sensors in the sensors_test application. The syscfg.yml file for the hw/sensor/creator/ package specifies the off-board sensors that Mynewt currently supports. You will need to:
 - Enable the <SENSORNAME>_OFB setting to include the sensor driver package and to create and initialize the sensor device.
 - Enable the correct interface in the nRF52 BSP to communicate with the sensor device.
 - Enable the sensor device driver shell command if the driver supports the shell. You can check the syscfg.yml file for the sensor device driver package in the hw/drivers/sensor/<sensorname> directory.
- Try one of the other sensor tutorials listed in the [Sensor Tutorials Overview](#)

Changing the Default Configuration for a Sensor

This tutorial shows you how to change default configuration values for an off-board sensor. It continues with the example in the [Enabling an Off-Board Sensor in an Existing Application tutorial](#)

Note: You can also follow most of the instructions in this tutorial to change the default configuration for an onboard sensor. The difference is that the BSP, instead of the sensor creator package, creates and configures the onboard sensor devices in the `hal_bsp.c` file. You should check the BSP to determine whether the default configuration for a sensor meets your application requirements.

- *Prerequisites*
- *Overview on How to Initialize the Configuration Values for a Sensor*
- *Changing the Default Configuration*
 - *Step 1: Adding the Sensor Device Driver Header File*
 - *Step 2: Adding a New Configuration Function*
 - *Step 3: Changing the Default Configuration Settings*
 - *Step 4: Calling the Configuration Function From main()*
 - *Step 5: Building a New Application Image*
 - *Step 6: Loading the New Image and Rebooting the Device*
 - *Step 7: Verifying the Sensor is Configured with the New Values*
 - *Step 8: Verifying that the Accelerometer Data Samples Cannot be Read*

Prerequisites

Complete the tasks described in the [Enabling an Off-Board Sensor in an Existing Application tutorial](#)

Overview on How to Initialize the Configuration Values for a Sensor

The sensor creator package, `hw/sensor/creator`, creates, for each enabled sensor, an os device in the kernel for the sensor and initializes the sensor with its default configuration when the package is initialized. The steps to configure a sensor device are:

1. Open the os device for the sensor.
2. Initialize the sensor driver configuration data structure with default values.
3. Call the `<sensorname>_config()` function that the sensor device driver package exports.
4. Close the os device for the sensor.

For the BNO055 sensor device, the creator package calls the local `config_bno055_sensor()` function to configure the sensor. A code excerpt for this function is shown below:

```
static int  
config_bno055_sensor(void)  
{  
    int rc;
```

(continues on next page)

(continued from previous page)

```

struct os_dev *dev;
struct bno055_cfg bcfg;

dev = (struct os_dev *) os_dev_open("bno055_0", OS_TIMEOUT_NEVER, NULL);
assert(dev != NULL);

bcfg.bc_units = BNO055_ACC_UNIT_MS2 | BNO055_ANGRATE_UNIT_DPS |
    BNO055_EULER_UNIT_DEG | BNO055_TEMP_UNIT_DEGC |
    BNO055_DO_FORMAT_ANDROID;

bcfg.bc_opr_mode = BNO055_OPR_MODE_NDOF;
bcfg.bc_pwr_mode = BNO055_PWR_MODE_NORMAL;
bcfg.bc_acc_bw = BNO055_ACC_CFG_BW_125HZ;
bcfg.bc_acc_range = BNO055_ACC_CFG_RNG_16G;
bcfg.bc_mask = SENSOR_TYPE_ACCELEROMETER |
    SENSOR_TYPE_MAGNETIC_FIELD |
    SENSOR_TYPE_GYROSCOPE |
    SENSOR_TYPE_EULER |
    SENSOR_TYPE_GRAVITY |
    SENSOR_TYPE_LINEAR_ACCEL |
    SENSOR_TYPE_ROTATION_VECTOR;

rc = bno055_config((struct bno055 *) dev, &bcfg);

os_dev_close(dev);
return rc;
}

```

Changing the Default Configuration

To change the default configuration, you can directly edit the fields in the `config_bno055_sensor()` function in the `hw/sensor/creator/sensor_creator.c` file or add code to your application to reconfigure the sensor during application initialization.

This tutorial shows you how to add the code to the `apps/sensors_test/src/main.c` file to configure the sensor without the accelerometer sensor type. When you reconfigure a sensor in the application, you must initialize all the fields in the sensor configuration data structure even if you are not changing the default values.

Step 1: Adding the Sensor Device Driver Header File

Add the bno055 device driver header file:

```
#include <bno055/bno055.h>
```

Step 2: Adding a New Configuration Function

Add the `sensors_test_config_bno055()` function and copy the code from the `config_bno055_sensor()` function in the `hw/sensor/creator/sensor_creator.c` file to the body of the `sensors_test_config_bno055()` function. The content of the `sensors_test_config_bno055()` function should look like the example below:

```
static int
sensors_test_config_bno055(void)
{
    int rc;
    struct os_dev *dev;
    struct bno055_cfg bcfg;

    dev = (struct os_dev *) os_dev_open("bno055_0", OS_TIMEOUT_NEVER, NULL);
    assert(dev != NULL);

    bcfg.bc_units = BNO055_ACC_UNIT_MS2 | BNO055_ANGRATE_UNIT_DPS |
                    BNO055_EULER_UNIT_DEG | BNO055_TEMP_UNIT_DEGC |
                    BNO055_DO_FORMAT_ANDROID;

    bcfg.bc_opr_mode = BNO055_OPR_MODE_NDOF;
    bcfg.bc_pwr_mode = BNO055_PWR_MODE_NORMAL;
    bcfg.bc_acc_bw = BNO055_ACC_CFG_BW_125HZ;
    bcfg.bc_acc_range = BNO055_ACC_CFG_RNG_16G;
    bcfg.bc_use_ext_xtal = 1;
    bcfg.bc_mask = SENSOR_TYPE_ACCELEROMETER |
                   SENSOR_TYPE_MAGNETIC_FIELD |
                   SENSOR_TYPE_GYROSCOPE |
                   SENSOR_TYPE_EULER |
                   SENSOR_TYPE_GRAVITY |
                   SENSOR_TYPE_LINEAR_ACCEL |
                   SENSOR_TYPE_ROTATION_VECTOR;

    rc = bno055_config((struct bno055 *) dev, &bcfg);

    os_dev_close(dev);
    return rc;
}
```

Step 3: Changing the Default Configuration Settings

Delete the SENSOR_TYPE_ACCELEROMETER type from the bcfg.bc_mask initialization setting values:

```
static int
sensors_test_config_bno055(void)
{
    int rc
    ...

    /* Delete the SENSOR_TYPE_ACCELEROMETER from the mask */
    bcfg.bc_mask = SENSOR_TYPE_MAGNETIC_FIELD |
        SENSOR_TYPE_GYROSCOPE |
        SENSOR_TYPE_EULER |
        SENSOR_TYPE_GRAVITY |
        SENSOR_TYPE_LINEAR_ACCEL |
        SENSOR_TYPE_ROTATION_VECTOR;

    rc = bno055_config((struct bno055 *) dev, &bcfg);

    os_dev_close(dev);
    return rc;
}
```

Step 4: Calling the Configuration Function From main()

Add the **int** **rc** declaration and the call to the **sensors_test_config_bno055()** function in **main()**:

```
int
main(int argc, char **argv)
{
    ...

    /* Add rc for the return value from sensors_test_config_bno055() */
    int rc;

    ...

    /* Add call to sensors_test_config_bno055() and abort on error */
    rc = sensors_test_config_bno055();
    assert(rc == 0);

    /* log reboot */
    reboot_start(hal_reset_cause());

    /*
     * As the last thing, process events from default event queue.
     */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }

    return (0);
}
```

Step 5: Building a New Application Image

Run the `newt build nrf52_bno055_test` and the `newt create-image nrf52_bno055_test 2.0.0` commands to rebuild and create a new application image.

Step 6: Loading the New Image and Rebooting the Device

Run the `newt load nrf52_bno055_test` command and power the device OFF and On.

Step 7: Verifying the Sensor is Configured with the New Values

Start a terminal emulator, and run the `sensor list` command to verify the accelerometer (0x1) is not configured. The `configured type` listed for the sensor should not have the value `0x1`.

```
045930 compat> sensor list
046482 sensor dev = bno055_0, configured type = 0x2 0x4 0x200 0x1000 0x2000 0x4000
046484 compat>
```

Step 8: Verifying that the Accelerometer Data Samples Cannot be Read

Run the `sensor read` command to read data samples from the accelerometer to verify that the sensor cannot be read:

```
046484 compat> sensor read bno055_0 0x1 -n 5
092387 Cannot read sensor bno055_0
```

Developing an Application for an Onboard Sensor

This tutorial shows you how to develop a simple application for an onboard sensor. The Mynewt sensor framework enables you to easily and quickly develop Mynewt sensor applications. We assume that you have completed the [Enabling an Off-Board Sensor in an Existing Application Tutorial](#) and are familiar with the sensor framework and sensor shell command.

This tutorial shows you how to:

- Develop an application for the Nordic Thingy LIS2DH12 accelerometer onboard sensor with the sensor framework `sensor` shell command enabled to view sensor data.
- Extend the application to use the sensor framework API to read the sensor data and output the data to the Mynewt console.

- *Prerequisites*
- *Developing a Sensor Enabled Application with Shell Support*
 - *Step 1: Creating a New App Package*
 - *Step 2: Adding the Package Dependencies*
 - *Step 3: Using the Skeleton main.c File*
 - *Step 4: Creating the Target for the my_sensor_app Application*

- Step 5: Creating and Building the Bootloader Target
- Step 6: Connecting the Thingy to your Computer
- Step 7: Loading the Image and Connecting to the Console via RTT
- Step 8: Viewing the list of Sensors and Sensor Data
- Extending the Application to Use the Sensor API to Read Sensor Data
- Step 1: Modifying main.c to Add a Sensor Listener
- Step 2: Rebuilding the Application and Connecting to Console
- Step 3: Modifying main.c to Use sensor_read() Instead of a Listener
- Step 4: Rebuilding the Application and Connecting to Console

Prerequisites

- Meet the prerequisites listed in the [Sensor Tutorials Overview](#)
- Have a Nordic Thingy.
- Segger J-Link Debug Probe.
- J-Link 9 pin Cortex-M Adapter that allows JTAG, SWD and SWO connections between J-Link and Cortex M based target hardware systems.
- Install the [Segger JLINK Software](#) and documentation pack.
- Complete the [Enabling an Off-Board Sensor in an Existing Application Tutorial](#)

Developing a Sensor Enabled Application with Shell Support

We first develop a simple application with the LIS2DH12 onboard sensor on the Nordic Thingy and the sensor shell command enabled.

Step 1: Creating a New App Package

We name the new app package `my_sensor_app`. From your project base directory, run the `newt pkg new` command to create a new app package. This tutorial uses `~/dev/myproj` for the project.

```
$ cd ~/dev/myproj
$ newt pkg new -t app apps/my_sensor_app
Download package template for package type app.
Package successfully installed into ~/dev/myproj/apps/my_sensor_app
```

The newt tool creates a skeleton `my_sensor_app` package directory in the `~/dev/myproj/apps/` directory. Go to the `my_sensor_app` directory to update the package `pkg.yml` and source files.

```
$ cd apps/my_sensor_app
```

Step 2: Adding the Package Dependencies

The my_sensor_app package requires the sensor framework, hw/sensor, package as a package dependency. Note that the BSP creates the sensor devices for the onboard sensors, so the hw/sensor/creator package that creates off-board sensor is not needed.

Add the highlighted line to the pkg.yml file to add the hw/sensor package as package dependency:

```
pkg.deps:  
  - "@apache-mynewt-core/kernel/os"  
  - "@apache-mynewt-core/sys/console/full"  
  - "@apache-mynewt-core/sys/log/full"  
  - "@apache-mynewt-core/sys/stats/full"  
  - "@apache-mynewt-core/hw/sensor"
```

Step 3: Using the Skeleton main.c File

The newt tool creates a skeleton main.c file in the my_sensor_app/src directory. The skeleton main() code shown is all you need to build an application that only uses the sensor shell command to read sensor data. You do not need to make any changes to the file. The sensor framework implements the sensor shell command and the shell package processes shell command events from the OS default event queue.

```
int  
main(int argc, char **argv)  
{  
    /* Perform some extra setup if we're running in the simulator. */  
#ifdef ARCH_sim  
    mcu_sim_parse_args(argc, argv);  
#endif  
  
    /* Initialize all packages. */  
    sysinit();  
  
    /* As the last thing, process events from default event queue. */  
    while (1) {  
        os_eventq_run(os_eventq_dflt_get());  
    }  
  
    return 0;  
}
```

Step 4: Creating the Target for the my_sensor_app Application

You create a target for the my_sensor_app to run on the Nordic Thingy. The following syscfg settings must be set:

- I2C_0=1 : Enables the I2C interface 0 for the nRF52 Thingy BSP HAL setting to communicate with the onboard sensor.
- LIS2DH12_ONB=1: Enables the lis2dh12 onboard sensor support in the nRF52 Thingy BSP.

A BSP with onboard sensors defines a syscfg setting for each onboard sensor it supports and uses the naming convention <SENSORNAME>_ONB. The <SENSORNAME>_ONB setting specifies whether the sensor named SENSORNAME is enabled. The setting is disabled by default. The BSP includes the sensor device driver package

`hw/drivers/sensors/<sensorname>` and creates and configures the onboard sensor named SENSORNAME when the `<SENSORNAME>_ONB` setting is enabled by the application.

- `SHELL_TASK=1`: Enables the shell task for the shell command support. Note that the `hw/sensor` package enables the `SENSOR_CLI` setting by default.
- `SENSOR_OIC=0`: Disables the OIC sensor server support in the sensor framework.
- `CONSOLE_RTT=1`: Enables console communication via the SEGGER RTT. The nRF52 Thingy does not have a UART so we use the RTT for the console.
- `CONSOLE_UART=0`: Disables the console communication via a UART.

Note: The lis2dh12 sensor device driver package, `/hw/driver/sensors/lis2dh12`, currently does not support a shell command to view information on the device.

1. Run the following newt commands to create the target and set the application and BSP.

```
$ newt target create thingy_my_sensor
Target targets/thingy_my_sensor successfully created
$ newt target set thingy_my_sensor bsp=@apache-mynewt-core/hw/bsp/nrf52-thingy
Target targets/thingy_my_sensor successfully set target.bsp to @apache-mynewt-core/hw/
↪ bsp/nrf52-thingy
$ newt target set thingy_my_sensor app=apps/my_sensor_app
Target targets/thingy_my_sensor successfully set target.app to apps/my_sensor_app
$ newt target set thingy_my_sensor build_profile=debug
Target targets/thingy_my_sensor successfully set target.build_profile to debug
```

2. Run the following `newt target set` command to set the syscfg settings:

```
$ newt target set thingy_my_sensor syscfg=I2C_0=1:LIS2DH12_ONB=1:SHELL_TASK=1:CONSOLE_
↪ RTT=1:CONSOLE_UART=0:SENSOR_OIC=0
Target targets/thingy_my_sensor successfully set target.syscfg to I2C_0=1:LIS2DH12_
↪ ONB=1:SHELL_TASK=1:CONSOLE_RTT=1:CONSOLE_UART=0:SENSOR_OIC=0
```

Step 5: Creating and Building the Bootloader Target

Create a target for the bootloader for the nRF52 Thingy. We name the target `thingy_boot`.

1. Run the following `newt target` commands to create the target:

```
$ newt target create thingy_boot
Target targets/thingy_boot successfully created
$ newt target set thingy_boot bsp=@apache-mynewt-core/hw/bsp/nrf52-thingy
Target targets/thingy_boot successfully set target.bsp to @apache-mynewt-core/hw/bsp/
↪ nrf52-thingy
$ newt target set thingy_boot app=@mcuboot/boot/mynewt
Target targets/thingy_boot successfully set target.app to @mcuboot/boot/mynewt
$ newt target set thingy_boot build_profile=optimized
Target targets/thingy_boot successfully set target.build_profile to optimized
```

2. Run the `newt build` command to build the bootloader target:

```
$ newt build thingy_boot
Building target targets/thingy_boot
```

(continues on next page)

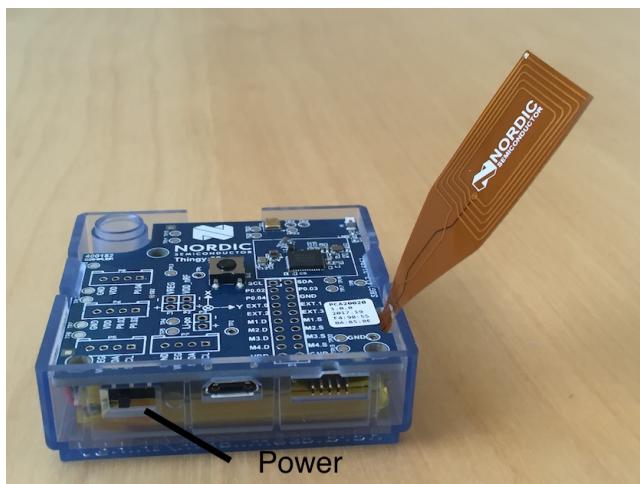
(continued from previous page)

```
...
Archiving thingy_boot-sysinit-app.a
Archiving util_mem.a
Linking ~/dev/myproj/bin/targets/thingy_boot/app/boot/mynewt/mynewt.elf
Target successfully built: targets/thingy_boot
```

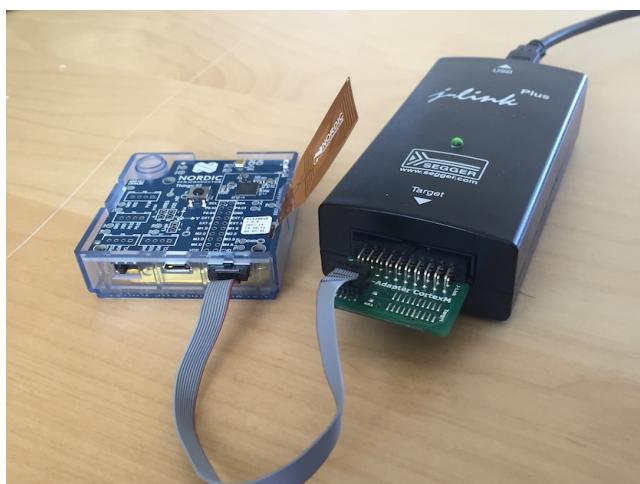
Step 6: Connecting the Thingy to your Computer

Perform the following steps to connect the Thingy to your computer:

1. Move the power switch to the left to power ON the Thingy:



2. Connect the debug probe to the JTAG port on the board using the Jlink 9-pin adapter and cable, and connect the probe to your computer.



Step 7: Loading the Image and Connecting to the Console via RTT

To run the application, you need to load the bootloader on to the device, load the application image, and start a GDB debug process for RTT to attach to.

- Run the `newt load` command to load the bootloader:

```
$ newt load thingy_boot
Loading bootloader
```

- Run the `newt run` command to build and create an image for the `my_sensor_app`, load the image, and start a GDB process to debug the application:

```
$ newt run thingy_my_sensor 1.0.0
Assembling repos/apache-mynewt-core/hw/bsp/nrf52-thingy/src/arch/cortex_m4/gcc_startup_
↳ nrf52_splits.s
Compiling repos/apache-mynewt-core/hw/cmsis-core/src/cmsis_nvic.c
Assembling repos/apache-mynewt-core/hw/bsp/nrf52-thingy/src/arch/cortex_m4/gcc_startup_
↳ nrf52.s
Compiling repos/apache-mynewt-core/encoding/base64/src/hex.c
Compiling apps/my_sensor_app/src/main.c

...
Archiving thingy_my_sensor-sysinit-app.a
Archiving time_datetime.a
Archiving util_cbmem.a
Archiving util_crc.a
Archiving util_mem.a
Archiving util_parse.a
Linking ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_sensor_app.
↳ elf
App image successfully generated: ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_
↳ sensor_app/my_sensor_app.img
Loading app image into slot 1
[~/dev/myproj/repos/apache-mynewt-core/hw/bsp/nrf52-thingy/nrf52-thingy_debug.sh ~/dev/
↳ myproj/repos/apache-mynewt-core/hw/bsp/nrf52-thingy ~/dev/myproj/bin/targets/thingy_my_
↳ sensor/app/apps/my_sensor_app/my_sensor_app]
Debugging ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_sensor_app.
↳ elf
GNU gdb (GNU Tools for ARM Embedded Processors) 7.8.0.20150604-cvs
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-apple-darwin10 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
```

(continues on next page)

(continued from previous page)

```
Reading symbols from ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_
→sensor_app.elf...done.
os_tick_idle (ticks=24)
    at repos/apache-mynewt-core/hw/mcu/nordic/nrf52xxx/src/hal_os_tick.c:204
204      if (ticks > 0) {
Resetting target
0x000000dc in ?? ()
(gdb)
```

3. Enter `c <return>` in the (gdb) prompt to continue.
4. Run the following telnet command to connect to the Mynewt console via RTT and enter <return> to get the shell prompt after you are connected.

```
$ telnet localhost 19021
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SEGGER J-Link V6.14h - Real time terminal output
SEGGER J-Link ARM V10.0, SN=600000268
Process: JLinkGDBServer

011468 compat>
```

Step 8: Viewing the list of Sensors and Sensor Data

1. Enter `sensor list` to see the sensors that are registered with the sensor manager. You should see the `lis2dh12_0` sensor. This sensor is only configured for the accelerometer (0x1).

```
011468 compat> sensor list
sensor list
029706 sensor dev = lis2dh12_0, configured type = 0x1
029707 compat>
```

2. Enter the `sensor read` command to read some data samples from the accelerometer:

```
029707 compat> sensor read lis2dh12_0 0x1 -n 5
sensor read lis2dh12_0 0x1 -n 5
042537 ts: [ secs: 331 usecs: 102682 cputime: 331436945 ]
042537 x = 9.806650176 y = 58.839900992 z = -9894.910156
042537 ts: [ secs: 331 usecs: 104832 cputime: 331439095 ]
042537 x = 19.613300352 y = 98.066497804 z = -9924.330078
042537 ts: [ secs: 331 usecs: 106988 cputime: 331441251 ]
042537 x = 9.806650176 y = 49.033248902 z = -9904.716796
042538 ts: [ secs: 331 usecs: 109137 cputime: 331443400 ]
042538 x = 9.806650176 y = 29.419950496 z = -9904.716796
042538 ts: [ secs: 331 usecs: 111288 cputime: 331445551 ]
042538 x = 58.839900992 y = 0.000000000 z = -9816.457031
042538 compat>
```

Extending the Application to Use the Sensor API to Read Sensor Data

As this tutorial demonstrates so far, the Mynewt sensor framework enables you to easily and quickly develop an application with a sensor and view the sensor data from the `sensor shell` command. We now extend the application to use the sensor API to read the sensor data.

There are two sensor functions that you can use to read data from a sensor device:

- `sensor_register_listener()`: This function allows you to register a listener for a sensor device. You specify a bit mask of the types of sensor data to listen for and a callback to call when data is read from the sensor device. The listener callback is called whenever the `sensor_read()` function reads data for a sensor type from a sensor device that the listener is listening for.

The sensor framework supports polling of sensor devices. For a sensor device that has a polling rate configured, the sensor framework poller reads sensor data for all the configured sensor types from the sensor device at each polling interval and calls the registered listener callbacks with the sensor data.

- `sensor_read()`: This function reads sensor data from a sensor device and calls the specified user callback to receive the sensor data. You specify a bit mask of the types of sensor data to read from a sensor device and a callback. This callback is called for each sensor type you specify to read.

We first extend the application to a register a sensor listener to demonstrate how to use the sensor framework polling support. We then extend the application to use the `sensor_read()` function instead of a listener. An application may not need to poll sensors. For example, an application that processes remote requests for sensor data might only read the sensor data when it receives a request.

Step 1: Modifying main.c to Add a Sensor Listener

Add the following code to the `my_sensor_app/src/main.c` file:

1. Add the highlighted include files:

```
#include "sysinit/sysinit.h"
#include "os/os.h"

#include <defs/error.h>
#include <sensor/sensor.h>
#include <sensor/accel.h>
#include <console/console.h>
```

2. Add the `struct sensor *my_sensor`. This is the handle for the sensor that the sensor API uses to perform operations on the sensor. We set this variable when we lookup the sensor.

```
static struct sensor *my_sensor;
```

3. Declare and initialize a sensor listener. You specify a bit mask for the sensor types to listen for, the callback function, and an opaque argument to pass to the callback. You initialize the type to `SENSOR_TYPE_ACCELEROMETER`, the listener callback to the `read_accelerometer()` function, and the callback opaque argument to the `LISTENER_CB` value.

Note: We define `LISTENER_CB` and `READ_CB` values because we also use the `read_accelerometer()` function as the callback for the `sensor_read()` function later in the tutorial. The `LISTENER_CB` or the `READ_CB` value is passed to the `read_accelerometer()` function to indicate whether it is invoked as a listener or a `sensor_read()` callback.

```
#define LISTENER_CB 1
#define READ_CB 2

static int read_accelerometer(struct sensor* sensor, void *arg, void *databuf, sensor_
→type_t type);

static struct sensor_listener listener = {
    .sl_sensor_type = SENSOR_TYPE_ACCELEROMETER,
    .sl_func = read_accelerometer,
    .sl_arg = (void *)LISTENER_CB,
};
```

4. Add the code for the `read_accelerometer()` function. The sensor data is stored in the `databuf` and `type` specifies the type of sensor data.

```
static int
read_accelerometer(struct sensor* sensor, void *arg, void *databuf, sensor_type_t type)
{

    char tmpstr[13];
    struct sensor_accel_data *sad;

    if (!databuf) {
        return SYS_EINVAL;
    }

    sad = (struct sensor_accel_data *)databuf;

    if (!sad->sad_x_is_valid ||
        !sad->sad_y_is_valid ||
        !sad->sad_z_is_valid) {

        return SYS_EINVAL;
    }

    console_printf("%s: [ secs: %ld usecs: %d cputime: %u ]\n",
        ((int)arg == LISTENER_CB) ? "LISTENER_CB" : "READ_CB",
        (long int)sensor->s_sts.st_ostv.tv_sec,
        (int)sensor->s_sts.st_ostv.tv_usec,
        (unsigned int)sensor->s_sts.st_cputime);

    console_printf("x = %s ", sensor_ftostr(sad->sad_x, tmpstr, 13));
    console_printf("y = %s ", sensor_ftostr(sad->sad_y, tmpstr, 13));
    console_printf("z = %s\n\n", sensor_ftostr(sad->sad_z, tmpstr, 13));
    return 0;
}
```

5. Set the poll rate for the sensor to two seconds. The `sensor_set_poll_rate_ms()` function sets the poll rate for a named sensor.

Note: You set the poll rate for a sensor programmatically and must set the poll rate to a non zero value in order for the sensor manager to poll the sensor. You may set a different poll rate for each sensor. The sensor framework also defines a `SENSOR_MGR_WAKEUP_RATE` syscfg setting that specifies the default rate that the sensor manager polls. The sensor manager uses the poll rate for a sensor if a sensor is configured to poll more frequently than the `SENSOR_MGR_WAKEUP_RATE`

setting value.

```
#define MY_SENSOR_DEVICE "lis2dh12_0"
#define MY_SENSOR_POLL_TIME 2000

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

    ...

    /* Initialize all packages. */
    sysinit();

    rc = sensor_set_poll_rate_ms(MY_SENSOR_DEVICE, MY_SENSOR_POLL_TIME);
    assert(rc == 0);

    /* As the last thing, process events from default event queue. */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }

    return 0;
}
```

6. Look up the sensor by name to get the handle for the sensor and register a listener for the sensor.

```
int
main(int argc, char **argv)
{
    ...

    rc = sensor_set_poll_rate_ms(MY_SENSOR_DEVICE, MY_SENSOR_POLL_TIME);
    assert(rc == 0);

    my_sensor = sensor_mgr_find_next_bydevname(MY_SENSOR_DEVICE, NULL);
    assert(my_sensor != NULL);
    rc = sensor_register_listener(my_sensor, &listener);
    assert(rc == 0);

    /* As the last thing, process events from default event queue. */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }

    return 0;
}
```

Step 2: Rebuilding the Application and Connecting to Console

1. Run the `newt run` command to rebuild the application, create a new image, load the image, and start a GDB process:

```
$ newt run thingy_my_sensor 2.0.0
Compiling apps/my_sensor_app/src/main.c
Archiving apps_my_sensor_app.a
Linking ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_sensor_app.elf
App image successfully generated: ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_sensor_app.img
Loading app image into slot 1
[~/dev/myproj/repos/apache-mynewt-core/hw/bsp/nrf52-thingy/nrf52-thingy_debug.sh ~/dev/myproj/repos/apache-mynewt-core/hw/bsp/nrf52-thingy ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_sensor_app]
Debugging ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_sensor_app.elf
GNU gdb (GNU Tools for ARM Embedded Processors) 7.8.0.20150604-cvs

...
Reading symbols from ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_sensor_app.elf...done.
os_tick_idle (ticks=12)
    at repos/apache-mynewt-core/hw/mcu/nordic/nrf52xxx/src/hal_os_tick.c:204
204      if (ticks > 0) {
Resetting target
0x0000000dc in ?? ()
(gdb) c
Continuing.
```

2. Connect to the console via RTT:

```
$ telnet localhost 19021
Connected to localhost.
Escape character is '^]'.
SEGGER J-Link V6.14h - Real time terminal output
J-Link OB-SAM3U128-V2-NordicSemi compiled Mar 2 2017 12:22:13 V1.0, SN=682562963
Process: JLinkGDBServer
000003 LISTENER_CB: [ secs: 0 usecs: 23407 cputime: 331783 ]
000003 x = 117.67980192 y = -19.61330035 z = -9885.103515

000259 LISTENER_CB: [ secs: 2 usecs: 21190 cputime: 2327645 ]
000259 x = 117.67980192 y = -9.806650176 z = -9914.523437

000515 LISTENER_CB: [ secs: 4 usecs: 17032 cputime: 4323487 ]
000515 x = 78.453201280 y = 0.000000000 z = -9924.330078

000771 LISTENER_CB: [ secs: 6 usecs: 13131 cputime: 6319586 ]
000771 x = 117.67980192 y = -19.61330035 z = -9914.523437

001027 LISTENER_CB: [ secs: 8 usecs: 8810 cputime: 8315265 ]
```

(continues on next page)

(continued from previous page)

```
001027 x = 127.48645020 y = 0.000000000 z = -9924.330078
001283 LISTENER_CB: [ secs: 10 usecs: 4964 cputime: 10311419 ]
001283 x = 58.839900992 y = -9.806650176 z = -9885.103515
```

You should see the accelerometer sensor data output from the listener callback.

Step 3: Modifying main.c to Use `sensor_read()` Instead of a Listener

Lets extend the application to use the `sensor_read()` function instead of a listener. We setup an OS callout to call the `sensor_read()` function for illustration purposes. A real application will most likely read the sensor data when it gets a request or some other event.

1. Add an OS callout and initialize an OS timer to fire every 5 seconds. The timer callback calls the `sensor_read()` function to read the sensor data. The `read_accelerometer()` callback is called when the sensor data is read. The `READ_CB` value is passed to the `read_accelerometer()` function and indicates that the callback is from the `sensor_read()` function and not from the listener.

```
/*
 * Event callback function for timer events. The callback reads the sensor data
 */

#define READ_SENSOR_INTERVAL (5 * OS_TICKS_PER_SEC)

static struct os_callout sensor_callout;

static void
timer_ev_cb(struct os_event *ev)
{
    assert(ev != NULL);

    /*
     * Read the accelerometer sensor. Pass the READ_CB value for the callback opaque
     * arg to indicate that it is the sensor_read() callback.
     */
    sensor_read(my_sensor, SENSOR_TYPE_ACCELEROMETER, read_accelerometer,
               (void *)READ_CB, OS_TIMEOUT_NEVER);
    os_callout_reset(&sensor_callout, READ_SENSOR_INTERVAL);
    return;
}

static void
init_timer(void)
{
    /*
     * Initialize the callout for a timer event.
     */
    os_callout_init(&sensor_callout, os_eventq_dflt_get(),
                   timer_ev_cb, NULL);
```

(continues on next page)

(continued from previous page)

```

    os_callout_reset(&sensor_callout, READ_SENSOR_INTERVAL);
    return;
}

```

2. Remove the listener registration and call the `init_timer()` function in `main()`. You can delete the `sensor_register_listener()` function call, but we call the `sensor_unregister_listener()` function to illustrate how to use this function.

```

int
main(int argc, char **argv)
{
    ...

    assert(my_sensor != NULL);
    rc = sensor_register_listener(my_sensor, &listener);
    assert(rc == 0);

    rc = sensor_unregister_listener(my_sensor, &listener);
    assert(rc == 0);

    init_timer();

    /* As the last thing, process events from default event queue. */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }

    return 0;
}

```

Step 4: Rebuilding the Application and Connecting to Console

1. Run the `newt run` command to rebuild the application, create an new image, and start a GDB process:

```

$ newt run thingy_my_sensor 3.0.0
Compiling apps/my_sensor_app/src/main.c
Archiving apps_my_sensor_app.a
Linking ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_sensor_app.
→elf
App image successfully generated: ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_
→sensor_app/my_sensor_app.img
Loading app image into slot 1
[~/dev/myproj/repos/apache-mynewt-core/hw/bsp/nrf52-thingy/nrf52-thingy_debug.sh ~/dev/
→myproj/repos/apache-mynewt-core/hw/bsp/nrf52-thingy ~/dev/myproj/bin/targets/thingy_my_
→sensor/app/apps/my_sensor_app/my_sensor_app]
Debugging ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_sensor_app.
→elf
GNU gdb (GNU Tools for ARM Embedded Processors) 7.8.0.20150604-cvs
...

```

(continues on next page)

(continued from previous page)

```
Reading symbols from ~/dev/myproj/bin/targets/thingy_my_sensor/app/apps/my_sensor_app/my_
→sensor_app.elf...done.
os_tick_idle (ticks=12)
    at repos/apache-mynewt-core/hw/mcu/nordic/nrf52xxx/src/hal_os_tick.c:204
204      if (ticks > 0) {
Resetting target
0x000000dc in ?? ()
(gdb) c
Continuing.
```

3. Connect to the console via RTT:

```
$ telnet localhost 19021
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^].
SEGGER J-Link V6.14h - Real time terminal output
J-Link OB-SAM3U128-V2-NordicSemi compiled Mar 2 2017 12:22:13 V1.0, SN=682562963
Process: JLinkGDBServer

000629 compat> READ_CB: [ secs: 5 usecs: 4088 cputime: 5295643 ]
000642 x = 98.066497804 y = 0.000000000 z = -9806.650390

001282 READ_CB: [ secs: 9 usecs: 992459 cputime: 10284014 ]
001282 x = 117.67980192 y = -39.22660064 z = -9894.910156

001922 READ_CB: [ secs: 14 usecs: 981159 cputime: 15272714 ]
001922 x = 78.453201280 y = -29.41995049 z = -9885.103515

002562 READ_CB: [ secs: 19 usecs: 970088 cputime: 20261643 ]
002562 x = 107.87315366 y = -29.41995049 z = -9885.103515
```

You should see the accelerometer sensor data output from the sensor read data callback.

Enabling OIC Sensor Data Monitoring

This tutorial shows you how to enable sensor data monitoring via the OIC protocol over BLE transport in an application. It extends the example application in the [Enabling an Off-Board Sensor in an Existing Application Tutorial](#) and assumes that you have worked through that tutorial.

This tutorial has two parts:

- *Part 1* shows you how to enable OIC sensor support in the sensors_test application. The procedure only requires setting the appropriate syscfg setting values for the application target. The objective is to show you to quickly bring up the sensors_test application and view the sensor data with the Mynewt Smart Device Controller app that is available for iOS and Android devices.
- *Part 2* shows you how to modify the bleprph_oic application code to add OIC sensor support. The objective is to show you how to use the Sensor Framework API and OIC server API to develop an OIC over BLE sensor server application.

- *Prerequisites*
- *Overview of OIC Support in the Sensor Framework*

Prerequisites

Ensure that you meet the following prerequisites before continuing with the tutorials:

- Complete the [*Enabling an Off-Board Sensor in an Existing Application Tutorial*](#)
- Install the Mynewt Smart Device Controller on an iOS or Android Device. See the [*Sensor Tutorials Overview*](#) on how to install the Mynewt Smart Device Controller app.

Overview of OIC Support in the Sensor Framework

The sensor framework provides support for a sensor enabled application to host the sensor devices as OIC resources. The sensor framework provides the following OIC support:

- Creates OIC resources for each sensor device that is enabled in the application. It creates an OIC discoverable and observable resource for each sensor type that the sensor device is configured for.
- Processes CoAP GET requests for the sensor OIC resources. It reads the sensor data samples, encodes the data, and sends back a response.

The sensor package (`hw/sensor`) defines the following syscfg settings for OIC support:

- `SENSOR_OIC`: This setting specifies whether to enable sensor OIC server support. The setting is enabled by default. The sensor package includes the `net/oic` package for the OIC support when this setting is enabled. The `OIC_SERVER` syscfg setting that specifies whether to enable OIC server support in the `net/oic` package must also be enabled.
- `SENSOR_OIC_OBS_RATE`: Sets the OIC server observation rate.

Enabling OIC Sensor Data Monitoring in the `sensors_test` Application

This tutorial shows you how to enable sensor data monitoring via the OIC protocol over BLE transport in the `sensors_test` application. It extends the example application in the [*Enabling an Off-Board Sensor in an Existing Application Tutorial*](#) and assumes that you have worked through that tutorial.

Like the other off-board sensor tutorials, this tutorial uses an nRF52-DK board connected to an off-board BNO055 sensor device.

This tutorial shows you how to:

- Create and build the target to enable sensor OIC support in the `sensors_test` application.
- Use the Mynewt Smart Device Controller Android or iOS app to view the sensor data from the device.

- *Prerequisites*
- *Step 1: Creating and Building the `sensors_test` Application Image*
- *Step 2: Connecting the Sensor and Loading the Images to the Board*

- Step 3: Viewing Sensor Data from the Mynewt Smart Device Controller

Prerequisites

Read the [Overview of OIC Support in the Sensor Framework](#)

Step 1: Creating and Building the sensors_test Application Image

In this step of the tutorial, we set the following syscfg settings to create a target for the sensors_test application.

- BNO055_OFB and I2C_0: Set to 1 to enable the BNO055 off-board sensor device and the I2C interface 0 in the nRF52 BSP.
- BLE_MAX_CONNECTIONS: Set the number of BLE connections to 4.
- MSYS_1_BLOCK_COUNT: Set the number of entries for the mbuf pool to 52.
- MSYS_1_BLOCK_SIZE: Set the size of mbuf entry to 100.
- OC_APP_RESOURCES: Set the number of server resources to 12.

Note: The SENSOR_OIC, OC_SERVER, BLE_ROLE_PERIPHERAL and BLE_ROLE_BROADCASTER syscfg settings must be enabled to add OIC sensor monitoring over BLE transport support to an application. You do not need to set these settings in the target because the apps/sensors_test package enables the SENSORS_OIC and OC_SERVER syscfg settings by default, and the net/nimble package enables the BLE_ROLE_PERIPHERAL and BLE_ROLE_BROADCASTER settings by default.

1. Run the newt target create command to create the target. We name the target nrf52_bno055_oic_test.

```
$ newt target create nrf52_bno055_oic_test
Target targets/nrf52_bno055_oic_test successfully created
$
```

2. Run the newt target set command to set the app, bsp, and build_profile variables for the target:

```
$ newt target set nrf52_bno055_oic_test app=@apache-mynewt-core/apps/sensors_test_
 ↪bsp=@apache-mynewt-core/hw/bsp/nrf52dk build_profile=debug
Target targets/nrf52_bno055_oic_test successfully set target.app to @apache-mynewt-core/
 ↪apps/sensors_test
Target targets/nrf52_bno055_oic_test successfully set target.bsp to @apache-mynewt-core/
 ↪hw/bsp/nrf52dk
Target targets/nrf52_bno055_oic_test successfully set target.build_profile to debug
$
```

3. Run the newt target set command to set I2C_0=1, BNO055_OFB=1, BLE_MAX_CONNECTIONS=4, MSYS_1_BLOCK_COUNT=52, MSYS_1_BLOCK_SIZE=100, and OC_APP_RESOURCES=11.

Note: If you want to disable the sensor and bno055 shell commands, also set SENSOR_CLI=0 and BNO055_CLI=0.

```
$ newt target set nrf52_bno055_oic_test syscfg=BNO055_OFB=1:I2C_0=1:BLE_MAX_
 ↪CONNECTIONS=4:MSYS_1_BLOCK_COUNT=52:MSYS_1_BLOCK_SIZE=100:OC_APP_RESOURCES=11
Target targets/nrf52_bno055_oic_test successfully set target.syscfg to BNO055_OFB=1:I2C_
 ↪0=1:BLE_MAX_CONNECTIONS=4:MSYS_1_BLOCK_COUNT=52:MSYS_1_BLOCK_SIZE=100:OC_APP_
 (continues on next page)
```

(continued from previous page)

```
↪RESOURCES=11  
$
```

4. Run the `newt build nrf52_bno055_oic_test` and `newt create-image nrf52_bno055_oic_test 1.0.0` commands to build and create the application image.

Step 2: Connecting the Sensor and Loading the Images to the Board

Perform the following steps to reboot the board with the new images:

1. Connect the BNO055 sensor to the nRF52-DK board. See the [Enabling an Off-Board Sensor in an Existing Application Tutorial](#) for instructions.

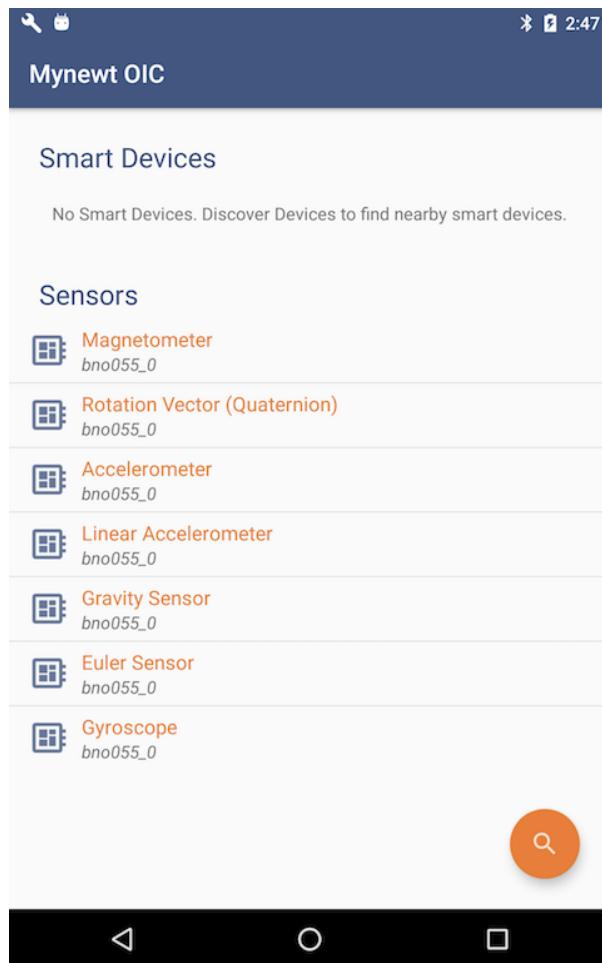
Note: You do not need the serial connection from your computer to the nRF52-DK board for this tutorial because we are not using the shell to view the sensor data.

2. Run the `newt load nrf52_boot` command to load the bootloader.
3. Run the `newt load nrf52_bno055_oic_test` command to load the application image.
4. Power the device OFF and ON to reboot.

Step 3: Viewing Sensor Data from the Mynewt Smart Device Controller

Start the Mynewt Smart Device Controller app on your iOS or Android device to view the sensor data. If you have not installed the Mynewt Smart Device Controller follow the instructions in the [Sensor Tutorials Overview](#) to install the app, then continue with this step of the tutorial.

The Mynewt Smart Device Controller scans for the devices when it starts up and displays the sensors it can view. The following is an example from the Android App:



1. Select Accelerometer to see the sensor data samples:



- Move your BNO055 sensor device around to see the values for the coordinates change.

Adding OIC Sensor Support to the bleprph_oic Application

This tutorial shows you how to modify and add OIC sensor support to the `bleprph_oic` application. This tutorial assumes that have you completed the [Enabling OIC Sensor Data Monitoring in the sensors_test Application Tutorial](#)

Like the other off-board sensor tutorials, this tutorial uses an nRF52-DK board connected to an off-board BNO055 sensor device.

This tutorial shows you how to:

- Modify the `bleprph_oic` application to add OIC sensor support.
- Create and build the target for the new application.
- Use the Mynewt Smart Device Controller iOS or Android app to view the sensor data from the device.

- Prerequisites*
- Overview on How to Add OIC Sensor Support to a BLE Application*
- Step 1: Copying the bleprph_oic source*

- Step 2: Adding Package Dependencies
- Step 3: Setting Syscfg Values to Enable OIC Support
- Step 4: Modifying main.c
 - Adding the Sensor Package Header File:
 - Modifying the `omgr_app_init()` Function
 - Deleting the `app_get_light()` and `app_set_light()` Functions
- Step 5: Creating and Building the Application Image
- Step 6: Connecting the Sensor and Loading the Images to the Board
- Step 7: Viewing Sensor Data from the Mynewt Smart Device Controller

Prerequisites

- Read the [Overview of OIC Support in the Sensor Framework](#)
- Complete the tasks described in the [Enabling OIC Sensor Data Monitoring in the sensors_test Application](#) tutorial.

Overview on How to Add OIC Sensor Support to a BLE Application

The sensor framework makes it very easy to add OIC sensor support to an existing BLE application. The sensor framework exports the `sensor_oic_init()` function that an application calls to create the OIC resources for the sensors and to set up the handlers to process CoAP requests for the resources.

An application uses the `oc` server API in the `net/oic` package to implement OIC server functionality. An application defines an OIC application initialization handler that sets up the OIC resources it supports. The `oc_main_init()` function, that an application calls during initialization in `main()`, calls the OIC application handler.

To add OIC sensor support, we modify the `bleprph_oic` application to call the `sensor_oic_init()` function in the OIC application initialization handler.

Step 1: Copying the `bleprph_oic` source

1. Copy the `@apache-mynewt-core/apps/bleprph_oic` to a new package. We name the new package `apps/bleprph_oic_sensor`. From your project base directory, run the `newt pkg copy` command:

```
$ newt pkg copy @apache-mynewt-core/apps/bleprph_oic @apache-mynewt-core/apps/bleprph_
˓→oic_sensor
Copying package @apache-mynewt-core/apps/bleprph_oic to @apache-mynewt-core/apps/bleprph_
˓→oic_sensor
```

2. The `newt` tools creates the `bleprph_oic_sensor` package in the `~/dev/myproj/repos/apache-mynewt-core/apps/bleprph_oic_sensor` directory. Go to the directory to update the package `pkg.yml` and source files.

```
$ cd repos/apache-mynewt-core/apps/bleprph_oic_sensor
```

Step 2: Adding Package Dependencies

Add the `hw/sensor/` and the `hw/sensor/creator` packages as dependencies in the `pkg.yml` file to include the sensor framework and off-board sensor support.

Note: The `hw/sensor` package automatically includes the `net/oic` package when the `SENSOR_OIC` setting is enabled, so you do not need to include the `net/oic` package as a dependency in this package.

`pkg.deps:`

- `kernel/os`
- `net/nimble/controller`
- `net/nimble/host`
- `net/nimble/host/services/gap`
- `net/nimble/host/services/gatt`
- `net/nimble/host/store/ram`
- `net/nimble/transport/ram`
- ...
- `hw/sensor`
- `hw/sensor/creator`

Step 3: Setting Syscfg Values to Enable OIC Support

Add the following setting values to `syscfg.vals` in the `syscfg.yml` file:

- `SENSOR_OIC: 1`: This setting enables OIC sensor support in the `hw/sensors` package.
- `OC_SERVER: 1` : This setting enables OIC server support in the `net/oic` package.
- `FLOAT_USER: 1`: This setting enables floating pointing support in the `encoding/tinycbor` package.
- `ADVERTISE_128BIT_UUID: 1` and `ADVERTISE_16BIT_UUID: 0`: These settings enable BLE 128 bit UUID and disables 16 bit UUID advertisement. The IoTivity library that is used to build the OIC Apps on the iOS and Android devices only sees 128 bit UUID advertisements.

`syscfg.vals:`

```
...
SENSOR_OIC: 1
OC_SERVER: 1
FLOAT_USER: 1
ADVERTISE_128BIT_UUID: 1
ADVERTISE_16BIT_UUID: 0
```

Step 4: Modifying main.c

The `bleprph_oic` application defines the `omgr_app_init()` function for the OIC application initialization handler. The function creates an OIC light resource. We modify the function to call the `sensor_oic_init()` function to create the OIC sensor resources instead of creating the OIC light resource.

We make the following modifications to `main.c`:

- Add the sensor package header file.
- Modify the `omgr_app_init()` function to call the `sensor_oic_init()` function, and delete the code to create the OIC light resource.

- Delete the OIC application request handler functions that process the CoAP requests for the light resource.

Adding the Sensor Package Header File:

Add the sensor package header file `sensor/sensor.h` below `#include "bleprph.h"` file:

```
#include "bleprph.h"

#include <sensor/sensor.h>
```

Modifying the `omgr_app_init()` Function

Make the following modifications to the `omgr_app_init()` function:

1. Delete the code segment that creates the OIC device and resource. The lines to delete are highlighted below:

```
static void
omgr_app_init(void)
{
    oc_resource_t *res;

    oc_init_platform("MyNewt", NULL, NULL);
    oc_add_device("/oic/d", "oic.d.light", "MynewtLed", "1.0", "1.0", NULL,
                  NULL);

    res = oc_new_resource("/light/1", 1, 0);
    oc_resource_bind_resource_type(res, "oic.r.light");
    oc_resource_bind_resource_interface(res, OC_IF_RW);
    oc_resource_set_default_interface(res, OC_IF_RW);

    oc_resource_set_discoverable(res);
    oc_resource_set_periodic_observable(res, 1);
    oc_resource_set_request_handler(res, OC_GET, app_get_light);
    oc_resource_set_request_handler(res, OC_PUT, app_set_light);
    oc_add_resource(res);

}
```

2. Add the following ```oc_add_device()``` function call to create an OIC resource for the sensor device:

```
static void
omgr_app_init(void)
{
    oc_init_platform("MyNewt", NULL, NULL);

    oc_add_device("/oic/d", "oic.d.sensy", "sensy", "1.0", "1.0", NULL, NULL);

}
```

3. Add the call to the `sensor_oic_init()` function to initialize the sensor framework OIC server support:

```
static void
omgr_app_init(void)
{
    oc_init_platform("MyNewt", NULL, NULL);

    oc_add_device("/oic/d", "oic.d.sensy", "sensy", "1.0", "1.0", NULL, NULL);

    sensor_oic_init();

}
```

Deleting the app_get_light() and app_set_light() Functions

Since we modify the application to no longer create an OIC light resource, the `app_get_light()` and the `app_set_light()` handler functions that process read and write requests are not used. We need to delete the functions to avoid compilation errors. Search for the two functions and delete them.

Step 5: Creating and Building the Application Image

In this step of the tutorial we create and build an application image for the `bleprph_oic_sensor` application to verify that the application serves sensor data over OIC correctly.

We use the same syscfg settings from the [Enabling OIC Sensor Data Monitoring in the sensors_test Application Tutorial](#)

- From your project base directory, run the `newt create target` command to create a new target named `nrf52_bleprph_oic_bno055`:

```
$ newt target create nrf52_bleprph_oic_bno055
Target targets/nrf52_bleprph_oic_bno055 successfully created
```

- Run the `newt target set` command to set the app, bsp, and build_profile variables for the target.

```
$ newt target set nrf52_bleprph_oic_bno055 app=@apache-mynewt-core/apps/bleprph_oic_
↳ sensor bsp=@apache-mynewt-core/hw/bsp/nrf52dk build_profile=debug
Target targets/nrf52_bleprph_oic_bno055 successfully set target.app to @apache-mynewt-
↳ core/apps/bleprph_oic_sensor
Target targets/nrf52_bleprph_oic_bno055 successfully set target.bsp to @apache-mynewt-
↳ core/hw/bsp/nrf52dk
Target targets/nrf52_bleprph_oic_bno055 successfully set target.build_profile to debug
$
```

- Run the `newt target set` command to set `I2C_0=1`, `BN0055_OFB=1`, `BLE_MAX_CONNECTIONS=4`, `MSYS_1_BLOCK_COUNT=52`, `MSYS_1_BLOCK_SIZE=100`, and `OC_APP_RESOURCES=11`.

```
$ newt target set nrf52_bleprph_oic_bno055 syscfg=BN0055_OFB=1:I2C_0=1:BLE_MAX_
↳ CONNECTIONS=4:MSYS_1_BLOCK_COUNT=52:MSYS_1_BLOCK_SIZE=100:OC_APP_RESOURCES=11
Target targets/nrf52_bleprph_oic_bno055 successfully set target.syscfg to BN0055_
↳ OFB=1:I2C_0=1:BLE_MAX_CONNECTIONS=4:MSYS_1_BLOCK_COUNT=52:MSYS_1_BLOCK_SIZE=100:OC_APP_
↳ RESOURCES=11
$
```

4. Run the `newt build nrf52_bleprph_oic_bno055` and `newt create-image nrf52_bleprph_oic_bno055 1.0.0` commands to build and create the application image.

Step 6: Connecting the Sensor and Loading the Images to the Board

Perform the following steps to reboot the board with the new images:

1. Connect the BNO055 sensor to the nRF52-DK board. See the [Enabling an Off-Board Sensor in an Existing Application Tutorial](#) for instructions.

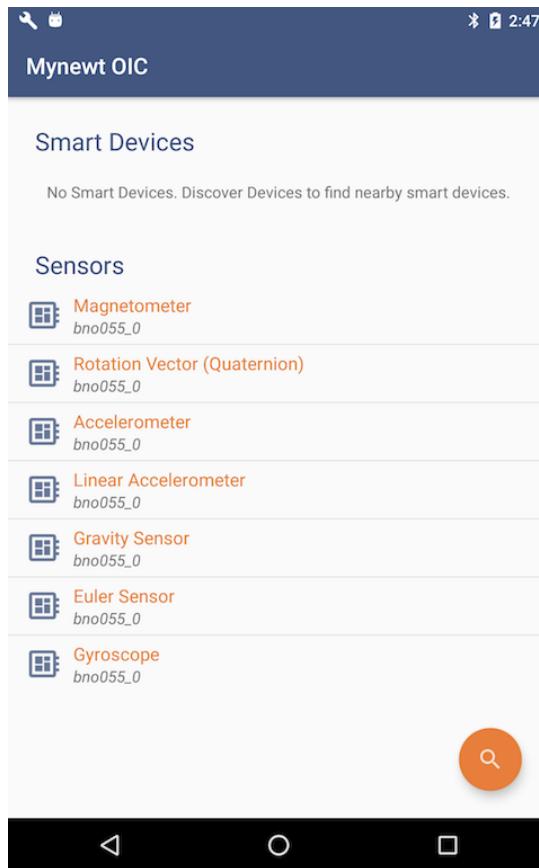
Note: You do not need the serial connection from your computer directly to the nRF52-DK board because we are not using the shell to view the sensor data.

2. Run the `newt load nrf52_boot` command to load the bootloader. You should already have this target built from the [Enabling an Off-Board Sensor in an Existing Application Tutorial](#)
3. Run the `newt load nrf52_bno055_oic_test` command to load the application image.
4. Power the device OFF and ON to reboot.

Step 7: Viewing Sensor Data from the Mynewt Smart Device Controller

Start the Mynewt Smart Device Controller app on your iOS or Android device to view the sensor data. You should already have the app installed from the [Enabling OIC Sensor Data Monitoring in the sensors_test Application Tutorial](#)

The Mynewt Smart Device Controller scans for the devices when it starts up and displays the sensors it can view. The following is an example from the Android App:



1. Select Accelerometer to see the sensor data samples:



2. Move your BNO055 sensor device around to see the values for the coordinates change.

4.10.6 Air Quality Sensor Project

Air Quality Sensor Project

This tutorial will show you how to set up and use the Senseair K30 air quality sensor with the nRF52 Development Kit. Afterwards, you can set it up via Bluetooth so you can read values remotely.

- *Prerequisites*
- *Setting Up the Source Tree*
- *Create a Test Project*
- *Create Packages For Drivers*
- *Add CLI Commands For Testing Drivers*
- *Using HAL for Drivers*

Prerequisites

- Complete one of the other tutorials (e.g. [Project Blinky](#)) to familiarize yourself with Mynewt
- Nordic nRF52 Development - PCA 10040
- Senseair K30 CO2 Sensor

Setting Up the Source Tree

To start, create a new project under which you will do development for this application:

```
$ mkdir $HOME/src
$ cd $HOME/src
$ newt new air_quality
$ cd air_quality
```

If you are using a different development board, you will need to know the board support package for that hardware. You can look up its location, add it to your project, and fetch that along with the core OS components. Since the nRF52DK is supported in the Mynewt Core, we don't need to do much here.

```
[user@IsMyLaptop:~/src/air_quality]$ cat project.yml
```

Your `project.yml` file should look like this:

```
project.name: "air_quality"

project.repositories:
    - apache-mynewt-core

# Use github's distribution mechanism for core ASF libraries.
# This provides mirroring automatically for us.
#
repository.apache-mynewt-core:
    type: github
    vers: 1.12.0
    user: apache
    repo: mynewt-core
```

After updating `vers` to the latest release of Mynewt (see [Latest News](#) at the top of the page), it's time to fetch needed repositories:

```
[user@IsMyLaptop:~/src/air_quality]$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
Downloading repository mynewt-nimble (commit: master) ...
```

Once the command finishes (which may take a while), there should be `repos` folder in your project's root directory. You can find there Mynewt Core components, NimBLE stack and other dependencies.

Next, create a target for the nRF52DK bootloader:

```
[user@IsMyLaptop:~/src/air_quality]$ newt target create boot_nrf52dk
Target targets/boot_nrf52dk successfully created
[user@IsMyLaptop:~/src/air_quality]$ newt target set boot_nrf52dk bsp=@apache-mynewt-
```

(continues on next page)

(continued from previous page)

```

→core/hw/bsp/nordic_pca10040
Target targets/boot_nrf52dk successfully set target.bsp to @apache-mynewt-core/hw/bsp/
→nordic_pca10040
[user@IsMyLaptop:~/src/air_quality]$ newt target set boot_nrf52dk app=@mcuboot/boot/
→mynewt
Target targets/boot_nrf52dk successfully set target.app to @mcuboot/boot/mynewt
[user@IsMyLaptop:~/src/air_quality]$ newt target set boot_nrf52dk build_profile=optimized
Target targets/boot_nrf52dk successfully set target.build_profile to optimized
[user@IsMyLaptop:~/src/air_quality]$ newt target show
targets/boot_nrf52dk
    app=@mcuboot/boot/mynewt
    bsp=@apache-mynewt-core/hw/bsp/nordic_pca10040
    build_profile=optimized
targets/my_blinky_sim
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/native
    build_profile=debug

```

Build the bootloader target and load it onto the board:

```

[user@IsMyLaptop:~/src/air_quality]$ newt build boot_nrf52dk
Building target targets/boot_nrf52dk
...
Linking /home/user/src/air_quality/bin/targets/boot_nrf52dk/app/@mcuboot/boot/mynewt/
→mynewt.elf
Target successfully built: targets/boot_nrf52dk

[user@IsMyLaptop:~/src/air_quality]$ newt load boot_nrf52dk
Loading bootloader ...

```

Create a Test Project

Now that you have your system setup, you can start building the application. First you want to create a project for yourself - since we're eventually going to want to be able to access the data via Bluetooth, let's use the bleprph Bluetooth Peripheral project as the project template.

```

[user@IsMyLaptop:~/src/air_quality]$ mkdir apps/air_quality
[user@IsMyLaptop:~/src/air_quality]$ cp repos/apache-mynewt-nimble/apps/bleprph/pkg.yml
→apps/air_quality/
[user@IsMyLaptop:~/src/air_quality]$ cp -Rp repos/apache-mynewt-nimble/apps/bleprph/src
→apps/air_quality/

```

Modify the apps/air_quality/pkg.yml for air_quality in order to change the *pkg.name* to be *apps/air_quality*. You'll need to add the @apache-mynewt-nimble/ path to all the package dependencies, since the app no longer resides within the apache-mynewt-nimble repository. Remember to add double quotes to the paths, as in the example below:

```
[user@IsMyLaptop:~/src/air_quality]$ cat apps/air_quality/pkg.yml
```

```

pkg.name: apps/air_quality
pkg.type: app
pkg.description: BLE Air Quality application.

```

(continues on next page)

(continued from previous page)

```

pkg.author: "Apache Mynewt <dev@mynewt.apache.org>"
pkg.homepage: "http://mynewt.apache.org/"
pkg.keywords:

pkg.deps:
  - "@apache-mynewt-core/boot/split"
  - "@mcuboot/boot/bootutil"
  - "@apache-mynewt-core/kernel/os"
  - "@apache-mynewt-core/mgmt/imgmgr"
  - "@apache-mynewt-core/mgmt/smp"
  - "@apache-mynewt-core/mgmt/smp/transport/ble"
  - "@apache-mynewt-core/sys/console"
  - "@apache-mynewt-core/sys/log"
  - "@apache-mynewt-core/sys/log/modlog"
  - "@apache-mynewt-core/sys/stats"
  - "@apache-mynewt-core/sys/sysinit"
  - "@apache-mynewt-core/sys/id"
  - "@apache-mynewt-nimble/nimble/host"
  - "@apache-mynewt-nimble/nimble/host/services/ans"
  - "@apache-mynewt-nimble/nimble/host/services/dis"
  - "@apache-mynewt-nimble/nimble/host/services/gap"
  - "@apache-mynewt-nimble/nimble/host/services/gatt"
  - "@apache-mynewt-nimble/nimble/host/store/config"
  - "@apache-mynewt-nimble/nimble/host/util"

```

Next create a target for it:

```

[user@IsMyLaptop:~/src/air_quality]$ newt target create air_q
Target targets/air_q successfully created
[user@IsMyLaptop:~/src/air_quality]$ newt target set air_q bsp=@apache-mynewt-core/hw/
 ↪bsp/nordic_pca10040
Target targets/air_q successfully set target.bsp to @apache-mynewt-core/hw/bsp/nordic_
 ↪pca10040
[user@IsMyLaptop:~/src/air_quality]$ newt target set air_q app=apps/air_quality
Target targets/air_q successfully set target.app to apps/air_quality
[user@IsMyLaptop:~/src/air_quality]$ newt target set air_q build_profile=debug
Target targets/air_q successfully set target.build_profile to debug
[user@IsMyLaptop:~/src/air_quality]$ newt build air_q
...
Linking /home/user/src/air_quality/bin/targets/air_q/app/apps/air_quality/air_quality.elf
Target successfully built: targets/air_q

```

Create Packages For Drivers

We need to enable the SenseAir K30 CO₂ sensor, which will connect to the board over a serial port. To start development of the driver, you first need to create a package description for it, and add stubs for sources.

The first thing to do is to create the directory structure for your driver:

```
[user@IsMyLaptop:~/src/air_quality]$ mkdir -p libs/my_drivers/senseair/include/senseair  
[user@IsMyLaptop:~/src/air_quality]$ mkdir -p libs/my_drivers/senseair/src
```

Now you can add the files you need. You'll need a `pkg.yml` to describe the driver, and then header stub followed by source stub.

```
[user@IsMyLaptop:~/src/air_quality]$ cat libs/my_drivers/senseair/pkg.yml
```

```
#  
# Licensed to the Apache Software Foundation (ASF) under one  
# or more contributor license agreements. See the NOTICE file  
# distributed with this work for additional information  
# regarding copyright ownership. The ASF licenses this file  
# to you under the Apache License, Version 2.0 (the  
# "License"); you may not use this file except in compliance  
# with the License. You may obtain a copy of the License at  
#  
# http://www.apache.org/licenses/LICENSE-2.0  
#  
# Unless required by applicable law or agreed to in writing,  
# software distributed under the License is distributed on an  
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY  
# KIND, either express or implied. See the License for the  
# specific language governing permissions and limitations  
# under the License.  
#  
pkg.name: libs/my_drivers/senseair  
pkg.description: Host side of the nimble Bluetooth Smart stack.  
pkg.author: "Apache Mynewt <dev@mynewt.apache.org>"  
pkg.homepage: "http://mynewt.apache.org/"  
pkg.keywords:  
    - ble  
    - bluetooth  
  
pkg.deps:  
    - "@apache-mynewt-core/kernel/os"
```

```
[user@IsMyLaptop:~/src/air_quality]$ cat libs/my_drivers/senseair/include/senseair/  
senseair.h
```

```
/*  
 * Licensed to the Apache Software Foundation (ASF) under one  
 * or more contributor license agreements. See the NOTICE file  
 * distributed with this work for additional information  
 * regarding copyright ownership. The ASF licenses this file  
 * to you under the Apache License, Version 2.0 (the
```

(continues on next page)

(continued from previous page)

```

* "License"); you may not use this file except in compliance
* with the License. You may obtain a copy of the License at
*
*   http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
* software distributed under the License is distributed on an
* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
* KIND, either express or implied. See the License for the
* specific language governing permissions and limitations
* under the License.
*/
#ifndef _SENSEAIR_H_
#define _SENSEAIR_H_

void senseair_init(void);

#endif /* _SENSEAIR_H_ */

```

[user@IsMyLaptop:~/src/air_quality]\$ cat libs/my_drivers/senseair/src/senseair.c

```

/***
* Licensed to the Apache Software Foundation (ASF) under one
* or more contributor license agreements. See the NOTICE file
* distributed with this work for additional information
* regarding copyright ownership. The ASF licenses this file
* to you under the Apache License, Version 2.0 (the
* "License"); you may not use this file except in compliance
* with the License. You may obtain a copy of the License at
*
*   http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
* software distributed under the License is distributed on an
* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
* KIND, either express or implied. See the License for the
* specific language governing permissions and limitations
* under the License.
*/

void
senseair_init(void)
{
}

```

And add a dependency to this package in your pkg.yml file.

[user@IsMyLaptop:~/src/air_quality]\$ cat apps/air_quality/pkg.yml

```

pkg.name: apps/air_quality
pkg.type: app

```

(continues on next page)

(continued from previous page)

```

pkg.description: BLE Air Quality application.
pkg.author: "Apache Mynewt <dev@mynewt.apache.org>"
pkg.homepage: "http://mynewt.apache.org/"
pkg.keywords:

pkg.deps:
  - "@apache-mynewt-core/boot/split"
  - "@mcuboot/boot/bootutil"
  - "@apache-mynewt-core/kernel/os"
  - "@apache-mynewt-core/mgmt/imgmgr"
  ...
  - "@apache-mynewt-nimble/nimble/host/store/config"
  - "@apache-mynewt-nimble/nimble/host/util"
  - libs/my_drivers/senseair

```

Include driver's header and add a call to `senseair_init()` in your main function to initialize this driver:

```
$ cat apps/air_quality/src/main.c
```

```

#include <senseair/senseair.h>
...

int
mynewt_main(int argc, char **argv)
{
    ...
    int rc;

    /* Initialize OS */
    sysinit();

    /* Initialize sensor driver */
    senseair_init();
    ...

    return 0;
}

```

Add CLI Commands For Testing Drivers

While developing the driver, it would be helpful to issue operations from the console to verify the driver is responding correctly. Since the nRF52DK only has one UART, which will be used to connect to the CO2 sensor, the console we'll use instead is the *Segger RTT Console*. To configure this, make the following changes in your project's `syscfg.yml` file:

```
[user@IsMyLaptop:~/src/air_quality]$ cat targets/air_q/syscfg.yml
```

```

syscfg.vals:
  # Enable the shell task.
  SHELL_TASK: 1
  # Use the RTT Console

```

(continues on next page)

(continued from previous page)

```
CONSOLE_UART: 0
CONSOLE_RTT: 1
```

Also, add a shell dependency in your app's `pkg.yml`:

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

Then register your `senseair` command with the shell by adding the following to `libs/my_drivers/senseair/src/senseair.c`

```
#include <syscfg/syscfg.h>
#include <shell/shell.h>
#include <console/console.h>
#include <assert.h>

static int senseair_shell_func(int argc, char **argv);
static struct shell_cmd senseair_cmd = {
    .sc_cmd = "senseair",
    .sc_cmd_func = senseair_shell_func,
};

void
senseair_init(void)
{
    int rc;

    rc = shell_cmd_register(&senseair_cmd);
    assert(rc == 0);
}

static int
senseair_shell_func(int argc, char **argv)
{
    console_printf("Yay! Somebody called!\n");
    return 0;
}
```

Build the target, create an image, and load it onto your board.

```
[user@IsMyLaptop:~/src/air_quality]$ newt create-image air_q 1.0
Compiling ...
Linking /home/user/src/air_quality/bin/targets/air_q/app/apps/air_quality/air_quality.elf
App image successfully generated: /home/user/src/air_quality/bin/targets/air_q/app/apps/
air_quality/air_quality.img
[user@IsMyLaptop:~/src/air_quality]$ newt load air_q
Loading app image into slot 1 ...
```

Then run `telnet localhost 19021` to start the RTT Console.

```
[user@IsMyLaptop:~]$ telnet localhost 19021
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SEGGER J-Link V6.30j - Real time terminal output
J-Link OB-SAM3U128-V2-NordicSemi compiled Jan 12 2018 16:05:20 V1.0, SN=682771074
Process: JLinkGDBServerCLExe
x03 0x03 0x11 0x18 0x0f 0x09 0x6e 0x69 0x6d 0x62 0x6c 0x65 0x2d 0x62 0x6c 0x65 0x70 0x72
↪ 0x70 0x68 0x02 0x0a 0x00 0x00      0x00 0x00 0x00 0x00
000006 [ts=46872ssb, mod=4 level=0] Command complete: cmd_pkts=1 ogf=0x8 ocf=0x8 status=0
000006 [ts=46872ssb, mod=4 level=1] GAP procedure initiated: advertise; disc_mode=2 adv_
↪ channel_map=0 own_addr_type=0      adv_filter_policy=0 adv_itvl_min=0 adv_itvl_max=0
000006 [ts=46872ssb, mod=4 level=0] ble_hs_hci_cmd_send: ogf=0x08 ocf=0x0006 len=15
000006 [ts=46872ssb, mod=4 level=0] 0x06 0x20 0x0f 0x30 0x00 0x60 0x00 0x00 0x00 0x00
↪ 0x00 0x00 0x00 0x00 0x00      0x07 0x00
000006 [ts=46872ssb, mod=4 level=0] Command complete: cmd_pkts=1 ogf=0x8 ocf=0x6 status=0
000006 [ts=46872ssb, mod=4 level=0] ble_hs_hci_cmd_send: ogf=0x08 ocf=0x000a len=1
000006 [ts=46872ssb, mod=4 level=0] 0x0a 0x20 0x01 0x01
000006 [ts=46872ssb, mod=4 level=0] Command complete: cmd_pkts=1 ogf=0x8 ocf=0xa status=0
000006 [ts=46872ssb, mod=4 level=0] Command complete: cmd_pkts=1 ogf=0x0  ocf=0x0

001215 compat>

001957 compat> help
help
002162 help
002162 tasks
002162 mpool
002162 date
002162 senseair
002162 compat> senseair
senseair
002514 Yay! Somebody called!
002514 compat>
```

If you can see the `senseair` command, and get the proper response, you can connect the hardware to your board and start developing code for the driver itself.

Using HAL for Drivers

We will connect the CO₂ sensor using a serial port connection to the UART. We'll also use the HAL UART abstraction to do the UART port setup and data transfer. That way you don't need to have any platform dependent pieces within your little driver. Moreover, this also gives you the option to connect this sensor to another board, like Olimex or the Arduino Primo.

You will now see what the driver code ends up looking like. Here's the header file, filled in from the stub you created earlier:

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements. See the NOTICE file
 * distributed with this work for additional information
```

(continues on next page)

(continued from previous page)

```

* regarding copyright ownership. The ASF licenses this file
* to you under the Apache License, Version 2.0 (the
* "License"); you may not use this file except in compliance
* with the License. You may obtain a copy of the License at
*
*   http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
* software distributed under the License is distributed on an
* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
* KIND, either express or implied. See the License for the
* specific language governing permissions and limitations
* under the License.
*/
#ifndef _SENSEAIR_H_
#define _SENSEAIR_H_

enum senseair_read_type {
    SENSEAIR_CO2,
};

int senseair_init(int uartno);

int senseair_read(enum senseair_read_type);

#endif /* _SENSEAIR_H_ */

```

As you can see, logical UART number has been added to the init routine. A ‘read’ function has also been added, which is a blocking read. If you were making a commercial product, you would probably have a callback for reporting the results.

And here is the source for the driver:

```

/***
* Licensed to the Apache Software Foundation (ASF) under one
* or more contributor license agreements. See the NOTICE file
* distributed with this work for additional information
* regarding copyright ownership. The ASF licenses this file
* to you under the Apache License, Version 2.0 (the
* "License"); you may not use this file except in compliance
* with the License. You may obtain a copy of the License at
*
*   http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing,
* software distributed under the License is distributed on an
* "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
* KIND, either express or implied. See the License for the
* specific language governing permissions and limitations
* under the License.
*/
#include <string.h>
#include <syscfg/syscfg.h>

```

(continues on next page)

(continued from previous page)

```

#include <shell/shell.h>
#include <console/console.h>
#include <os/os.h>

#include <hal/hal_uart.h>

#include "senseair/senseair.h"

static const uint8_t cmd_read_co2[] = {
    0xFE, 0x44, 0X00, 0X08, 0X02, 0X9F, 0X25
};

static int senseair_shell_func(int argc, char **argv);
static struct shell_cmd senseair_cmd = {
    .sc_cmd = "senseair",
    .sc_cmd_func = senseair_shell_func,
};

struct senseair {
    int uart;
    struct os_sem sema;
    const uint8_t *tx_data;
    int tx_off;
    int tx_len;
    uint8_t rx_data[32];
    int rx_off;
    int value;
} senseair;

static int
senseair_tx_char(void *arg)
{
    struct senseair *s = &senseair;
    int rc;

    if (s->tx_off >= s->tx_len) {
        /*
         * Command tx finished.
         */
        s->tx_data = NULL;
        return -1;
    }

    rc = s->tx_data[s->tx_off];
    s->tx_off++;
    return rc;
}

/*
 * CRC for modbus over serial port.
 */
static const uint16_t mb_crc_tbl[] = {

```

(continues on next page)

(continued from previous page)

```

0x0000, 0xcc01, 0xd801, 0x1400, 0xf001, 0x3c00, 0x2800, 0xe401,
0xa001, 0x6c00, 0x7800, 0xb401, 0x5000, 0x9c01, 0x8801, 0x4400
};

static uint16_t
mb_crc(const uint8_t *data, int len, uint16_t crc)
{
    while (len-- > 0) {
        crc ^= *data++;
        crc = (crc >> 4) ^ mb_crc_tbl[crc & 0xf];
        crc = (crc >> 4) ^ mb_crc_tbl[crc & 0xf];
    }
    return crc;
}

static int
mb_crc_check(const void *pkt, int len)
{
    uint16_t crc, cmp;
    uint8_t *bp = (uint8_t *)pkt;

    if (len < sizeof(crc) + 1) {
        return -1;
    }
    crc = mb_crc(pkt, len - 2, 0xffff);
    cmp = bp[len - 2] | (bp[len - 1] << 8);
    if (crc != cmp) {
        return -1;
    } else {
        return 0;
    }
}

static int
senseair_rx_char(void *arg, uint8_t data)
{
    struct senseair *s = (struct senseair *)arg;
    int rc;

    if (s->rx_off >= sizeof(s->rx_data)) {
        s->rx_off = 0;
    }
    s->rx_data[s->rx_off] = data;
    s->rx_off++;

    if (s->rx_off == 7) {
        rc = mb_crc_check(s->rx_data, s->rx_off);
        if (rc == 0) {
            s->value = s->rx_data[3] * 256 + s->rx_data[4];
            os_sem_release(&s->sema);
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    return 0;
}

void
senseair_tx(struct senseair *s, const uint8_t *tx_data, int data_len)
{
    s->tx_data = tx_data;
    s->tx_len = data_len;
    s->tx_off = 0;
    s->rx_off = 0;

    hal_uart_start_tx(s->uart);
}

int
senseair_read(enum senseair_read_type type)
{
    struct senseair *s = &senseair;
    const uint8_t *cmd;
    int cmd_len;
    int rc;

    if (s->tx_data) {
        /*
         * busy
         */
        return -1;
    }
    switch (type) {
    case SENSEAIR_CO2:
        cmd = cmd_read_co2;
        cmd_len = sizeof(cmd_read_co2);
        break;
    default:
        return -1;
    }
    senseair_tx(s, cmd, cmd_len);
    rc = os_sem_pend(&s->sema, OS_TICKS_PER_SEC / 2);
    if (rc == OS_TIMEOUT) {
        /*
         * timeout
         */
        return -2;
    }
    return s->value;
}

static int
senseair_shell_func(int argc, char **argv)
{
    int value;
    enum senseair_read_type type;
}

```

(continues on next page)

(continued from previous page)

```

if (argc < 2) {
usage:
    console_printf("%s co2\n", argv[0]);
    return 0;
}
if (!strcmp(argv[1], "co2")) {
    type = SENSEAIR_CO2;
} else {
    goto usage;
}
value = senseair_read(type);
if (value >= 0) {
    console_printf("Got %d\n", value);
} else {
    console_printf("Error while reading: %d\n", value);
}
return 0;
}

int
senseair_init(int uartno)
{
    int rc;
    struct senseair *s = &senseair;

    rc = shell_cmd_register(&senseair_cmd);
    if (rc) {
        return rc;
    }

    rc = os_sem_init(&s->sema, 1);
    if (rc) {
        return rc;
    }
    rc = hal_uart_init_cbs(uartno, senseair_tx_char, NULL,
                           senseair_rx_char, &senseair);
    if (rc) {
        return rc;
    }
    rc = hal_uart_config(uartno, 9600, 8, 1, HAL_UART_PARITY_NONE,
                         HAL_UART_FLOW_CTL_NONE);
    if (rc) {
        return rc;
    }
    s->uart = uartno;

    return 0;
}

```

And your modified `mynewt_main()` for senseair driver init.

```

int
mynewt_main(int argc, char **argv)
{
    ...
    senseair_init();
    ...
}

```

You can see from the code that you are using the HAL interface to open a UART port, and using OS semaphore as a way of blocking the task when waiting for read response to come back from the sensor.

Now comes the fun part: Hooking up the sensor! It's fun because a) hooking up a sensor is always fun and b) the SenseAir sensor's PCB is entirely unlabeled, so you'll have to trust us on how to hook it up.

You'll have to do a little soldering. I soldered some header pins to the SenseAir K30 board to make connecting wires easier using standard jumper wires, but you can also just solder wires straight to the board if you prefer.

Here's what your SenseAir board should look like once it's wired up:

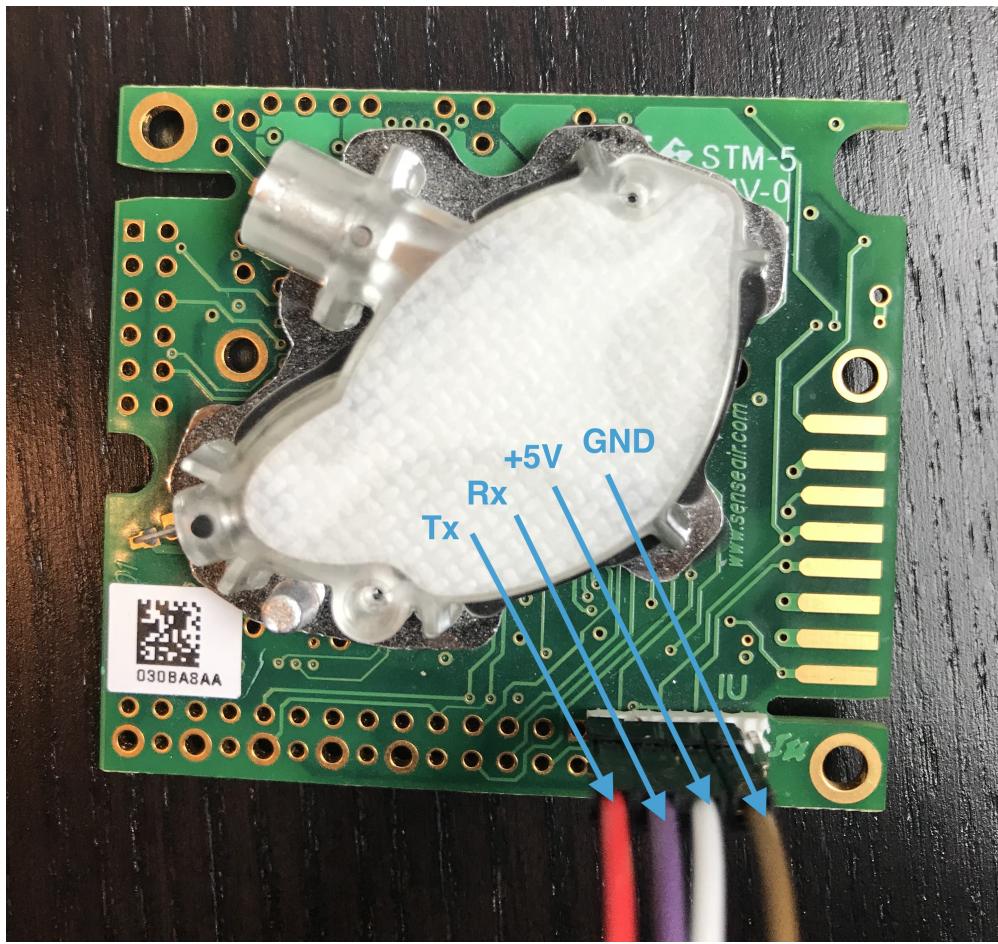


Fig. 5: SenseAir Wiring

Now that you have that wired up, let's connect it to the nRF52DK board. Since we will be using the built-in UART, we can simply connect it to the pre-configured pins for TX (P.06) and RX (P.08). Here's what your board should look like once everything is connected:

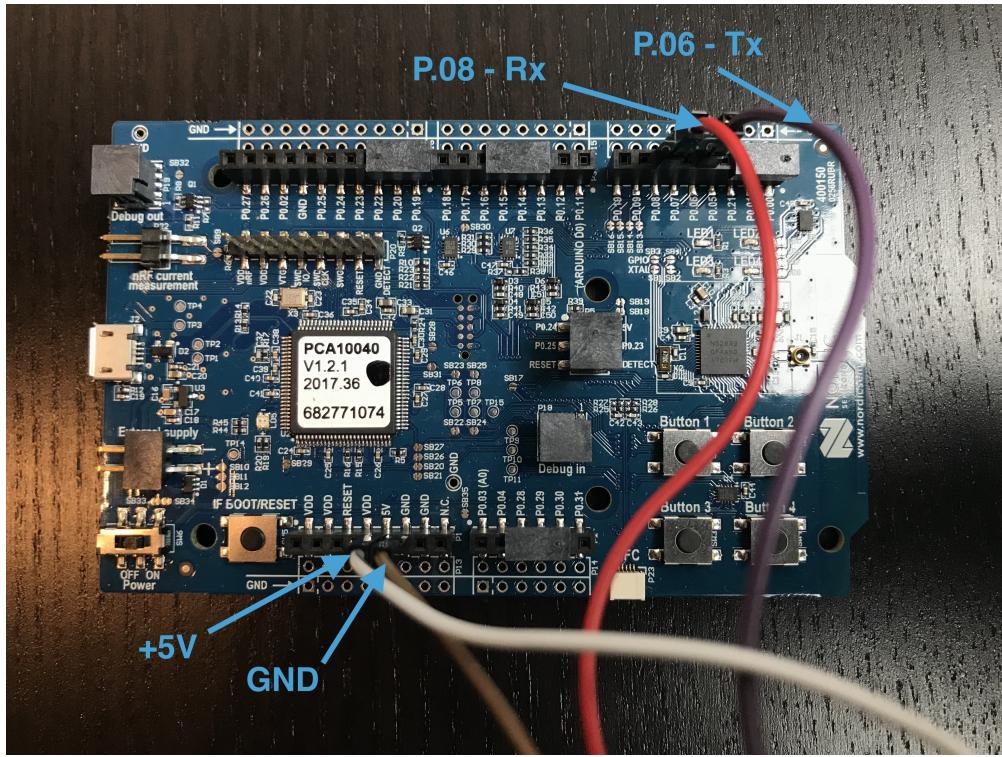


Fig. 6: SenseAir and nRF52DK Wiring

Everything is wired and you're ready to go! Build and load your new app:

```
[user@IsMyLaptop:~/src/air_quality]$ newt build air_q
Building target targets/air_q
Compiling apps/air_quality/src/main.c
Archiving apps_air_quality.a
Linking /home/user/src/air_quality/bin/targets/air_q/app/apps/air_quality/air_quality.elf
Target successfully built: targets/air_q
[user@IsMyLaptop:~/src/air_quality]$ newt create-image air_q 1.0.0
App image successfully generated: /home/user/src/air_quality/bin/targets/air_q/app/apps/
└─air_quality/air_quality.img
[user@IsMyLaptop:~/src/air_quality]$ newt load air_q
Loading app image into slot 1 ...
```

Now, you should be able to connect to your serial port and read values:

```
user@IsMyLaptop:~$ telnet localhost 19021
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SEGGER J-Link V6.30j - Real time terminal output
J-Link OB-SAM3U128-V2-NordicSemi compiled Jan 12 2018 16:05:20 V1.0, SN=682771074
Process: JLinkGDBServerCLExe
x03 0x03 0x11 0x18 0x0f 0x09 0x6e 0x69 0x6d 0x62 0x6c 0x65 0x2d 0x62 0x6c 0x65 0x70 0x72
↪ 0x70 0x68 0x02 0x0a 0x00 0x00      0x00 0x00 0x00 0x00
000006 [ts=46872ssb, mod=4 level=0] Command complete: cmd_pkts=1 ogf=0x8 ocf=0x8 status=0
(continues on next page)
```

(continued from previous page)

```
000006 [ts=46872ssb, mod=4 level=1] GAP procedure initiated: advertise; disc_mode=2 adv_
↪channel_map=0 own_addr_type=0           adv_filter_policy=0 adv_itvl_min=0 adv_itvl_max=0
000006 [ts=46872ssb, mod=4 level=0] ble_hs_hci_cmd_send: ogf=0x08 ocf=0x0006 len=15
000006 [ts=46872ssb, mod=4 level=0] 0x06 0x20 0x0f 0x30 0x00 0x60 0x00 0x00 0x00 0x00
↪0x00 0x00 0x00 0x00 0x00 0x07      0x00
000006 [ts=46872ssb, mod=4 level=0] Command complete: cmd_pkts=1 ogf=0x8 ocf=0x6 status=0
000006 [ts=46872ssb, mod=4 level=0] ble_hs_hci_cmd_send: ogf=0x08 ocf=0x000a len=1
000006 [ts=46872ssb, mod=4 level=0] 0x0a 0x20 0x01 0x01
000007 [ts=54684ssb, mod=4 level=0] Command complete: cmd_pkts=1 ogf=0x8 ocf=0xa status=0
000007 [ts=54684ssb, mod=4 level=0] Command complete: cmd_pkts=1 ogf=0x0 ocf=0x0

000895 compat>

000998 compat> help
help
001414 help
001414 tasks
001414 mpool
001414 date
001414 senseair
001414 compat> senseair
senseair
001714 senseair co2
001714 compat> senseair co2
senseair co2
002098 Got 0
002098 compat> senseair co2
senseair co2
002719 Got 1168
```

And you're getting valid readings! Congratulations!

Next we'll hook this all up via Bluetooth so that you can read those values remotely.

Air Quality Sensor Project via Bluetooth

This is a follow-on project to the [Basic Air Quality Sensor](#) project; so it is assumed that you have worked through that project and have your CO₂ sensor working properly with your nRF52DK board.

So let's get started making this thing Bluetooth enabled!

Add Bluetooth GATT Services

Since we already built the previous demo on the [bluetooth peripheral](#) basic app most of the Bluetooth plumbing has already been taken care of for us. What's left is for us to add the required GATT services for advertising the Carbon Dioxide sensor so that other devices can get those values.

First, we'll define the GATT Services in `apps/air_quality/src/bleprph.h`. Be sure to include the header files as well.

```
#include "host/ble_hs.h"
#include "host/ble_uuid.h"

...
/* Sensor Data */
/* e761d2af-1c15-4fa7-af80-b5729002b340 */
static const ble_uuid128_t gatt_svr_svc_co2_uuid =
    BLE_UUID128_INIT(0x40, 0xb3, 0x20, 0x90, 0x72, 0xb5, 0x80, 0xaf,
                      0xa7, 0x4f, 0x15, 0x1c, 0xaf, 0xd2, 0x61, 0xe7);
#define CO2_SNS_TYPE          0xDEAD
#define CO2_SNS_STRING "SenseAir K30 CO2 Sensor"
#define CO2_SNS_VAL           0xBEAD

uint16_t gatt_co2_val;
```

You can use any hex values you choose for the sensor type and sensor values, and you can even forget the sensor type and sensor string definitions altogether but they make the results look nice in our Bluetooth App for Mac OS X and iOS.

Next we'll add those services to `apps/air_quality/src/gatt_svr.c`.

```
static int
gatt_svrsns_access(uint16_t conn_handle, uint16_t attr_handle,
                    struct ble_gatt_access_ctxt *ctxt,
                    void *arg);

static uint16_t gatt_co2_val_len;
```

Make sure it is added as *primary* service.

```
static const struct ble_gatt_svc_def gatt_svr_svcs[] = {
{
    /*** Service: Security test. */
    .type = BLE_GATT_SVC_TYPE_PRIMARY,
    .uuid = &gatt_svr_svc_sec_test_uuid.u,
    .characteristics = (struct ble_gatt_chr_def[]) { {
        /*** Characteristic: Random number generator. */
        .uuid = &gatt_svr_chr_sec_test_rand_uuid.u,
        .access_cb = gatt_svr_chr_access_sec_test,
        .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_READ_ENC,
    }, {
        /*** Characteristic: Static value. */
        .uuid = &gatt_svr_chr_sec_test_static_uuid.u
        .access_cb = gatt_svr_chr_access_sec_test,
        .flags = BLE_GATT_CHR_F_READ |
                  BLE_GATT_CHR_F_WRITE | BLE_GATT_CHR_F_WRITE_ENC,
    }, {
        0, /* No more characteristics in this service. */
    } },
},
{
    /*** CO2 Level Notification Service. */
    .type = BLE_GATT_SVC_TYPE_PRIMARY,
    .uuid = &gatt_svr_svc_co2_uuid.u,
    .characteristics = (struct ble_gatt_chr_def[]) { {
```

(continues on next page)

(continued from previous page)

```

        .uuid = BLE_UUID16_DECLARE(CO2_SNS_TYPE),
        .access_cb = gatt_svr_sns_access,
        .flags = BLE_GATT_CHR_F_READ,
    }, {
        .uuid = BLE_UUID16_DECLARE(CO2_SNS_VAL),
        .access_cb = gatt_svr_sns_access,
        .flags = BLE_GATT_CHR_F_NOTIFY,
    }, {
        0, /* No more characteristics in this service. */
    } },
},
{
    0, /* No more services. */
},
};

```

Next we need to tell the GATT Server how to handle requests for CO2 readings :

```

static int
gatt_svr_sns_access(uint16_t conn_handle, uint16_t attr_handle,
                     struct ble_gatt_access_ctxt *ctxt,
                     void *arg)
{
    uint16_t uuid16;
    int rc;

    uuid16 = ble_uuid_u16(ctxt->chr->uuid);

    switch (uuid16) {
    case CO2_SNS_TYPE:
        assert(ctxt->op == BLE_GATT_ACCESS_OP_READ_CHR);
        rc = os_mbuf_append(ctxt->om, CO2_SNS_STRING, sizeof CO2_SNS_STRING);
        BLEPRPH_LOG(INFO, "CO2 SENSOR TYPE READ: %s\n", CO2_SNS_STRING);
        return rc == 0 ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;

    case CO2_SNS_VAL:
        if (ctxt->op == BLE_GATT_ACCESS_OP_WRITE_CHR) {
            rc = gatt_svr_chr_write(ctxt->om, 0,
                                    sizeof gatt_co2_val,
                                    &gatt_co2_val,
                                    &gatt_co2_val_len);
            return rc;
        } else if (ctxt->op == BLE_GATT_ACCESS_OP_READ_CHR) {
            rc = os_mbuf_append(ctxt->om, &gatt_co2_val,
                                sizeof gatt_co2_val);
            return rc == 0 ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;
        }
    }

    default:
        assert(0);
        return BLE_ATT_ERR_UNLIKELY;
}

```

(continues on next page)

(continued from previous page)

```

    }
}
```

Now it's time to go into our `apps/air_quality/src/main.c` and change how we read CO2 readings and respond to requests.

We'll need a task handler with an event queue for the CO2 readings.

```

/* CO2 Task settings */
#define CO2_TASK_PRIO          128
#define CO2_STACK_SIZE         (OS_STACK_ALIGN(336))
struct os_eventq co2_evq;
struct os_task co2_task;
bssnz_t os_stack_t co2_stack[CO2_STACK_SIZE];
```

And of course we'll need to go to our `main()` and do all the standard task and event setup we normally do by adding the following:

```

/* Initialize sensor eventq */
os_eventq_init(&co2_evq);

/* Create the CO2 reader task.
 * All sensor reading operations are performed in this task.
 */
os_task_init(&co2_task, "sensor", co2_task_handler,
             NULL, CO2_TASK_PRIO, OS_WAIT_FOREVER,
             co2_stack, CO2_STACK_SIZE);
```

We'll also need to add a task handler – since we initialized it above:

```

/**
 * Event loop for the sensor task.
 */
static void
co2_task_handler(void *unused)
{
    while (1) {
        co2_read_event();
        /* Wait 2 second */
        os_time_delay(OS_TICKS_PER_SEC * 2);

    }
}
```

And finally, we'll take care of that `co2_read_event()` function:

```

int
co2_read_event(void)
{
    int value;
    enum senseair_read_type type = SENSEAIR_CO2;
    uint16_t chr_val_handle;
    int rc;
```

(continues on next page)

(continued from previous page)

```

value = senseair_read(type);
if (value >= 0) {
    console_printf("Got %d\n", value);
} else {
    console_printf("Error while reading: %d\n", value);
    goto err;
}
gatt_co2_val = value;
rc = ble_gatts_find_chr(&gatt_svr_svc_co2_uuid.u, BLE_UUID16_DECLARE(CO2_SNS_VAL),  

    ↵NULL, &chr_val_handle);
assert(rc == 0);
ble_gatts_chr_updated(chr_val_handle);
return (0);
err:
    return (rc);
}

```

This one simply reads and updates the CO2 value and sends that over BLE to any connected clients instead.

We can now build, create-image and load the app onto our nRF52DK board, and then connect and see the updated values! To view the results over Bluetooth, you can use LightBlue or any other application that can connect to, and read, Bluetooth data. By default, the device will show up as nimble-bleprph, since we used the bleprph app as our template. I've changed mine to something a bit more applicable: BLE CO2 Sensor.

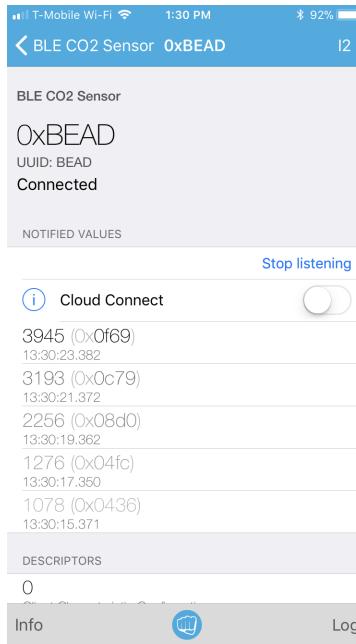


Fig. 7: LightBlue app connected to BLE CO2 Sensor

Congratulations!!

4.10.7 Adding an Analog Sensor on nRF52

We will be adding an analog sensor to the NRF52DK development board and using the Analog to Digital Converter (ADC) to read the values from the sensor. It's also using Bluetooth to allow you to connect to the app and read the value of the sensor.

- *Required Hardware*
- *Create a Project*
- *Building a Driver*
- *Building the BLE Services*
- *Adding the eTape Water Sensor*
- *View Data Remotely via Bluetooth*
- *Conclusion*

Required Hardware

- nRF52 Development Kit (one of the following)
 - Dev Kit from Nordic - PCA 10040
 - Eval Kit from Rigado - BMD-300-EVAL-ES
- eTape Liquid Sensor – buy from [Adafruit](#)
- Laptop running Mac OS
- It is assumed you have already installed newt tool.
- It is assumed you already installed native tools as described *here*

Create a Project

Create a new project to hold your work. For a deeper understanding, you can read about project creation in [Get Started – Creating Your First Project](#) or just follow the commands below.

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new myadc
Downloading project skeleton from apache/mynewt-blinky...
Downloading repository mynewt-blinky (commit: master) ...
Installing skeleton in myadc...
Project myadc successfully created.
$ cd myadc
```

Change the Repo Version in project.yml

Recently, members of the Mynewt community have contributed great efforts to integrate the nrfx drivers, but to take advantage of such resources, we will need to use the master of the Mynewt repo (pre v1.4.0 release). To be set at the master of the Mynewt repo, we will need to change the repo version in our `project.yml` file. If you're not familiar with using repositories, see the section on [repositories](#) before continuing.

In your `project.yml` file, change the `vers` field under `repository.apache-mynewt-core` from `1-latest` to `0-dev`. When you're done, your `project.yml` file should look like this:

```
project.name: "my_project"

project.repositories:
    - apache-mynewt-core
    - mynewt_nordic

# Use github's distribution mechanism for core ASF libraries.
# This provides mirroring automatically for us.

repository.apache-mynewt-core:
    type: github
    vers: 0-dev
    user: apache
    repo: mynewt-core
```

If you already have newt previously installed, you can still change the repo version in your `project.yml` file, but you will need to run `newt upgrade` for the change to take effect.

Install Everything

Now that you have defined the needed repositories, it's time to install everything so that you can get started.

```
$ newt upgrade -v
Downloading repository mynewt-core (commit: master) ...
Downloading repository mynewt-nimble (commit: master) ...
Downloading repository mynewt-mcumgr (commit: master) ...
Downloading repository mcuboot (commit: master) ...
apache-mynewt-core successfully installed version 0.0.0
apache-mynewt-nimble successfully installed version 0.0.0
apache-mynewt-mcumgr successfully installed version 0.0.0
mcuboot successfully installed version 0.0.0
```

Create the App and Targets

For the sensor app we will be building and modifying on top of the `bleprph` app so that we get the Bluetooth communications built in. The first thing we'll need to do is copy that app into our own app directory:

```
$ mkdir -p apps/nrf52_adc
$ cp -Rp repos/apache-mynewt-core/apps/bleprph/* apps/nrf52_adc
```

Next, you'll modify the `pkg.yml` file for your app. Note the change in `pkg.name` and `pkg.description`. Also make sure that you specify the full path of all the packages with the prefix `@apache-mynewt-core/` as shown in the third

highlighted line.

```
$ cat apps/nrf52_adc/pkg.yml
...
pkg.name: apps/nrf52_adc
pkg.type: app
pkg.description: Simple BLE peripheral
application for ADC Sensors.
pkg.author: "Apache Mynewt <dev@mynewt.incubator.apache.org>"
pkg.homepage: "http://mynewt.apache.org/"
pkg.keywords:

pkg.deps:
- "@apache-mynewt-core/boot/split"
- "@mcuboot/boot/bootutil"
- "@apache-mynewt-core/kernel/os"
- "@apache-mynewt-core/mgmt/imgmgr"
- "@apache-mynewt-core/mgmt/newtmgr"
- "@apache-mynewt-core/mgmt/newtmgr/transport/ble"
- "@apache-mynewt-core/net/nimble/host"
- "@apache-mynewt-core/net/nimble/host/services/ans"
- "@apache-mynewt-core/net/nimble/host/services/gap"
- "@apache-mynewt-core/net/nimble/host/services/gatt"
- "@apache-mynewt-core/net/nimble/host/store/config"
- "@apache-mynewt-core/net/nimble/host/util"
- "@apache-mynewt-core/net/nimble/transport"
- "@apache-mynewt-core/sys/console/full"
- "@apache-mynewt-core/sys/log/full"
- "@apache-mynewt-core/sys/stats/full"
- "@apache-mynewt-core/sys/sysinit"
- "@apache-mynewt-core/sys/id"
```

Create two targets - one for the bootloader and one for the nrf52 board.

Note: The correct bsp must be chosen for the board you are using.

- For the Nordic Dev Kit choose @apache-mynewt-core/hw/bsp/nrf52dk for the bsp
- For the Rigado Eval Kit choose @apache-mynewt-core/hw/bsp/bmd300eval for the bsp

```
$ newt target create nrf52_adc
$ newt target set nrf52_adc app=apps/nrf52_adc
$ newt target set nrf52_adc bsp=@apache-mynewt-core/hw/bsp/nrf52dk
$ newt target set nrf52_adc build_profile=debug

$ newt target create nrf52_boot
$ newt target set nrf52_boot app=@mcuboot/boot/mynewt
$ newt target set nrf52_boot bsp=@apache-mynewt-core/hw/bsp/nrf52dk
$ newt target set nrf52_boot build_profile=optimized

$ newt target show
targets nrf52_adc
    app=apps/nrf52_adc
    bsp=@apache-mynewt-core/hw/bsp/nrf52dk
    build_profile=debug
```

(continues on next page)

(continued from previous page)

```
targets nrf52_boot
    app=@mcuboot/boot/mynewt
    bsp=@apache-mynewt-core/hw/bsp/nrf52dk
    build_profile=optimized
```

Note: If you've already built and installed a bootloader for your NRF52dk then you do not need to create a target, build and load it here.

Build the Target Executables

```
$ newt build nrf52_boot
...
Compiling boot.c
Archiving boot.a
Linking boot.elf
App successfully built: ~/dev/myadc/bin/nrf52_boot/boot/mynewt/mynewt.elf
```

```
$ newt build nrf52_adc
...
Compiling main.c
Archiving nrf52_adc.a
Linking nrf52_adc.elf
App successfully built: ~/dev/myadc/bin/nrf52\_\_adc/apps/nrf52_adc/nrf52_adc.elf
```

Sign and Create the nrf52_adc Application Image

You must sign and version your application image to download it using newt to the board. Use the `newt create-image` command to perform this action. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt create-image nrf52_adc 1.0.0
App image successfully generated: ~/dev/myadc/bin/nrf52_adc/apps/nrf52_adc/nrf52_adc.img
Build manifest: ~/dev/myadc/bin/nrf52_adc/apps/nrf52_adc/manifest.json
```

Connect the Board

Connect the evaluation board via micro-USB to your PC via USB cable.

Download to the Target

Download the bootloader first and then the nrf52_adc executable to the target platform. Don't forget to reset the board if you don't see the LED blinking right away!

```
$ newt load nrf52_boot
$ newt load nrf52_adc
```

Note: If you want to erase the flash and load the image again, you can use JLinkExe to issue an `erase` command.

```
$ JLinkExe -device nRF52 -speed 4000 -if SWD
SEGGER J-Link Commander
V5.12c (Compiled Apr 21 2016 16:05:51)
DLL version V5.12c, compiled Apr 21 2016 16:05:45

Connecting to J-Link via USB...O.K.
Firmware: J-Link
OB-SAM3U128-V2-NordicSemi compiled Mar 15 2016 18:03:17
Hardware version: V1.00
S/N: 682863966
VTref = 3.300V

Type "connect" to establish a target connection, '?' for help
J-Link>erase
Cortex-M4 identified.
Erasing device (0;?i?)...
Comparing flash [100%] Done.
Erasing flash [100%] Done.
Verifying flash [100%] Done.
J-Link: Flash download: Total time needed: 0.363s (Prepare: 0.093s, Compare: 0.000s, ↵
Erase: 0.262s, Program: 0.000
s, Verify: 0.000s, Restore: 0.008s)
Erasing done.
J-Link>exit
$
```

So you have a BLE app, but really all you've done is change the name of the **bleprph** app to **nrf52_adc** and load that. Not all that impressive, and it certainly won't read an Analog Sensor right now. So let's do that next. In order to read an ADC sensor, we'll create a driver for it here in our app that will leverage the existing nrfx driver. It adds another layer of indirection, but it will also give us a look at building our own driver, so we'll do it this way.

Building a Driver

The first thing to do is to create the directory structure for your driver:

```
$ mkdir -p libs/my_drivers/myadc/include/myadc
$ mkdir -p libs/my_drivers/myadc/src
```

Now you can add the files you need. You'll need a pkg.yml to describe the driver, and then header stub followed by source stub.

```
$ cat libs/my_drivers/myadc/pkg.yml
```

```
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements. See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership. The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License. You may obtain a copy of the License at
#
```

(continues on next page)

(continued from previous page)

```
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied. See the License for the
# specific language governing permissions and limitations
# under the License.
#
pkg.name: libs/my_drivers/myadc
pkg.deps:
  - "@apache-mynewt-core/hw/hal"
```

First, let's create the required header file `myadc.h` in the include directory (i.e. `libs/my_drivers/myadc/include/myadc/myadc.h`). It's a pretty straightforward header file, since we only need to do 2 things:

- Initialize the ADC device
- Read ADC Values

```
#ifndef _NRF52_ADC_H_
#define _NRF52_ADC_H_

void * adc_init(void);
int adc_read(void *buffer, int buffer_len);

#endif /* _NRF52_ADC_H_ */
```

Next we'll need a corresponding source file `myadc.c` in the `src` directory. This is where we'll implement the specifics of the driver:

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

/* ADC */
#include "myadc/myadc.h"
#include "nrf.h"
#include <adc/adc.h>

/* Nordic headers */
#include <nrfx.h>
#include <nrf_saadc.h>
#include <nrfx_saadc.h>
#include <nrfx_config.h>

#define ADC_NUMBER_SAMPLES (2)
#define ADC_NUMBER_CHANNELS (1)

nrfx_saadc_config_t adc_config = NRFX_SAADC_DEFAULT_CONFIG;

struct adc_dev *adc;
uint8_t *sample_buffer1;
```

(continues on next page)

(continued from previous page)

```

uint8_t *sample_buffer2;

void *
adc_init(void)
{
    nrf_saadc_channel_config_t cc = NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_
→AIN1);
    cc.gain = NRF_SAADC_GAIN1_6;
    cc.reference = NRF_SAADC_REFERENCE_INTERNAL;
    adc = (struct adc_dev *) os_dev_open("adc0", 0, &adc_config);
    assert(adc != NULL);
    adc_chan_config(adc, 0, &cc);
    sample_buffer1 = malloc(adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_SAMPLES));
    sample_buffer2 = malloc(adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_SAMPLES));
    memset(sample_buffer1, 0, adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_
→SAMPLES));
    memset(sample_buffer2, 0, adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_
→SAMPLES));
    adc_buf_set(adc, sample_buffer1, sample_buffer2,
                adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_SAMPLES));
    return adc;
}

int
adc_read(void *buffer, int buffer_len)
{
    int i;
    int adc_result;
    int my_result_mv = 0;
    int rc;
    for (i = 0; i < ADC_NUMBER_SAMPLES; i++) {
        rc = adc_buf_read(adc, buffer, buffer_len, i, &adc_result);
        if (rc != 0) {
            goto err;
        }
        my_result_mv = adc_result_mv(adc, 0, adc_result);
    }
    adc_buf_release(adc, buffer, buffer_len);
    return my_result_mv;
err:
    return (rc);
}

```

There's a lot going on in here, so let's walk through it step by step.

First, we don't need to define a default configuration for our ADC - this has already been created. So we initialize the ADC in `adc_init()`:

```

void *
adc_init(void)
{
    nrf_saadc_channel_config_t cc = NRFX_SAADC_DEFAULT_CHANNEL_CONFIG_SE(NRF_SAADC_INPUT_
→AIN1);

```

(continues on next page)

(continued from previous page)

```

cc.gain = NRF_SAADC_GAIN1_6;
cc.reference = NRF_SAADC_REFERENCE_INTERNAL;
adc = (struct adc_dev *) os_dev_open("adc0", 0, &adc_config);
assert(adc != NULL);
adc_chan_config(adc, 0, &cc);
sample_buffer1 = malloc(adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_SAMPLES));
sample_buffer2 = malloc(adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_SAMPLES));
memset(sample_buffer1, 0, adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_
SAMPLES));
memset(sample_buffer2, 0, adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_
SAMPLES));
adc_buf_set(adc, sample_buffer1, sample_buffer2,
            adc_buf_size(adc, ADC_NUMBER_CHANNELS, ADC_NUMBER_SAMPLES));
return adc;
}

```

A few things need to be said about this part, as it is the most confusing. First, we're using a **default** configuration for the ADC Channel via the `NRF_SAADC_DEFAULT_CHANNEL_CONFIG_SE` macro. The important part here is that we're actually using AIN1. I know what you're thinking, "But we want ADC-0!" and that's true. The board is actually labelled 'A0, A1, A2' etc., and the actual pin numbers are also listed on the board, which seems handy. At first. But it gets messy very quickly.

If you try to use AIN0, and then go poke around in the registers while this is running,

```
(gdb) p/x {NRF_SAADC_Type}0x40007000
...
CH = {{PSEL_P = 0x1,
       PSEL_N = 0x0,
       CONFIG = 0x20000,
       LIMIT = 0x7ffff8000
      },
      
```

You'll see that the pin for channel 0 is set to 1, which corresponds to AIN0, but that's **NOT** the same as A0 – pin P0.03, the one we're using. For that, you use AIN1, which would set the pin value to 2. Messy. Someone, somewhere, thought this made sense.

The only other thing to note here is that we're using the internal reference voltage, rather than setting our own. There's nothing wrong with that, but since we are, we'll have to crank up the gain a bit by using `NRF_SAADC_GAIN1_6`.

Then, in `adc_read()` we will take readings, convert the raw readings to a millivolt equivalent, and return the result.

```

int
adc_read(void *buffer, int buffer_len)
{
    int i;
    int adc_result;
    int my_result_mv = 0;
    int rc;
    for (i = 0; i < ADC_NUMBER_SAMPLES; i++) {
        rc = adc_buf_read(adc, buffer, buffer_len, i, &adc_result);
        if (rc != 0) {
            goto err;
        }
    }
}

```

(continues on next page)

(continued from previous page)

```

    my_result_mv = adc_result_mvadc, 0, adc_result);
}
adc_buf_releaseadc, buffer, buffer_len);
return my_result_mv;
err:
return (rc);
}

```

Finally, we'll need to enable the ADC, which is disabled by default. To override this setting, we need to add a `syscfg.yml` file to our `nrf52_adc` target:

```
$ cat targets/nrf52_adc/syscfg.yml
syscfg.vals:
# Enable ADC 0
ADC_0: 1
```

Once that's all done, you should have a working ADC Driver for your nRF52dk board. The last step in getting the driver set up is to include it in the package dependency defined by `pkg.deps` in the `pkg.yml` file of your app. You will also need to add the `mcu/nordic` package. Add them in `apps/nrf52_adc/pkg.yml` as shown below:

```

# Licensed to the Apache Software Foundation (ASF) under one
# <snip>

pkg.name: apps/nrf52_adc
pkg.type: app
pkg.description: Simple BLE peripheral application for ADC sensor.
pkg.author: "Apache Mynewt <dev@mynewt.incubator.apache.org>"
pkg.homepage: "http://mynewt.apache.org/"
pkg.keywords:

pkg.deps:
- "@apache-mynewt-core/boot/split"
- "@mcuboot/boot/bootutil"
- "@apache-mynewt-core/kernel/os"
- "@apache-mynewt-core/mgmt/imgmgr"
- "@apache-mynewt-core/mgmt/newtmgr"
- "@apache-mynewt-core/mgmt/newtmgr/transport/ble"
- "@apache-mynewt-core/net/nimble/host"
- "@apache-mynewt-core/net/nimble/host/services/ans"
- "@apache-mynewt-core/net/nimble/host/services/gap"
- "@apache-mynewt-core/net/nimble/host/services/gatt"
- "@apache-mynewt-core/net/nimble/host/store/config"
- "@apache-mynewt-core/net/nimble/host/util"
- "@apache-mynewt-core/net/nimble/transport"
- "@apache-mynewt-core/sys/console/full"
- "@apache-mynewt-core/sys/log/full"
- "@apache-mynewt-core/sys/stats/full"
- "@apache-mynewt-core/sys/sysinit"
- "@apache-mynewt-core/sys/id"
- "@apache-mynewt-core/hw/mcu/nordic"
- "libs/my_drivers/myadc"
```

Creating the ADC Task

Now that the driver is done, we'll need to add calls to the main app's `main.c` file, as well as a few other things. First, we'll need to update the includes, and add a task for our ADC sampling.

```
#include <adc/adc.h>
...
#include "myadc/myadc.h"
...
/* ADC Task settings */
#define ADC_TASK_PRIO      5
#define ADC_STACK_SIZE     (OS_STACK_ALIGN(336))
struct os_eventq adc_evq;
struct os_task adc_task;
bssnz_t os_stack_t adc_stack[ADC_STACK_SIZE];
```

Next we'll need to initialize the task `event_q` so we'll add the highlighted code to `main()` as shown below. You can also change the name of your Bluetooth device in `ble_svc_gap_device_name_set`:

```
/* Set the default device name. */
rc = ble_svc_gap_device_name_set("nimble-adc");
assert(rc == 0);

#if MYNEWT_VAL(BLEPRPH_LE_PHY_SUPPORT)
    phy_init();
#endif

conf_load();

/* Initialize adc sensor task eventq */
os_eventq_init(&adc_evq);

/* Create the ADC reader task.
 * All sensor operations are performed in this task.
 */
os_task_init(&adc_task, "sensor", adc_task_handler,
             NULL, ADC_TASK_PRIO, OS_WAIT_FOREVER,
             adc_stack, ADC_STACK_SIZE);
```

We'll need that `adc_task_handler()` function to exist, and that's where we'll initialize the ADC Device and set the event handler. In the task's `while()` loop, we'll just make a call to ```adc_sample()``` to cause the ADC driver to sample the adc device.

```
/***
 * Event loop for the sensor task.
 */
static void
adc_task_handler(void *unused)
{
    struct adc_dev *adc;
    int rc;
    /* ADC init */
    adc = adc_init();
    rc = adc_event_handler_set(adc, adc_read_event, (void *) NULL);
```

(continues on next page)

(continued from previous page)

```

assert(rc == 0);

while (1) {
    adc_sample(adc);
    /* Wait 2 second */
    os_time_delay(OS_TICKS_PER_SEC * 2);
}
}

```

Above the adc_task_handler, add code to handle the adc_read_event() calls:

```

int
adc_read_event(struct adc_dev *dev, void *arg, uint8_t etype,
               void *buffer, int buffer_len)
{
    int value;
    uint16_t chr_val_handle;
    int rc;

    value = adc_read(buffer, buffer_len);
    if (value >= 0) {
        console_printf("Got %d\n", value);
    } else {
        console_printf("Error while reading: %d\n", value);
        goto err;
    }
    gatt_adc_val = value;
    rc = ble_gatts_find_chr(&gatt_svr_svc_adc_uuid.u, BLE_UUID16_DECLARE(ADC_SNS_VAL),  

                           NULL, &chr_val_handle);
    assert(rc == 0);
    ble_gatts_chr_updated(chr_val_handle);
    return (0);
err:
    return (rc);
}

```

This is where we actually read the ADC value and then update the BLE Characteristic for that value.

But wait, we haven't defined those BLE services and characteristics yet! Right, so don't try to build and run this app just yet or it will surely fail. Instead, move on to the next section and get all of those services defined.

Building the BLE Services

If the nrf52_adc app is going to be a Bluetooth-enabled sensor app that will allow you to read the value of the eTape Water Level Sensor via Bluetooth we'll need to actually define those Services and Characteristics.

As with the *ble peripheral* app, we will advertise a couple of values from our app. The first is not strictly necessary, but it will help us build an iOS app later. We've defined a service and the characteristics in that service in bleprph.h in the apps/nrf52_adc/src/ directory. Make sure to include the host/ble_uuid.h header:

```

#include "host/ble_uuid.h"
...
/* Sensor Data */

```

(continues on next page)

(continued from previous page)

```
/* e761d2af-1c15-4fa7-af80-b5729002b340 */
static const ble_uuid128_t gatt_svr_svc_adc_uuid =
    BLE_UUID128_INIT(0x40, 0xb3, 0x20, 0x90, 0x72, 0xb5, 0x80, 0xaf,
                      0xa7, 0x4f, 0x15, 0x1c, 0xaf, 0xd2, 0x61, 0xe7);
#define ADC_SNS_TYPE          0xDEAD
#define ADC_SNS_STRING "eTape Liquid Level Sensor"
#define ADC_SNS_VAL            0xBEAD
uint16_t gatt_adc_val;
```

The first is the UUID of the service, followed by the 2 characteristics we are going to offer. The first characteristic is going to advertise the *type* of sensor we are advertising, and it will be a read-only characteristic. The second characteristic will be the sensor value itself, and we will allow connected devices to ‘subscribe’ to it in order to get constantly-updated values.

Note: You can choose any valid Characteristic UUIDs to go here. We’re using these values for illustrative purposes only.

The value that we’ll be updating is also defined here as `gatt_adc_val`.

If we then go look at `gatt_svr.c` we can see the structure of the service and characteristic offering that we set up:

```
static const struct ble_gatt_svc_def gatt_svr_svcs[] = {
{
    /*** Service: Security test. */
    .type = BLE_GATT_SVC_TYPE_PRIMARY,
    .uuid = &gatt_svr_svc_sec_test_uuid.u,
    .characteristics = (struct ble_gatt_chr_def[]) { {
        /*** Characteristic: Random number generator. */
        .uuid = &gatt_svr_chr_sec_test_rand_uuid.u,
        .access_cb = gatt_svr_chr_access_sec_test,
        .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_READ_ENC,
    }, {
        /*** Characteristic: Static value. */
        .uuid = &gatt_svr_chr_sec_test_static_uuid.u,
        .access_cb = gatt_svr_chr_access_sec_test,
        .flags = BLE_GATT_CHR_F_READ |
                 BLE_GATT_CHR_F_WRITE | BLE_GATT_CHR_F_WRITE_ENC,
    }, {
        0, /* No more characteristics in this service. */
    } },
},
{
    /*** ADC Level Notification Service. */
    .type = BLE_GATT_SVC_TYPE_PRIMARY,
    .uuid = &gatt_svr_svc_adc_uuid.u,
    .characteristics = (struct ble_gatt_chr_def[]) { {
        .uuid = BLE_UUID16_DECLARE(ADC_SNS_TYPE),
        .access_cb = gatt_svr_sns_access,
        .flags = BLE_GATT_CHR_F_READ,
    }, {
        .uuid = BLE_UUID16_DECLARE(ADC_SNS_VAL),
        .access_cb = gatt_svr_sns_access,
        .flags = BLE_GATT_CHR_F_NOTIFY,
    }, {

```

(continues on next page)

(continued from previous page)

```

    0, /* No more characteristics in this service. */
} },
},
{
    0, /* No more services. */
},
};

```

You should recognize the first services from the [BLE Peripheral](#) tutorial earlier. We're just adding another Service, with 2 new Characteristics, to that application.

We'll need to fill in the function that will be called for this service, `gatt_svr_sns_access` next so that the service knows what to do.

```

static int
gatt_svr_sns_access(uint16_t conn_handle, uint16_t attr_handle,
                     struct ble_gatt_access_ctxt *ctxt,
                     void *arg)
{
    uint16_t uuid16;
    int rc;

    uuid16 = ble_uuid_u16(ctxt->chr->uuid);

    switch (uuid16) {
        case ADC_SNS_TYPE:
            assert(ctxt->op == BLE_GATT_ACCESS_OP_READ_CHR);
            rc = os_mbuf_append(ctxt->om, ADC_SNS_STRING, sizeof ADC_SNS_STRING);
            BLEPRPH_LOG(INFO, "ADC SENSOR TYPE READ: %s\n", ADC_SNS_STRING);
            return rc == 0 ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;

        case ADC_SNS_VAL:
            if (ctxt->op == BLE_GATT_ACCESS_OP_WRITE_CHR) {
                rc = gatt_svr_chr_write(ctxt->om, 0,
                                         sizeof gatt_adc_val,
                                         &gatt_adc_val,
                                         NULL);
                return rc;
            } else if (ctxt->op == BLE_GATT_ACCESS_OP_READ_CHR) {
                rc = os_mbuf_append(ctxt->om, &gatt_adc_val,
                                    sizeof gatt_adc_val);
                return rc == 0 ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;
            }
        default:
            assert(0);
            return BLE_ATT_ERR_UNLIKELY;
    }
}

```

You can see that when request is for the `ADC_SNS_TYPE`, we return the Sensor Type we defined earlier. If the request if for `ADC_SNS_VAL` we'll return the `gatt_adc_val` value.

If you build, load and run this application now, you will see all those Services and Characteristics advertised, and you will even be able to read the “Sensor Type” String via the ADC_SNS_TYPE Characteristic.

Adding the eTape Water Sensor

Now that we have a fully functioning BLE App that we can subscribe to sensor values from, it's time to actually wire up the sensor!

As previously mentioned, we're going to be using an eTape Water Level Sensor. You can get one from [Adafruit](#).

We're going to use the sensor as a resistive sensor, and the setup is very simple. I'll be using a breadboard to put this all together for illustrative purposes. First, attach a jumper-wire from V_{DD} on the board to the breadboard. Next, attach a jumper wire from pin P0.03 on the board to the breadboard. This will be our ADC-in. The sensor should have come with a 560 ohm resistor, so plug that into the board between V_{DD} and ADC-in holes. Finally, attach a jumper from GND on the board to your breadboard. At this point, your breadboard should look like this:

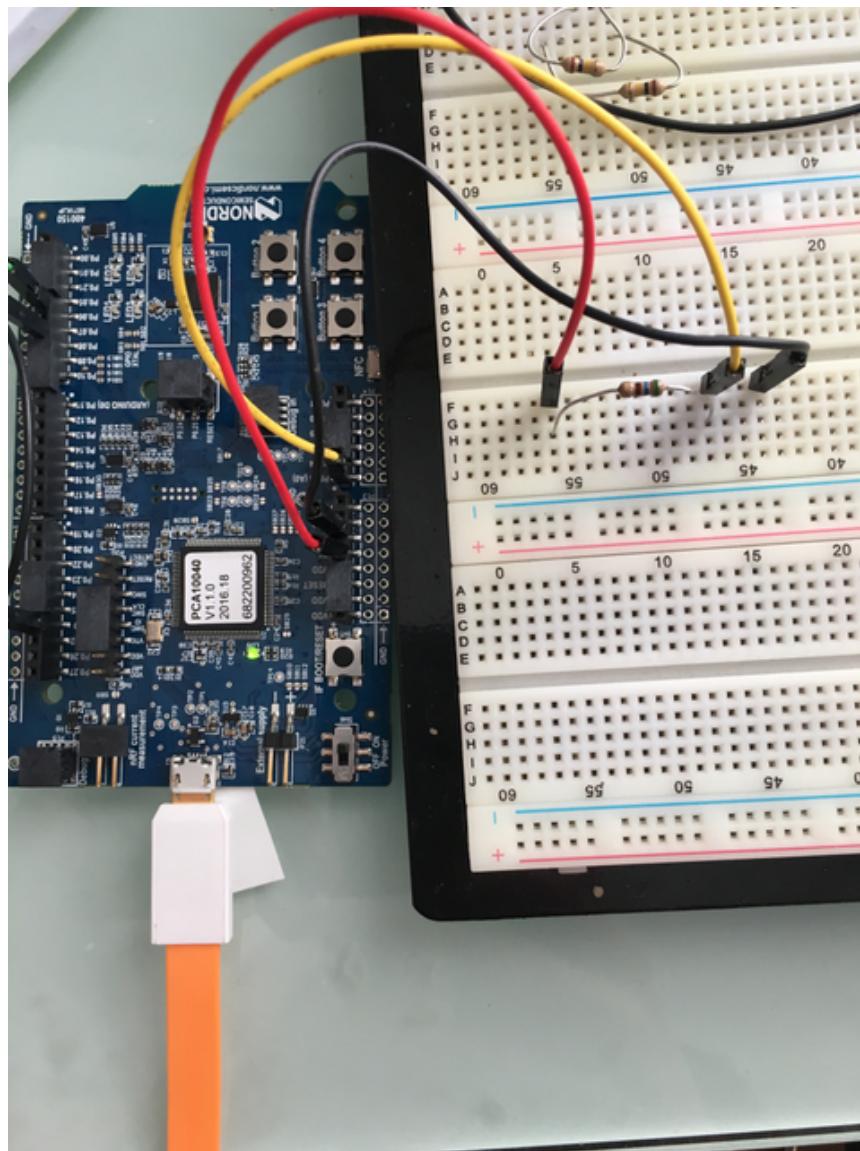


Fig. 8: Bread Board Setup

Now attach one of the middle 2 leads from the sensor to ground on the breadboard and the other middle lead to the ADC-in on the breadboard. Your breadboard should now look like this:

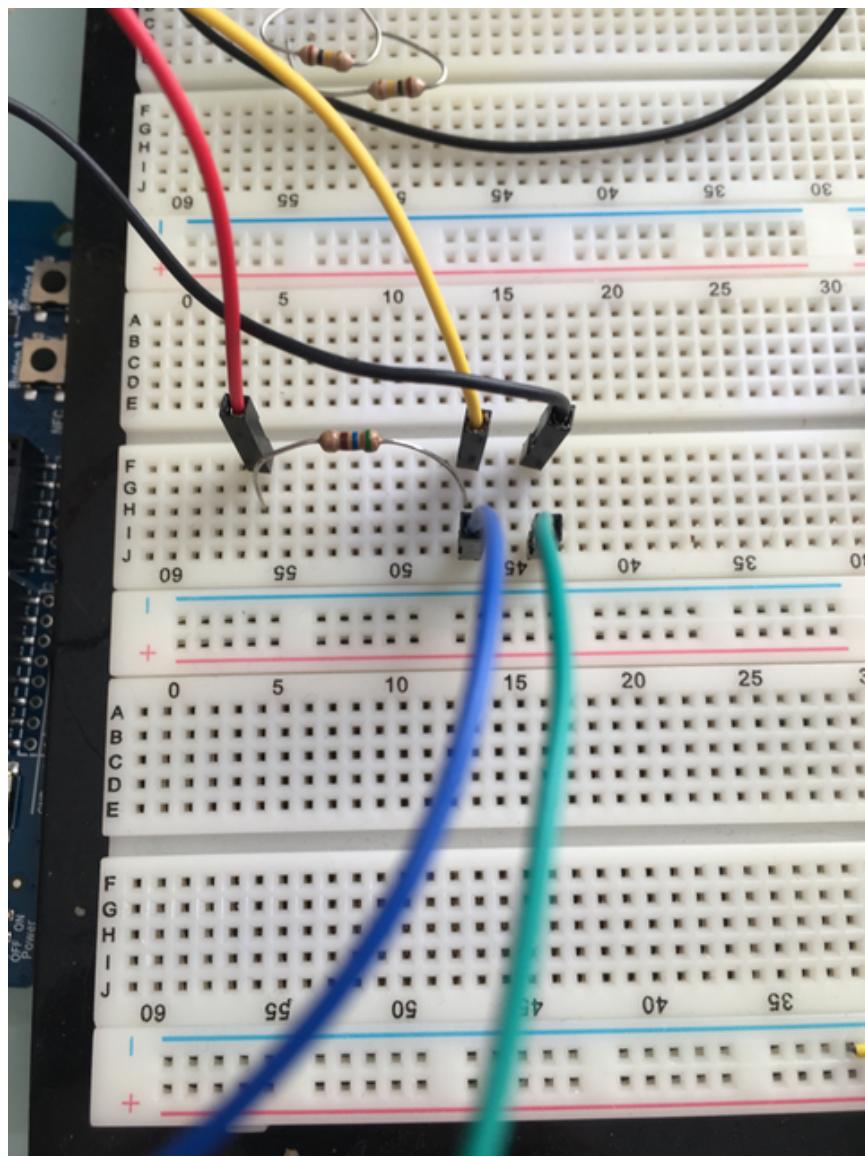


Fig. 9: Bread Board Final

And your eTap Sensor should look like this (at least if you have it mounted in a graduated cylinder as I do).

That concludes the hardware portion. Easy!

At this point you should be able to build, create-image and load your application and see it properly sending readings.

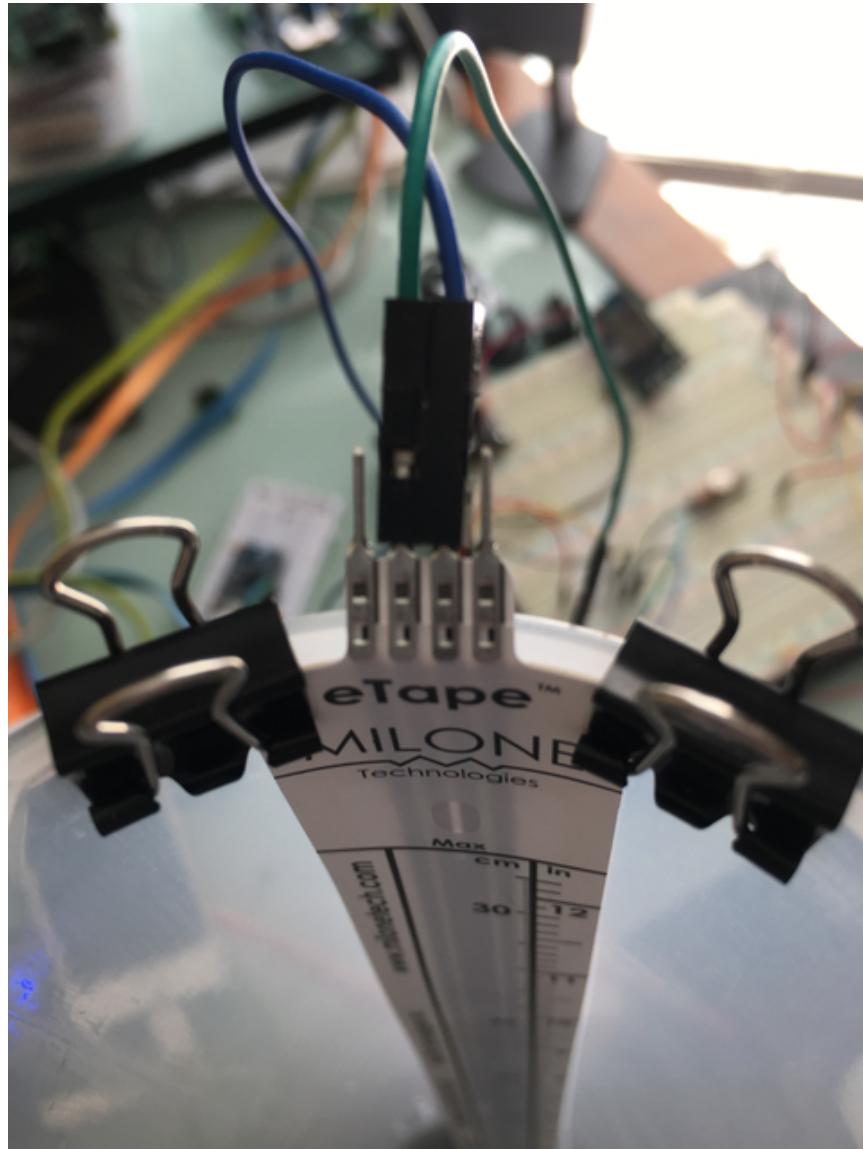


Fig. 10: eTape Sensor Setup

View Data Remotely via Bluetooth

To view these sensor readings via Bluetooth, you can use LightBlue or a similar app that can connect to Bluetooth devices and read data. Once your sensor application is running, you should see your device show up in LightBlue as `nimble-adc` (or whatever you named your Bluetooth device):

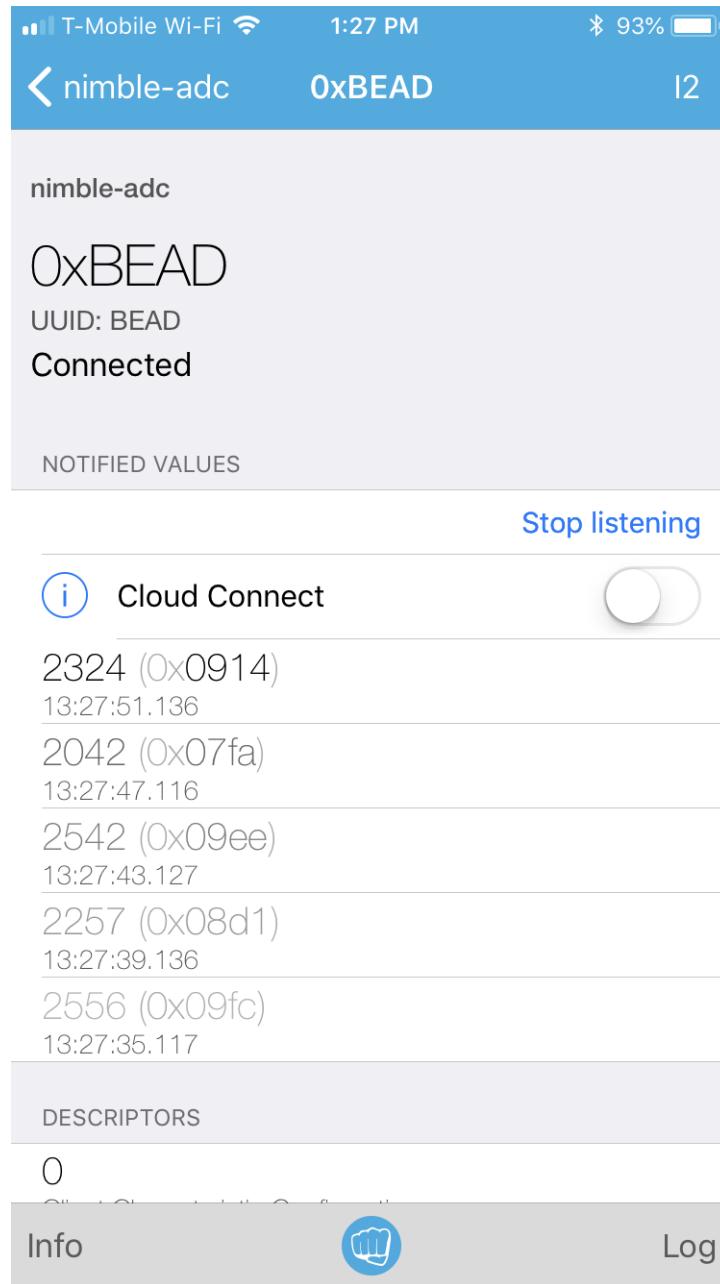


Fig. 11: LightBlue connected to the ADC app

Conclusion

Congratulations, you've now completed both a hardware project and a software project by connecting a sensor to your device and using Mynewt to read data from that sensor and send it via Bluetooth to a connected device. That's no small feat!

If you see anything missing or want to send us feedback, please do so by signing up for appropriate mailing lists on our [Community Page](#).

Keep on hacking and sensing!

Enjoy!

4.10.8 Enabling and Calibrating Off-Board DRV2605 LRA Actuator

This tutorial shows you how to run the `sensors_test` application on an nRF52-DK board to communicate, via the I2C interface, with the [Adafruit DRV2605](#) device. The DRV2605 driver includes the `drv2605` shell command that allows you to quickly view the status, calibrate and actuate your motor in preparation for using it programmatically in your own app.

This tutorial shows you how to:

- Create and build the application and bootloader targets.
- Connect a DRV2605 actuator device to an nRF52-DK board.
- Run `drv2605` shell commands to view the actuator data and control the drv2605 device.

- *Prerequisites*
- *Step 1: Creating the Application Target*
- *Step 2: Creating an Application Image and loading it*
- *Step 3: Communicating with and Calibrating the DRV2605 device*
- *Step 4: Actually Actuate*
- *Conclusion*

Prerequisites

- Meet the prerequisites listed in [Sensor Tutorials](#)
- Understand how to [add an offboard sensor](#), we won't be covering bootloader and application console connection here.
- Have a Nordic nRF52-DK board.
- Have an [Adafruit DRV2605](#) actuator and LRA motor.

Step 1: Creating the Application Target

In this step, you create a target for the sensors_test application that enables the DRV2605 off-board device.

To add the DRV2605 device support, you create the application target with the following syscfg settings enabled:

- I2C_0: Enables the I2C interface 0 in the nRF52 BSP HAL setting.
- DRV2605_OFB: Enables support for the DRV2605 actuator in the sensor creator package (hw/sensor/creator).

When this setting is enabled, the creator package performs the following:

- Includes the DRV2605 driver package (hw/drivers/drv2605) as a package dependency.
- Creates an os device for the device in the Mynewt kernel.
- Configures the device with default values.
- DRV2605_CLI: Enables the drv2605 shell command in the DRV2605 device driver package. The sensors_test application also uses this setting to conditionally include the call to the `drv2605_shell_init()` function to initialize the shell support in the driver.
- DRV2605_EN_PIN: The pin the driver needs to gpio in order to enable your DRV2605 device.
- DRV2605_RATED_VOLTAGE: I have no easy default for you here, it is entirely based on your purchased LRA motor. See 8.5.2.1 Rated Voltage Programming in the TI DRV2605 Datasheet or maybe talk to your motor supplier.
- DRV2605_OD_CLAMP: I have no easy default for you here, it is entirely based on your purchased LRA motor. See 8.5.2.2 Overdrive Voltage-Clamp Programming in the TI DRV2605 Datasheet or maybe talk to your motor supplier. NOTE: LRA and ERM (8) and (9) equations are swapped in the datasheet revision “SLOS854C REVISED SEPTEMBER 2014”
- DRV2605_DRIVE_TIME: I have no easy default for you here, it is entirely based on your purchased LRA motor. See 8.6.21, but generally it is computable with the equation $\text{LRA DRIVE_TIME} = (0.5 \times (1/\text{fLRA} * 1000) - 0.5 \text{ ms}) / 0.1 \text{ ms}$

1. Run the `newt target create` command, from your project base directory, to create the target. We name the target `nrf52_drv2605_test`:

```
$ newt target create nrf52_drv2605_test
Target targets/nrf52_drv2605_test successfully created
$
```

2. Run the `newt target set` command to set the app, bsp, and build_profile variables for the target:

```
$ newt target set nrf52_drv2605_test app=@apache-mynewt-core/apps/sensors_test_
↪bsp=@apache-mynewt-core/hw/bsp/nrf52dk build_profile=debug
Target targets/nrf52_drv2605_test successfully set target.app to @apache-mynewt-core/
↪apps/sensors_test
Target targets/nrf52_drv2605_test successfully set target.bsp to @apache-mynewt-core/hw/
↪bsp/nrf52dk
Target targets/nrf52_drv2605_test successfully set target.build_profile to debug
$
```

3. Run the `newt target set` command to enable the I2C_0, DRV2605_OFB, DRV2605_CLI, DRV2605_EN_PIN, DRV2605_RATED_VOLTAGE, DRV2605_OD_CLAMP, and DRV2605_DRIVE_TIME syscfg settings:

```
$ newt target set nrf52_drv2605_test syscfg=DRV2605_OFB=1:I2C_0=1:DRV2605_CLI=1:DRV2605_
↪EN_PIN=3:DRV2605_RATED_VOLTAGE=0x53:DRV2605_OD_CLAMP=0x69:DRV2605_DRIVE_TIME=20
Target targets/nrf52_drv2605_test successfully set target.syscfg to DRV2605_OFB=1:I2C_
↪_0=1:DRV2605_CLI=1:DRV2605_EN_PIN=3:DRV2605_RATED_VOLTAGE=0x53:DRV2605_OD_
↪_CLAMP=0x69:DRV2605_DRIVE_TIME=20

$
```

Step 2: Creating an Application Image and loading it

This tutorial assumes you have a functioning bootloader as taught in [add an offboard sensor](#) Now run the `newt create-image` command to create an image file. You may assign an arbitrary version (e.g. 1.0.0) to the image.

```
$ newt build nrf52_drv2605_test && newt create-image nrf52_drv2605_test 1.0.0
App image successfully generated: ~/dev/myproj/bin/targets/nrf52_drv2605_test/app/apps/
↪sensors_test/sensors_test.img
```

Step 3: Communicating with and Calibrating the DRV2605 device

This tutorial assumes you have a functioning application console as taught in [add an offboard sensor](#) The DRV2605 device driver implements the `drv2605` shell command that allows you to:

- Query the chip id, content of registers, calibrations.
- Reset the device.
- Change the power mode.
- Change the operation mode.
- Load waveforms to actuate.
- Actuate the device.

Example 1: Query the device chip id:

```
711273 compat> drv2605 chip_id
769056 0x07
```

Example 2: Run Diagnostics on your motor setup numbers:

```
827472 compat> drv2605 op_mode diag
drv2605 op_mode diag
829717 op_mode succeeded
```

If that didn't work or you will have to compute different `DRV2605_RATED_VOLTAGE`, `DRV2605_OD_CLAMP`, and `DRV2605_DRIVE_TIME` values and try again or maybe talk to your motor manufacturer or TI for more help.

Example 3: Run Calibration on your motor: Theres a lot more setup numbers you could enter here for the DRV2605 to figure out how to actuate your motor, but some of them it can figure out itself through auto calibration. Lets run autocalibration and then dump the fresh calibration numbers:

```
001407 compat> drv2605 op_mode cal
drv2605 op_mode cal
001931 op_mode succeeded
```

(continues on next page)

(continued from previous page)

```
drv2605 dump_cal
DRV2605_CALIBRATED_COMP: 0x09
DRV2605_CALIBRATED_BEMF: 0x79
DRV2605_CALIBRATED_BEMF_GAIN: 1
```

You could programmatically run this on every startup, but more likely you'd want to save these as in your syscfg.yml and restart. Presumably you'd never have to do these steps ever again.

Step 4: Actually Actuate

Now you're ready to (sigh) rumble. One way to use the DRV2605 device is to enable the ROM mode to use its stored patterns. Technically you don't need to do this after first configure as ROM mode is the default mode:

```
021773 compat> drv2605 op_mode rom
drv2605 op_mode rom
037245 op_mode succeeded
```

Now you can load up to 8 internal roms or delays. In this case we'll use four hard clicks (1) with max delays (255) in between. You may only have to do this once per boot if you wanted to use this same sequence every time you trigger the DRV2605 device.

```
120858 compat> drv2605 load_rom 1 255 1 255 1 255 1 255
drv2605 load_rom 1 255 1 255 1 255 1 255
122555 Load succeeded
```

The motor is in standby by default after a mode change, so enable it:

```
002111 compat> drv2605 power_mode active
drv2605 power_mode active
003263 power_mode succeeded
```

Now you can trigger those forms as many times as you want or load new forms and trigger again:

```
122555 compat> drv2605 trigger
drv2605 trigger
128806 Trigger succeeded
```

Conclusion

You've successfully enabled a mynewt application to communicate with a drv2605 device, calibrated it and actuated a motor! Next you'll want to look at the code comments on the drv2605.c file and how the drv2605_shell.c file is implemented so you can setup and actuate your device programmatically within your application.

4.11 Tooling

4.11.1 SEGGER RTT Console

Objective

Sometimes you don't have UART on your board, or you want to use it for something else while still having newt logs/shell capability. With SEGGER's RTT capability you can swap UART for RTT, which is a very high-speed memory-mapped I/O.

- *Hardware needed*
- *Setup the target*
- *Run the target executables*
- *Connect to console*

Hardware needed

You'll need a SEGGER J-Link programmer in order to use this advanced functionality. You might have an external J-Link programmer you're already using, or maybe your board has a dedicated J-Link onboard as some development kits do. Another possibility is J-Link OB firmware available for some devices like the micro:bit.

Setup the target

We'll assume you have an existing project with some kind of console/shell like *Blinky with console and shell* that we're switching over to RTT from UART.

Note: We have tested RTT with J-Link version V6.14h. We recommend that you upgrade your J-Link if you have an earlier version of J-Link installed. Earlier versions of J-Link use the BUFFER_SIZE_DOWN value defined in hw/drivers/rtt/include/rtt/SEGGER_RTT_Conf.h for the maximum number of input characters. If an input line exceeds the BUFFER_SIZE_DOWN number of characters, RTT ignores the extra characters. The default value is 16 characters. For example, this limit causes shell commands with more than 16 characters of input to fail. You may set the Mynewt RTT_BUFFER_SIZE_DOWN syscfg setting in your target to increase this value if you do not upgrade your J-Link version.

We can disable uart and enable rtt with the newt target command:

```
newt target amend nrf52_blinky syscfg=CONSOLE_UART=0
newt target amend nrf52_blinky syscfg=CONSOLE_RTT=1
```

Run the target executables

Now ‘run’ the newt target as you’ll need an active debugger process to attach to:

```
$ newt run nrf52_blinky
App image successfully generated: ~/Downloads/myapp1/bin/targets/nrf52_blinky/app/apps/
→blinky/blinky.img
Loading app image into slot 1
[~/Downloads/myapp1/repos/apache-mynewt-core/hw/bsp/nrf52-thingy/nrf52-thingy_debug.sh ~/-
→Downloads/myapp1/repos/apache-mynewt-core/hw/bsp/nrf52-thingy ~/Downloads/myapp1/bin/
→targets/nrf52_blinky/app/apps/blinky/blinky]
Debugging ~/Downloads/myapp1/bin/targets/nrf52_blinky/app/apps/blinky/blinky.elf
GNU gdb (GNU Tools for ARM Embedded Processors) 7.8.0.20150604-cvs
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-apple-darwin10 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ~/Downloads/myapp1/bin/targets/nrf52_blinky/app/apps/blinky/blinky.-
→elf...done.
0x000000d8 in ?? ()
Resetting target
0x000000dc in ?? ()
(gdb)
```

Connect to console

In a separate terminal window `telnet localhost 19021` and when you continue your gdb session you should see your output. If you’re not familiar with telnet, when you’re ready to exit you may do so by using the hotkey `ctrl+]` then typing `quit`

```
$ telnet localhost 19021
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying fe80::1...
telnet: connect to address fe80::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^].
SEGGER J-Link V6.14e - Real time terminal output
SEGGER J-Link EDU V8.0, SN=268006294
Process: JLinkGDBServer
```

Then you can interact with the device:

```
stat
stat
000262 Must specify a statistic name to dump, possible names are:
000262 stat
000262 compat>
```

4.11.2 SEGGER SystemView

Objective

With [SEGGER's SystemView](#) you can “record data from the target system while it is running. The recorded data is analyzed and the system behavior is visualized in different views.”

- *Hardware needed*
- *Software needed*
- *Setup the target*
- *Run the target executables*
- *Launch the app*

Hardware needed

You'll need a SEGGER J-Link programmer in order to use this advanced functionality. You might have an external J-Link programmer you're already using, or maybe your board has a dedicated J-Link onboard as some development kits do. Another possibility is J-Link OB firmware available for some devices like the micro:bit.

Software needed

- Download [SEGGER's SystemView](#) app.
- Copy the description file from sys/sysview/SYSVIEW_Mynewt.txt to the /Description/ directory of SystemView

Setup the target

We'll assume you have an existing example we're enabling SystemView on, in this case [blinky on nrf52](#). We can do so with the newt target amend command:

```
newt target amend blink_nordic syscfg=OS_SYSVIEW=1
```

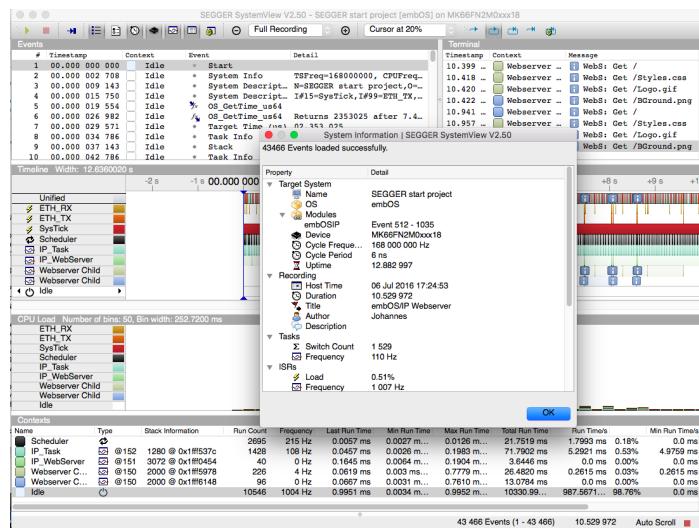
Run the target executables

Now ‘run’ the newt target as you’ll need an active debugger process to attach to:

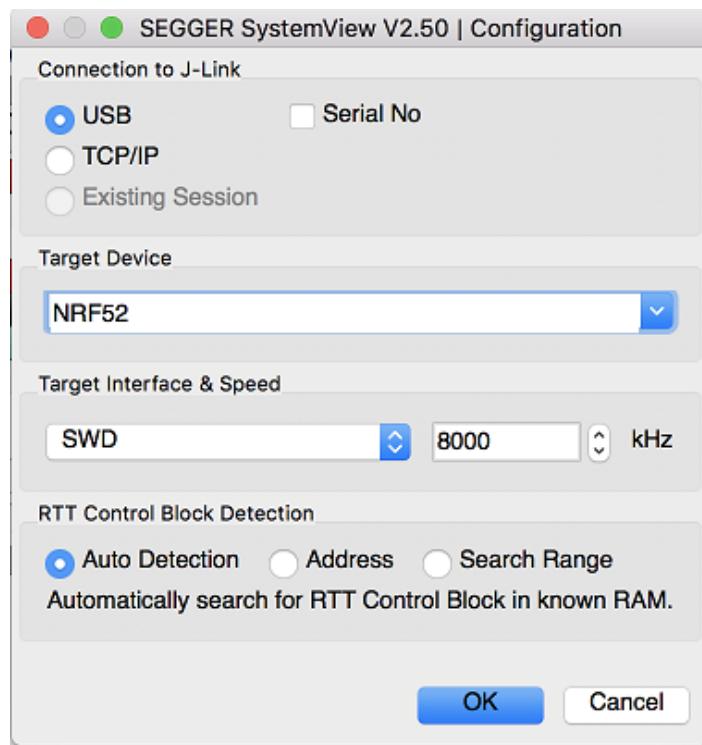
```
$ newt run blink_nordic 0
App image successfully generated: ~/Downloads/myproj/bin/targets/blink_nordic/app/apps/
→bleprph/bleprph.img
Loading app image into slot 1
[~/Downloads/myproj/repos/apache-mynewt-core/hw/bsp/nrf52-thingsy/nrf52-thingsy_debug.sh ~/
→Downloads/myproj/repos/apache-mynewt-core/hw/bsp/nrf52-thingsy ~/Downloads/myproj/bin/
→targets/blink_nordic/app/apps/bleprph/bleprph]
Debugging ~/Downloads/myproj/bin/targets/blink_nordic/app/apps/bleprph/bleprph.elf
GNU gdb (GNU Tools for ARM Embedded Processors) 7.8.0.20150604-cvs
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-apple-darwin10 --target=arm-none-eabi".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ~/Downloads/myproj/bin/targets/blink_nordic/app/apps/bleprph/
→bleprph.elf...done.
0x0000000d8 in ?? ()
Resetting target
0x0000000dc in ?? ()
```

Launch the app

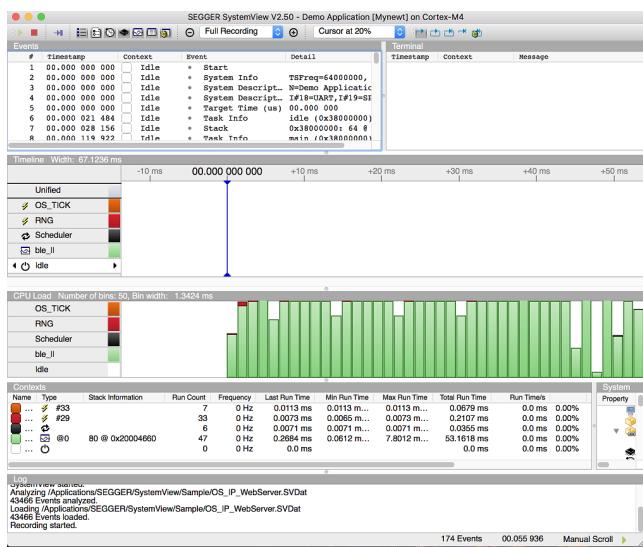
Launch the app and press **OK** in the System Information dialog box.



Select **Target > Start Recording** and press **OK** in the Configuration dialog box.



You should see the recording for your Mynewt application.



4.11.3 Diagnosing Errors

Here we list some of the tools Mynewt has available. This is by no means comprehensive. It does give an overview of some facilities more commonly used.

To create output for this demo, I'm including the ‘test/crash_test’ package while building the ‘slinky’ app.

Crash with Console

When system restarts due to an error, we always attempt to do it in a way which allows us to dump the current system state. Calls to assert(), error interrupts (regular faults, memory faults) both take this path.

The basic facility is the printout of registers to console.

Here is an example of what it looks like with Cortex-M4:

```
041088 Unhandled interrupt (3), exception sp 0x200001d28
041088 r0:0x00000000 r1:0x00017b61 r2:0x00000000 r3:0x0000002a
041088 r4:0x200001dc8 r5:0x00000000 r6:0x200034e4 r7:0x20003464
041088 r8:0x2000349c r9:0x20001f08 r10:0x00017bd4 r11:0x00000000
041088 r12:0x00000000 lr:0x00014e29 pc:0x00014e58 psr:0x61000000
041088 ICSR:0x00421803 HFSR:0x40000000 CFSR:0x02000000
041088 BFAR:0xe000ed38 MMFAR:0xe000ed34
```

Output includes the values of registers. The most interesting pieces would be \$pc (program counter), and \$lr (link register), which show the instruction where the fault happened, and the return address from that function.

You would then take these values, and match them against the image you’re running on the target.

For example:

```
[marko@IsMyLaptop:~/src2/incubator-mynewt-blinky]$ arm-elf-linux-gdb bin/targets/slinky_
˓→nrf52/app/apps/slinky/slinky.elf
GNU gdb (GDB) 7.8.1
...
Reading symbols from bin/targets/slinky_nrf52/app/apps/slinky/slinky.elf...done.
(gdb) list *0x00014e58
0x14e58 is in crash_device (repos/apache-mynewt-core/test/crash_test/src/crash_test.
˓→c:47).
42     if (!strcmp(how, "div0")) {
43
44         val1 = 42;
45         val2 = 0;
46
47         val3 = val1 / val2;
48         console_printf("42/0 = %d\n", val3);
49     } else if (!strcmp(how, "jump0")) {
50         ((void (*)())0)();
51     } else if (!strcmp(how, "ref0")) {
```

You can see that the system crashed due to divide-by-zero.

Crash with Verbose Location

We often call assert(), testing specific conditions which should be met for program execution to continue. If the condition fails, we reset with a message listing the instruction where assert() was called from.

```
829008 compat> Assert @ 0x14e9f  
831203 Unhandled interrupt (2), exception sp 0x20001d20
```

This address (here 0x14e9f) should then be used to find the line in the program where it happened. Note that this address is specific to your binary. So you have to do the search to locate the specific file/line using your .elf file.

You can also enable more verbose report by setting syscfg variable OS_CRASH_FILE_LINE to 1. The call to assert() from above would then look like so:

```
000230 compat> Assert @ 0x1503b - repos/apache-mynewt-core/test/crash_test/src/crash_  
→test.c:54  
001462 Unhandled interrupt (2), exception sp 0x20001d20
```

Note that this will increase the program text size, as we now need to store the names for the files which call assert().

Crash with Console with Stacktrace

Quite often you need a deeper view of the call chain to figure out how the program got to place where it crashed. Especially if the routine can be reached via multiple call chain routes.

You can enable a dump of possible call chain candidates by setting syscfg variable OS_CRASH_STACKTRACE to 1.

When crash happens, the dumping routine walks backwards in the stack, and prints any address which falls within text region. All these values are candidates of being part of the call chain. However, you need to filter it yourself to figure out which values are actually part of it, and which addresses just happen to be in the stack at that time.

Here is a sample output:

```
004067 Unhandled interrupt (3), exception sp 0x20001d28  
004067 r0:0x00000000 r1:0x00017c55 r2:0x00000000 r3:0x0000002a  
004067 r4:0x20001dc8 r5:0x00000000 r6:0x200034e4 r7:0x20003464  
004067 r8:0x2000349c r9:0x20001f08 r10:0x00017cc8 r11:0x00000000  
004067 r12:0x00000000 lr:0x00014ef9 pc:0x00014f28 psr:0x61000000  
004067 ICSR:0x00421803 HFSR:0x40000000 CFSR:0x02000000  
004067 BFAR:0xe000ed38 MMFAR:0xe000ed34  
004067 task:main  
004067 0x20001d64: 0x0001501b  
004067 0x20001d68: 0x00017338  
004067 0x20001dd0: 0x00017cc8  
004067 0x20001dd4: 0x00015c2b  
004067 0x20001dd8: 0x00015c1d  
004067 0x20001de4: 0x0001161d  
004067 0x20001df4: 0x00011841  
004067 0x20001e04: 0x000117c9  
004067 0x20001e0c: 0x00013d8f  
004067 0x20001e20: 0x00014375  
004067 0x20001e4c: 0x00013dfd  
004067 0x20001e54: 0x00013e09  
004067 0x20001e58: 0x00013e01
```

(continues on next page)

(continued from previous page)

```
004067 0x20001e5c: 0x0000925d
004067 0x20001e64: 0x00008933
004067 0x20001e7c: 0x00008cc3
004067 0x20001e98: 0x000169e0
004067 0x20001e9c: 0x00008cb9
004067 0x20001ea0: 0x000088a9
```

You can see that it contains the usual dump of registers in the beginning, followed by addresses to stack, and the value at that location. Note that we skip the area of memory within stack which contains the stashed register values. This means that \$pc and \$lr values will not show up in the array of addresses. Depending on your CPU architecture, different stuff gets pushed to stack. Given where Mynewt is ported to, there's always function return address pushed there.

You would then take these values, and see if they seem legit. Here we'll show what to do with the output above.

```
[marko@IsMyLaptop:~/src2/incubator-mynewt-blinky]$ arm-elf-linux-gdb bin/targets/slinky_
˓→nrf52/app/apps/slinky/slinky.elf
GNU gdb (GDB) 7.8.1
Copyright (C) 2014 Free Software Foundation, Inc.
...
No symbol table is loaded. Use the "file" command.
Reading symbols from bin/targets/slinky_nrf52/app/apps/slinky/slinky.elf...done.
(gdb) list *0x0001501b
0x1501b is in crash_test_nmgr_write (repos/apache-mynewt-core/test/crash_test/src/crash_
˓→nmgr.c:68).
63     if (rc != 0) {
64         return MGMT_ERR_EINVAL;
65     }
66
67     rc = crash_device(tmp_str);  <--- That is likely part of it
68     if (rc != 0) {
69         return MGMT_ERR_EINVAL;
70     }
71
72     rc = mgmt_cbuf_setoerr(cb, 0);
(gdb) list *0x00017338  <--- not relevant
(gdb) list *0x00017cc8  <--- neither is this
(gdb) list *0x00015c2b
0x15c2b is in cbor_mbuf_writer (repos/apache-mynewt-core/encoding/tinycbor/src/cbor_mbuf_
˓→writer.c:32).
27 {
28     int rc;
29     struct cbor_mbuf_writer *cb = (struct cbor_mbuf_writer *) arg;
30
31     rc = os_mbuf_append(cb->m, data, len);  <-- Does not seem likely. This probably_
˓→ended here due to an earlier work that this task was doing.
32     if (rc) {
33         return CborErrorOutOfMemory;
34     }
35     cb->enc.bytes_written += len;
36     return CborNoError;
(gdb) list *0x00015c1d
0x15c1d is in cbor_mbuf_writer (repos/apache-mynewt-core/encoding/tinycbor/src/cbor_mbuf_
˓→writer.c:27).
```

(continues on next page)

(continued from previous page)

```

22 #include <tinycbor/cbor.h>
23 #include <tinycbor/cbor_mbuf_writer.h>
24
25 int
26 cbor_mbuf_writer(struct cbor_encoder_writer *arg, const char *data, int len)
27 {
28     int rc;      ----- Nope. Not relevant.
29     struct cbor_mbuf_writer *cb = (struct cbor_mbuf_writer *) arg;
30
31     rc = os_mbuf_append(cb->m, data, len);
(gdb) list *0x0001161d
0x1161d is in create_container (repos/apache-mynewt-core/encoding/tinycbor/src/
→cborencoder.c:244).
239     memcpy(where, &v, sizeof(v));
240 }
241
242 static inline CborError append_to_buffer(CborEncoder *encoder, const void *data,
→size_t len)
243 {
244     return encoder->writer->write(encoder->writer, data, len);  <--- Hmmm, unlikely
245 }
246
247 static inline CborError append_byte_to_buffer(CborEncoder *encoder, uint8_t byte)
248 {
(gdb) list *0x00011841
0x11841 is in preparse_value (repos/apache-mynewt-core/encoding/tinycbor/src/cborparser.
→c:182).
177     /* are we at the end? */
178     if (it->offset == parser->end)
179         return CborErrorUnexpectedEOF;
180
181     uint8_t descriptor = parser->d->get8(parser->d, it->offset); <--- Probably not
→relevant.
182     uint8_t type = descriptor & MajorTypeMask;
183     it->type = type;
184     it->flags = 0;
185     it->extra = (descriptor &= SmallValueMask);
186
(gdb) list *0x000117c9
0x117c9 is in cbor_encoder_create_map (repos/apache-mynewt-core/encoding/tinycbor/src/
→cborencoder.c:521).
516 */
517 CborError cbor_encoder_create_map(CborEncoder *encoder, CborEncoder *mapEncoder,
→size_t length)
518 {
519     if (length != CborIndefiniteLength && length > SIZE_MAX / 2)
520         return CborErrorDataTooLarge;
521     return create_container(encoder, mapEncoder, length, MapType << MajorTypeShift);
→    <-- I don't think this is relevant either.
522 }
523
524 /**

```

(continues on next page)

(continued from previous page)

```

525 * Creates a indefinite-length byte string in the CBOR stream provided by
(gdb) list *0x00013d8f
0x13d8f is in nmgr_handle_req (repos/apache-mynewt-core/mgmt/newtmgr/src/newtmgr.c:261).
256     } else {
257         rc = MGMT_ERR_ENOENT;
258     }
259 } else if (hdr.nh_op == NMGR_OP_WRITE) {
260     if (handler->mh_write) {
261         rc = handler->mh_write(&nmgr_task_cbuf.n_b); <-- This is part of it.
262     } else {
263         rc = MGMT_ERR_ENOENT;
264     }
265 } else {

```

I was sending a newtmgr command which crashed the system. As you can see from this example, little less than 50% of the addresses were part of the call chain. So read this output with care.

Coredump

Coredump contains a full dump of system memory, and CPU registers. You can inspect the system state, including stack of the failing task, or any of the global state.

You can enable coredumps by setting syscfg variable OS_COREDUMP to 1. When crash happens, dumper will write the contents to flash_map region COREDUMP_FLASH_AREA. After the crash, you can use imgmgr to download the coredump for offline analysis. To enable download/erase commands, you need to set syscfg variable IMG-MGR_COREDUMP to 1.

Coredump package does not overwrite a previous coredump, if it exists in the flash. To get a new one, you first need to erase the area.

You can either use a dedicated area of flash, or use image slot 1 as the target. If image slot1 and coredump areas are co-located, coredump will overwrite the image, unless image upgrade is in progress (slot1 marked as pending, or slot0 is not confirmed).

Fetching Coredump with newtmgr

newtmgr coredump management commands are handled by imgmgr. You can query whether corefile is present, download one if it exists, and then erase the area afterwards.

```

[marko@IsMyLaptop:~/src2/incubator-mynewt-blinky]$ newtmgr -c ttys005 image corelist
Corefile present
[marko@IsMyLaptop:~/src2/incubator-mynewt-blinky]$ newtmgr -c ttys005 image coredownload_
↪ -e core.elf
0
512
1024
1536
...
65676
Done writing core file to core.elf;_
↪ hash=7e20dcdd136c6796fcb2a51e7384e90800d2ec045f2ee088af32529e929e2130

```

(continues on next page)

(continued from previous page)

```
[marko@IsMyLaptop:~/src2/incubator-mynewt-blinky]$ newtmgr -c ttys005 image coreerase  
Done
```

I specified option ‘-e’ to coredownload command. This converts the internal data representation to ELF file; a format that gdb understands.

Coredump Analysis Offline

Not all GDB configurations support corefiles. Specifically for the architecture we’re showing here, ‘arm-none-eabi-gdb’ does not have it built in. However, ‘arm-elf-linux-gdb’ does.

Here’s how I built it with MacOS:

```
tar xvzf gdb-7.8.1.tar.gz  
cd gdb-7.8.1  
.configure --target=arm-elf-linux --without-lzma --without-guile --without-libunwind-  
ia64 --with-zlib  
make  
gdb/gdb -v
```

And this is how I built it for Linux:

```
sudo apt-get install zlibc zlib1g zlib1g-dev libexpat-dev libncurses5-dev liblzma-dev  
tar xvzf gdb-7.8.1.tar.gz  
cd gdb-7.8.1  
.configure --target=arm-elf-linux --with-lzma --with-expat --without-libunwind-ia64 --  
--with-zlib --without-babeltrace  
make  
gdb/gdb -v
```

Now that I have a suitable version of gdb at hand, I can use it to analyze the corefile.

```
[marko@IsMyLaptop:~/src2/incubator-mynewt-blinky]$ arm-elf-linux-gdb bin/targets/slinky_  
nrf52/app/apps/slinky/slinky.elf core.elf  
GNU gdb (GDB) 7.8.1  
Copyright (C) 2014 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "--host=x86_64-apple-darwin14.5.0 --target=arm-elf-linux".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to "word"....  
No symbol table is loaded. Use the "file" command.  
Reading symbols from bin/targets/slinky_nrf52/app/apps/slinky/slinky.elf...done.  
[New process 1]  
#0 0x00014f28 in crash_device (
```

(continues on next page)

(continued from previous page)

```

how=how@entry=0x20001dc8 <os_main_stack+3904> "div0")
at repos/apache-mynewt-core/test/crash_test/src/crash_test.c:47
47      val3 = val1 / val2;
(gdb) bt
#0 0x00014f28 in crash_device (
    how=how@entry=0x20001dc8 <os_main_stack+3904> "div0")
at repos/apache-mynewt-core/test/crash_test/src/crash_test.c:47
#1 0x0001501a in crash_test_nmgr_write (cb=0x20003464 <nmgr_task_cbuf>)
at repos/apache-mynewt-core/test/crash_test/src/crash_nmgr.c:67
#2 0x00013d8e in nmgr_handle_req (
    nt=nt@entry=0x200034e4 <nmgr_shell_transport>,
    req=0x20002014 <os_msyst_init_1_data+292>)
at repos/apache-mynewt-core/mgmt/newtmgr/src/newtmgr.c:261
#3 0x00013dfc in nmgr_process (nt=0x200034e4 <nmgr_shell_transport>)
at repos/apache-mynewt-core/mgmt/newtmgr/src/newtmgr.c:325
#4 0x00013e08 in nmgr_event_data_in (ev=<optimized out>)
at repos/apache-mynewt-core/mgmt/newtmgr/src/newtmgr.c:332
#5 0x0000925c in os_eventq_run (evq=<optimized out>)
at repos/apache-mynewt-core/kernel/os/src/os_eventq.c:162
#6 0x00008932 in main (argc=<optimized out>, argv=<optimized out>)
at repos/apache-mynewt-core/apps/slinky/src/main.c:289
(gdb) p g_current_task
$1 = (struct os_task *) 0x20001e88 <os_main_task>
```

Setting up core dumps, setting up a mechanism to download them, managing them on the device and building gdb is certainly more work, but you'll have much more data available when you start analyzing problems. I don't think I can emphasize too much how valuable having all this data is, when you're stuck with a difficult-to-solve crash.

Task Stack Use

One of the common problems is stack overflow within a task. Mynewt OS fills the stack of a task with a pattern at the time task gets created. If you're suspecting that stack might be blown, check the start of the stack (and memory right before it).

There are also APIs to check the stack use. Task statistics, accessible with console CLI or newtmgr, attempt to report the stack use by the tasks. They do it by inspecting the presence of this pattern, so it is possible to fool them also.

Here is what that data looks like at console. ‘stksz’ is the amount of stack, and ‘stkuse’ is how much of it we think the task used it.

Tasks:										
	task	pri	tid	runtime	csw	stksz	stkuse	lcheck	ncheck	flg
7109387	idle	255	0	7109375	72810	64	26	0	0	
7109390	main	127	1	15	6645	1024	388	0	0	
7109394	task1	8	2	0	55543	192	116	0	0	
7109396	task2	9	3	0	55543	64	28	0	0	

There are some helper macros for this when using gdb also. You'll need to eyeball the stack manually, but at least it will tell you where to find the stacks.

```
[marko@IsMyLaptop:~/src2/incubator-mynewt-blinky]$ newt debug slinky_nrf52
Debugging bin/targets/slinky_nrf52/app/apps/slinky/slinky.elf
```

(continues on next page)

(continued from previous page)

```
...
Reading symbols from bin/targets/slinky_nrf52/app/apps/slinky/slinky.elf...done.
os_tick_idle (ticks=128)
    at repos/apache-mynewt-core/hw/mcu/nordic/nrf52xxx/src/hal_os_tick.c:164
164      if (ticks > 0) {
(gdb) os_tasks
Undefined command: "os_tasks". Try "help".
(gdb) source repos/apache-mynewt-core/compiler/gdbmacros/os.gdb
(gdb) os_tasks
  prio state      stack  stksz      task name
* 255  0x1 0x20000e88      64 0x200035fc  idle
  127  0x2 0x20001e88    1024 0x20001e88  main
     8  0x2 0x20003ab0     192 0x20000cbc  task1
     9  0x2 0x20003bc0      64 0x20000d0c  task2
```

Stack Check during Context Switch

You can ask OS to check the top of the stack on every context switch. It'll walk over X number of os_stack_t elements, checking if our pattern is still there for the task which is about to be switched out.

You can enable this by setting syscfg variable OS_CTX_SW_STACK_CHECK to 1. Another syscfg variable, OS_CTX_SW_STACK_GUARD, controls how far into the stack we peek.

Memory Corruption

The recommended way of allocating blocks of memory is by using the OS memory pools. As with all dynamically allocated memory, there can issues with using after it gets freed, or it does not get freed at all, code tries to free the same element multiple times, or you end up overwriting the data within the next element.

There are syscfg variables you can set to get OS to do some of sanity checks for you.

Sanity Check on Free

By setting syscfg variable OS_MEMPOOL_CHECK to 1, OS will act when an element is freed. It will walk through the list of already freed entries, and check that the entry being freed does not already appear in the free list. It will also check that the pool entry the code is trying to free is within the memory range belonging to that pool.

Poisoning Pool

If you set syscfg variable OS_MEMPOOL_POISON, the OS fills pool entry memory with a data pattern whenever that entry is being freed. It then checks that this pattern has not been disturbed when that pool entry is allocated again. This would help to detect the scenarios when code is still using a freed pool entry. Of course, the detection will only be triggered if the code tries to modify the data belonging tot that entry.

Pool Guard Areas

Normally OS takes the memory region allocated for pool use and splits it into chunks which are exactly the size of the pool entry (taking into account the alignment restriction). By setting syscfg variable OS_MEMPOOL_GUARD to 1, it creates a ‘guard area’ between pool elements, and writes a specific data pattern to it. This is then checked whenever the pool entry is freed, or allocated. If code writes past the data area belonging to pool entry, the pattern would likely be disturbed, and would cause an assert.

Exhausting Memory

Memory pool usage can be monitored through the mempool API. The data collected there can be seen via the console CLI and/or using newtmgr.

Here is that data as seen from console:

```
7375292 compat> mpool
7375581 Mempools:
7375581                         name blksz  cnt free  min
7375584                         msys_1   292    12    12    8
7375585                         modlog_mapping_pool 12     16    12    12
```

We list the memory pools OS knows about. ‘cnt’ tells how many elements the pool has in total. ‘free’ tells how many elements are currently in the free pool, and ‘min’ tells what’s the low point for the number of elements within the pool.

What you should check is the ‘min’ number. If it gets to 0, it means that the pool has been exhausted at one point. Depending on what the pool is used for, you might want to take action.

There’s some gdb helper macros to inspect the data belonging to the default networking memory pool, msys_1.

```
(gdb) source repos/apache-mynewt-core/compiler/gdbmacros/mbuf.gdb
(gdb) mn_msys1_print
Mbuf addr: 0x20001ef0
Mbuf header: $1 = {om_data = 0x20002380 <os_msys_init_1_data+1168> "\244$",
  om_flags = 0 '\000', om_pkthdr_len = 8 '\b', om_len = 14,
  om_omp = 0x20002ca0 <os_msys_init_1_mbuf_pool>, om_next = {sle_next = 0x0},
  om_databuf = 0x20001f00 <os_msys_init_1_data+16> "\016"}
Packet header: $2 = {omp_len = 14, omp_flags = 0, omp_next = {stqe_next = 0x0}}
....
```

---Type <return> to continue, or q <return> to quit---q

Quit

```
(gdb) help mn_msys1_print
usage: mn_msys1_print
```

Prints all mbufs in the first msys pool. Both allocated and unallocated mbufs are printed.

```
(gdb) mn_msys1_free_print
```

Mbuf addr: 0x20002138

....

Mbuf addr: 0x20002b7c

Crash in Log

System restarts can also be recorded in reboot log. Assuming reboot log has been set up, it'll write a record either when system is asked to reset (managed reset), or when system is coming back after an unexpected reset.

To facilitate the latter, you should include a call from main() to reboot package, reporting what HAL is telling you.

```
int
main(int argc, char **argv)
{
    sysinit();

    /* .... */
    reboot_start(hal_reset_cause());
}
```

This will then show if the system was restarted due to POR, brownout, or hardware watchdog (the cause determination is MCU specific, so YMMV). As you remember, exit via assert() and unhandled interrupt is going via the dumper. So these kind of reboots will show up as ‘SOFT’ reset.

Restarts due to hardware watchdog will show up here, however. You can also add a printout to beginning of main(), outputting what hal_reset_cause_str() returns. That will generate a report on the console.

4.12 Other

4.12.1 How to Reduce Application Code Size

Getting your application to fit in an image slot can be challenging, particularly on flash constrained hardware such as the nRF51. Below are some suggested system configuration settings that reduce the code size of your Mynewt image.

Setting	Description
LOG_LEVEL: 255	Disable all logging.
LOG_CLI: 0	Disable log shell commands.
STATS_CLI: 0	Disable stats shell commands.
SHELL_TASK: 0	Disable the interactive shell.
SHELL_OS_MODULE: 0	Disable memory management shell commands.
SHELL_CMD_HELP: 0	Disable help for shell commands.

You can use the `newt target set` command to set the syscfg settings in the `syscfg.yml` file for the target. See the [Newt Tool Command Guide](#) for the command syntax.

Note: The `newt target set` command deletes all the current syscfg settings in the target `syscfg.yml` file and only sets the syscfg settings specified in the command. If you are experimenting with different settings to see how they affect the code size and do not want to reenter all the setting values in the `newt target set` command, you can use the `newt target amend` command. This command adds or updates only the settings specified in the command and does not overwrite other setting values. While you can also edit the target `syscfg.yml` file directly, we recommend that you use the `newt target` commands.

4.12.2 Write a Test Suite for a Package

This document guides the reader through creating a test suite for a Mynewt package.

Introduction

Writing a test suite involves using the `test/testutil` package. The testutil library provides the functionality needed to define test suites and test cases.

Choose Your Package Under Test

Choose the package you want to write a test suite for. In this tutorial, we will use the `time/datetime` in the apache-mynewt-core repo. Throughout this tutorial, we will be inside the apache-mynewt-core repo directory, unlike most tutorials which operate from the top-level project directory.

Create A Test Package

Typically, a library has only one test package. The convention is name the test package by appending `/test` to the host library name. For example, the test package for `encoding/json` is `encoding/json/test`. The directory structure of the json package is shown below:

```
encoding/json
├── include
│   └── json
│       └── json.h
├── pkg.yml
└── src
    ├── json_decode.c
    └── json_encode.c
└── test
    ├── pkg.yml
    └── src
        ├── test_json.c
        ├── test_json.h
        ├── test_json_utils.c
        └── testcases
            ├── json_simple_decode.c
            └── json_simple_encode.c
```

The top-level `test` directory contains the json test package. To create a test package for the `datetime` package, we need to create a similar package called `time/datetime/test`.

```
$ newt pkg new time/datetime/test -t unittest
Download package template for package type pkg.
Package successfully installed into /home/me/mynewt-core/time/datetime/test.
```

We now have a test package inside `time/datetime`:

```
time/datetime
├── include
│   └── datetime
│       └── datetime.h
```

(continues on next page)

(continued from previous page)

```

├── pkg.yml
└── src
    └── datetime.c
test
    ├── README.md
    ├── pkg.yml
    └── src
        └── main.c
        syscfg.yml

```

There is one modification we need to make to the new package before we can start writing unit test code. A test package needs access to the code it will be testing, so we need to add a dependency on `@apache-mynewt-core/time/datetime` to our `pkg.yml` file:

```

pkg.name: "time/datetime/test"
pkg.type: unittest
pkg.description: "Description of your package"
pkg.author: "You <you@you.org>"
pkg.homepage: "http://your-url.org/"
pkg.keywords:

pkg.deps:
    - '@apache-mynewt-core/test/testutil'
    - '@apache-mynewt-core/time/datetime'

pkg.deps.SEFTTEST:
    - '@apache-mynewt-core/sys/console/stub'

```

While we have the `pkg.yml` file open, let's take a look at what newt filled in automatically:

- `pkg.type: unittest` designates this as a test package. A *test package* is special in that it can be built and executed using the `newt test` command.
- A test package always depends on `@apache-mynewt-core/test/testutil`. The testutil library provides the tools necessary for verifying package behavior,
- The `SELFTEST` suffix indicates that a setting should only be applied when the `newt test` command is used.

Regarding the conditional dependency on `sys/console/stub`, the datetime package requires some form of console to function. In a regular application, the console dependency would be supplied by a higher order package. Because `newt test` runs the test package without an application present, the test package needs to supply all unresolved dependencies itself when run in self-test mode.

Create Your Test Suite Code

We will be adding a *test suite* to the `main.c` file. The test suite will be empty for now. We also need to invoke the test suite from `main()`.

Our `main.c` file now looks like this:

```

#include "sysinit/sysinit.h"
#include "testutil/testutil.h"

TEST_SUITE(test_datetime_suite) {

```

(continues on next page)

(continued from previous page)

```

    /* Empty for now; add test cases later. */
}

#ifndef MYNEWT_VAL(SELFTEST)
int
main(int argc, char **argv)
{
    /* Initialize all packages. */
    sysinit();

    test_datetime_suite();

    /* Indicate whether all test cases passed. */
    return tu_any_failed;
}
#endif

```

Try It Out

We now have a working test suite with no tests. Let's make sure we get a passing result when we run `newt test`:

```

$ newt test time/datetime
<build output>
Executing test: /home/me/mynewt-core/bin/targets/unittest/time_datetime_test/app/time/
˓→datetime/test/time_datetime_test.elf
Passed tests: [time/datetime/test]
All tests passed

```

Create a Test

To create a test within your test suite, there are two things to do.

1. Implement the test case function using the `testutil` macros.
2. Call the test case function from within the test suite.

For this tutorial we will create a test case to verify the `datetime_parse()` function. The `datetime_parse()` function is declared as follows:

```

/**
 * Parses an RFC 3339 datetime string. Some examples of valid datetime strings
 * are:
 * 2016-03-02T22:44:00          UTC time (implicit)
 * 2016-03-02T22:44:00Z         UTC time (explicit)
 * 2016-03-02T22:44:00-08:00    PST timezone
 * 2016-03-02T22:44:00.1        fractional seconds
 * 2016-03-02T22:44:00.101+05:30 fractional seconds with timezone
 *
 * On success, the two output parameters are filled in (tv and tz).
 *
 * @return                      0 on success;

```

(continues on next page)

(continued from previous page)

```

/*
 * nonzero on parse error.
 */
int
datetime_parse(const char *input, struct os_timeval *tv, struct os_timezone *tz)

```

Our test case should make sure this function rejects invalid input, and that it parses valid input correctly. The updated `main.c` file looks like this:

```
#include "sysinit/sysinit.h"
#include "testutil/testutil.h"
#include "os/os_time.h"
#include "datetime/datetime.h"

TEST_SUITE(test_datetime_suite)
{
    test_datetime_parse_simple();
}

TEST_CASE(test_datetime_parse_simple)
{
    struct os_timezone tz;
    struct os_timeval tv;
    int rc;

    /*** Valid input. **/

    /* No timezone; UTC implied. */
    rc = datetime_parse("2017-06-28T22:37:59", &tv, &tz);
    TEST_ASSERT_FATAL(rc == 0);
    TEST_ASSERT(tv.tv_sec == 1498689479);
    TEST_ASSERT(tv.tv_usec == 0);
    TEST_ASSERT(tz.tz_minuteswest == 0);
    TEST_ASSERT(tz.tz_dsttime == 0);

    /* PDT timezone. */
    rc = datetime_parse("2013-12-05T02:43:07-07:00", &tv, &tz);
    TEST_ASSERT_FATAL(rc == 0);
    TEST_ASSERT(tv.tv_sec == 1386236587);
    TEST_ASSERT(tv.tv_usec == 0);
    TEST_ASSERT(tz.tz_minuteswest == 420);
    TEST_ASSERT(tz.tz_dsttime == 0);

    /*** Invalid input. **/

    /* Nonsense. */
    rc = datetime_parse("abc", &tv, &tz);
    TEST_ASSERT(rc != 0);

    /* Date-only. */
    rc = datetime_parse("2017-01-02", &tv, &tz);
    TEST_ASSERT(rc != 0);
}
```

(continues on next page)

(continued from previous page)

```

/* Zero month. */
rc = datetime_parse("2017-00-28T22:37:59", &tv, &tz);
TEST_ASSERT(rc != 0);

/* 13 month. */
rc = datetime_parse("2017-13-28T22:37:59", &tv, &tz);
TEST_ASSERT(rc != 0);
}

#if MYNEWT_VAL(SELFTEST)
int
main(int argc, char **argv)
{
    /* Initialize all packages. */
    sysinit();

    test_datetime_suite();

    /* Indicate whether all test cases passed. */
    return tu_any_failed;
}
#endif

```

Take a few minutes to review the above code. Then keep reading for some specifics.

Asserting

The `test/testutil` package provides two tools for verifying the correctness of a package:

- `TEST_ASSERT`
- `TEST_ASSERT_FATAL`

Both of these macros check if the supplied condition is true. They differ in how they behave when the condition is not true. On failure, `TEST_ASSERT` reports the error and proceeds with the remainder of the test case. `TEST_ASSERT_FATAL`, on the other hand, aborts the test case on failure.

The general rule is to only use `TEST_ASSERT_FATAL` when subsequent assertions depend on the condition being checked. For example, when `datetime_parse()` is expected to succeed, the return code is checked with `TEST_ASSERT_FATAL`. If `datetime_parse()` unexpectedly failed, the contents of the `tv` and `tz` objects would be indeterminate, so it is desirable to abort the test instead of checking them and reporting spurious failures.

Scaling Up

The above example is small and self contained, so it is reasonable to put everything in a single C file. A typical package will need a lot more test code, and it helps to follow some conventions to maintain organization. Let's take a look at a more realistic example. Here is the directory structure of the `fs/nffs/test` package:

```

fs/nffs/test
└── pkg.yml
└── src
    └── nffs_test.c

```

(continues on next page)

(continued from previous page)

```
└── nffs_test.h
└── nffs_test_debug.c
└── nffs_test_priv.h
└── nffs_test_system_01.c
└── nffs_test_utils.c
└── nffs_test_utils.h
└── testcases
    ├── append_test.c
    ├── cache_large_file_test.c
    ├── corrupt_block_test.c
    ├── corrupt_scratch_test.c
    ├── gc_on_oom_test.c
    ├── gc_test.c
    ├── incomplete_block_test.c
    ├── large_system_test.c
    ├── large_unlink_test.c
    ├── large_write_test.c
    ├── long_filename_test.c
    ├── lost_found_test.c
    ├── many_children_test.c
    ├── mkdir_test.c
    ├── open_test.c
    ├── overwrite_many_test.c
    ├── overwrite_one_test.c
    ├── overwrite_three_test.c
    ├── overwrite_two_test.c
    ├── read_test.c
    ├── readdir_test.c
    ├── rename_test.c
    ├── split_file_test.c
    ├── truncate_test.c
    ├── unlink_test.c
    └── wear_level_test.c
```

The `fs/nffs/test` package follows these conventions:

1. A maximum of one test case per C file.
2. Each test case file goes in the `testcases` subdirectory.
3. Test suites and utility functions go directly in the `src` directory.

Test packages contributed to the Mynewt project should follow these conventions.

Congratulations

Now you can begin the work of validating your packages.

4.12.3 Enable Wi-Fi on Arduino MKR1000

This tutorial shows you how to enable Wi-Fi on an Arduino MKR1000 board and connect to a Wi-Fi network.

Prerequisites

Ensure that you have met the following prerequisites before continuing with this tutorial:

- Have an Arduino MKR1000 board.
- Have Internet connectivity to fetch remote Mynewt components.
- Have a computer to build a Mynewt application and connect to the board over USB.
- Have a Micro-USB cable to connect the board and the computer.
- Have local Wi-Fi network that the computer is connected to and that the MKR1000 board can join.
- Have a *Serial Port Setup*.
- Have a Segger J-Link Debug Probe.
- Have a J-Link 9 pin Cortex-M Adapter that allows JTAG, SWD and SWO connections between J-Link and Cortex M based target hardware systems
- Install the [Segger JLINK Software](#) and documentation pack.
- Install the Newt tool and toolchains (See [Native Installation](#))
- Create a project space (directory structure) and populated it with the core code repository ([apache-mynewt-core](#)) or know how to as explained in [Creating Your First Project](#).
- Read the Mynewt OS [Concepts](#) section.

Create a Project

Create a new project if you do not have an existing one. You can skip this step and proceed to [Fetch External Packages](#) if you already created a project.

Run the following commands to create a new project:

```
$ mkdir ~/dev
$ cd ~/dev
$ newt new arduinowifi
Downloading project skeleton from apache/mynewt-blinky...
Downloading repository mynewt-blinky (commit: master) ...
Installing skeleton in arduinowifi...
Project arduinowifi successfully created.
$ cd arduinowifi
$ newt upgrade
Downloading repository mynewt-core (commit: master) ...
Downloading repository mcuboot (commit: master) ...
Downloading repository mynewt-nimble (commit: master) ...
Making the following changes to the project:
```

(continues on next page)

(continued from previous page)

```
apache-mynewt-core successfully upgraded to version 1.7.0
apache-mynewt-nimble successfully upgraded to version 1.2.0
mcuboot successfully upgraded to version 1.3.1
$
```

Fetch External Packages

Mynewt uses source code provided directly from the chip manufacturer for low level operations. Sometimes this code is licensed only for the specific manufacturer of the chipset and cannot live in the Apache Mynewt repository. That happens to be the case for the Arduino Zero board which uses Atmel SAMD21. Runtime's git hub repository hosts such external third-party packages and the Newt tool can fetch them.

To fetch the package with MCU support for Atmel SAMD21 for Arduino Zero from the Runtime git repository, you need to add the repository to the `project.yml` file in your base project directory.

Mynewt uses source code provided directly from the chip manufacturer for low level operations. Sometimes this code is licensed only for the specific manufacturer of the chipset and cannot live in the Apache Mynewt repository. That happens to be the case for the Arduino Zero board which uses Atmel SAMD21. Runtime's github repository hosts such external third-party packages and the Newt tool can fetch them.

To fetch the package with MCU support for Atmel SAMD21 for Arduino Zero from the Runtime git repository, you need to add the repository to the `project.yml` file in your base project directory (`arduinowifi`).

Here is an example `project.yml` file with the Arduino Zero repository added. The sections with `mynewt_arduino_zero` that need to be added to your project file are highlighted.

Note: On Windows platforms: You need to set `vers` to `0-dev` and use the latest master branch for both repositories.

```
$ more project.yml

project.name: "my_project"

project.repositories:
    - apache-mynewt-core
    - mynewt_arduino_zero

repository.apache-mynewt-core:
    type: github
    vers: 1-latest
    user: apache
    repo: mynewt-core

repository.mynewt_arduino_zero:
    type: github
    vers: 1-latest
    user: runtimeco
    repo: mynewt_arduino_zero
$
```

Install the project dependencies using the `newt upgrade` command (you can specify `-v` for verbose output):

```
$ newt upgrade
Downloading repository mynewt_arduino_zero (commit: master) ...
Skipping "apache-mynewt-core": already upgraded (1.7.0)
```

(continues on next page)

(continued from previous page)

```
Skipping "apache-mynewt-nimble": already upgraded (1.2.0)
Skipping "mcuboot": already upgraded (1.3.1)
Making the following changes to the project:
mynewt_arduino_zero successfully upgraded to version 1.7.0
$
```

NOTE: If there has been a new release of a repo used in your project since you last installed it, the 1-latest version for the repo in the `project.yml` file will refer to the new release and will not match the installed files. In that case you will get an error message saying so and you will need to run `newt upgrade` to overwrite the existing files with the latest codebase.

Create a Target for the Bootloader

You need to create two targets for the MKR1000 board, one for the bootloader and one for the `winc1500_wifi` application. Run the following `newt target` commands, from your project directory, to create a bootloader target. We name the target `mkr1000_boot`.

```
$ newt target create mkr1000_boot
$ newt target set mkr1000_boot bsp=@mynewt_arduino_zero/hw/bsp/arduino_mkr1000
$ newt target set mkr1000_boot app=@mcuboot/boot/mynewt
$ newt target set mkr1000_boot build_profile=optimized
$ newt target set mkr1000_boot syscfg=BSP_ARDUINO_ZERO_PRO=1
```

Create a Target for the Wi-Fi Application

Run the following `newt target` commands to create a target for the `winc1500_wifi` application in the arduino repository. We name the application target `mkr1000_wifi`.

```
$ newt target create mkr1000_wifi
$ newt target set mkr1000_wifi app=@mynewt_arduino_zero/apps/winc1500_wifi
$ newt target set mkr1000_wifi bsp=@mynewt_arduino_zero/hw/bsp/arduino_mkr1000
$ newt target set mkr1000_wifi build_profile=debug
$ newt target set mkr1000_boot syscfg=BSP_ARDUINO_ZERO_PRO=1
```

Build the Bootloader

Run the `newt build mkr1000_boot` command to build the bootloader:

```
$ newt build mkr1000_boot
Building target targets/mkr1000_boot
Compiling repos/mcuboot/boot/bootutil/src/image_rsa.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec256.c
Compiling repos/apache-mynewt-core/crypto/mbedtls/src/aes.c
Compiling repos/mcuboot/boot/bootutil/src/image_ec.c
Compiling repos/mcuboot/boot/bootutil/src/image_validate.c
Compiling repos/mcuboot/boot/mynewt/src/main.c
```

...

(continues on next page)

(continued from previous page)

```
Archiving util_mem.a
Linking ~/dev/arduinowifi/bin/targets/mkr1000_boot/app/@mcuboot/boot/mynewt/mynewt.elf
Target successfully built: targets/mkr1000_boot
$
```

Build the Wi-Fi Application

Run the `newt build mkr1000_wifi` command to build the wi-fi application image:

```
$newt build mkr1000_wifi
Building target targets/mkr1000_wifi
Compiling repos/apache-mynewt-core/boot/bootutil/src/image_ec.c
Compiling repos/apache-mynewt-core/boot/bootutil/src/image_ec256.c
Compiling repos/apache-mynewt-core/boot/bootutil/src/image_rsa.c
Compiling repos/apache-mynewt-core/boot/bootutil/src/image_validate.c
Compiling repos/apache-mynewt-core/boot/bootutil/src/loader.c
...
Archiving util_mem.a
Linking ~/dev/arduinowifi/bin/targets/mkr1000_wifi/app/apps/winc1500_wifi/winc1500_wifi.elf
Target successfully built: targets/mkr1000_wifi
$
```

Sign and Create the Wi-Fi Application Image

Run the `newt create-image mkr1000_wifi 1.0.0` command to sign and create an image file for the Wi-Fi application. You may assign an arbitrary version (e.g. 1.0.0) number.

```
$newt create-image mkr1000_wifi 1.0.0
Compiling bin/targets/mkr1000_wifi/generated/src/mkr1000_wifi-sysinit-app.c
Archiving mkr1000_wifi-sysinit-app.a
Linking ~/dev/arduinowifi/bin/targets/mkr1000_wifi/app/apps/winc1500_wifi/winc1500_wifi.elf
App image successfully generated: ~/dev/arduinowifi/bin/targets/mkr1000_wifi/app/apps/winc1500_wifi/winc1500_wifi.img
$
```

Connect to the Board

- Connect your computer to the MKR1000 board with the Micro-USB cable.
- Connect the debug probe to the JTAG port on the board using the Jlink 9-pin adapter and cable.



Mynewt will download and debug the target through this port. You should see a green LED come on and indicates the board has power.

Load the Bootloader onto the Board

Run the `newt load mkr1000_boot` command to load the bootloader onto the board:

```
$ newt load mkr1000_boot
Loading bootloader
$
```

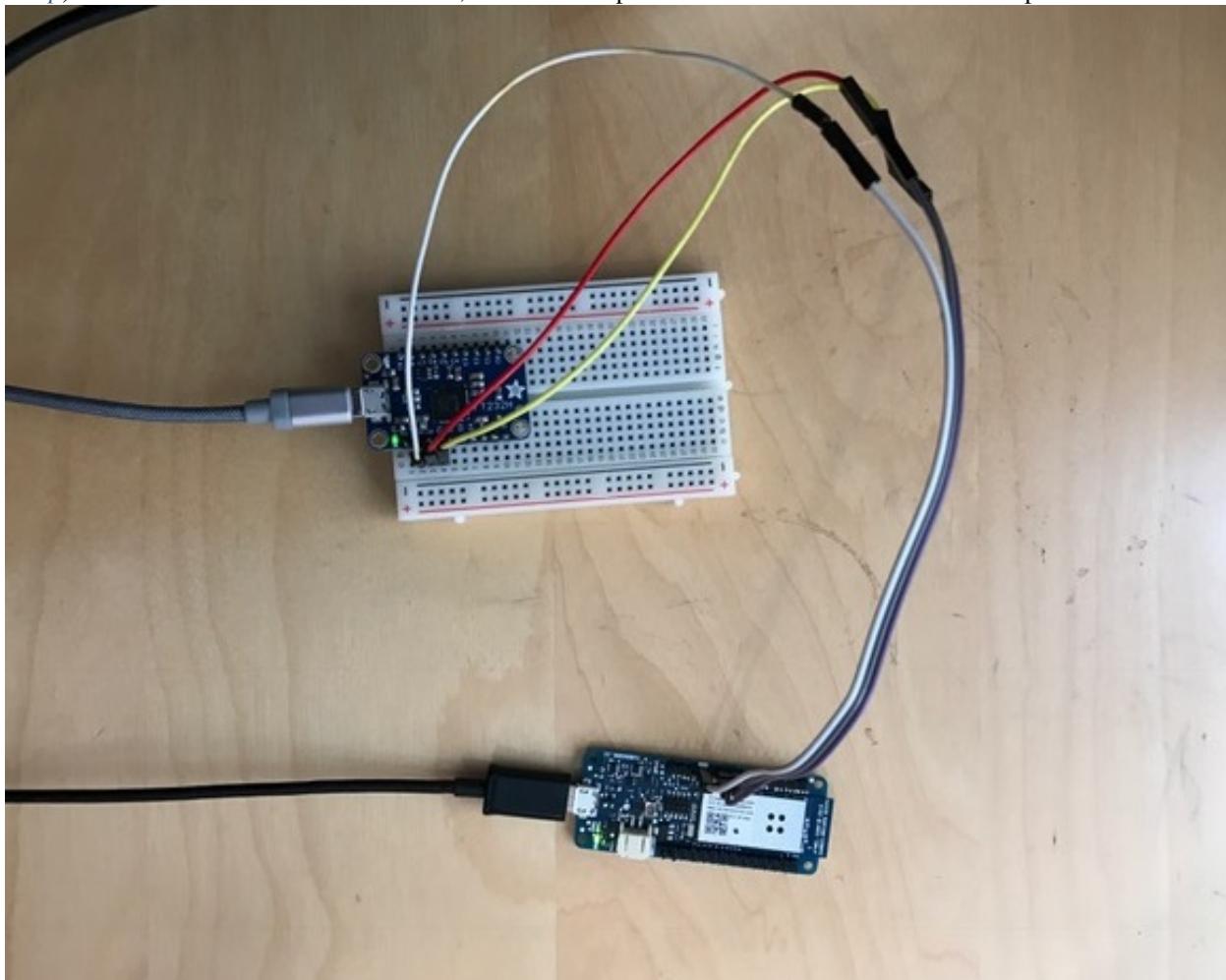
Load the Wi-Fi Application Image onto the Board

Run the `newt load mkr1000_wifi` command to load the wifi application onto the board:

```
$ newt load mkr1000_wifi
Loading app image into slot 1
$
```

Setup a Serial Connection Between Your Computer and the Board

Set up a serial connection from your computer to the MKR1000 board (See [Serial Port Setup](#)). On the MKR1000 board, the TX pin is PIN 14 and the RX pin is PIN 13.



Locate the port, in the /dev directory on your computer, that the serial connection uses. The format of the port name is platform dependent:

- Mac OS uses the format `tty.usbserial-<some identifier>`.
- Linux uses the format `TTYUSB<N>`, where N is a number. For example, `TTYUSB2`.
- MinGW on Windows uses the format `ttyS<N>`, where N is a number. You must map the port name to a Windows COM port: `/dev/ttyS<N>` maps to `COM<N+1>`. For example, `/dev/ttyS2` maps to `COM3`.

You can also use the Windows Device Manager to find the COM port number.

```
$ ls /dev/tty*usbserial*
/dev/tty.usbserial-1d13
$
```

Start Wi-Fi via console

Use a terminal emulation program to communicate with the board over the serial port. This tutorial shows a Minicom set up. Run the minicom command with the serial port you located on your computer:

Note: On Windows, you can use the PuTTY application.

```
$ minicom -D /dev/tty.usbserial-1d13 -b 115200
```

Type `wifi start` to start Wi-Fi.

```
Welcome to minicom 2.7.1

OPTIONS:
Compiled on May 17 2017, 15:29:14.
Port /dev/tty.usbserial, 15:12:10

Press Meta-Z for help on special keys

138465 compat> wifi start
144570 compat> (APP)(INFO)Chip ID 1503a0
(APP)(INFO)Firmware ver    : 19.4.4
(APP)(INFO)Min driver ver : 19.3.0
(APP)(INFO)Curr driver ver: 19.3.0
wifi_init : 0
```

Connect to the local Wi-Fi network. Note that the MKR1000 board only supports 2.4 GHz Wi-Fi networks.

Run the `wifi connect` command and specify your network and . After you are connected to your wi-fi network, run the `net service` command to start network services.

```
wifi connect
037624 wifi_request_scan : 0
037627 compat> scan_results 7: 0
038454 wifi_connect : 0
039451 connect_done : 0
039958 dhcp done
192.168.0.135
```

(continues on next page)

(continued from previous page)

```
040169 get sys time response 2017.7.12-22.41.33  
net service
```

The board is connected to the network successfully and has IP address: 192.168.0.135

Establish TCP Connection and Talk!

From a terminal on your computer, telnet to ports 7, 9, or 19 using the IP address your board has been assigned. Type something on this terminal and see the console output (on minicom). Can you see the difference in the behaviors?

```
$telnet 192.168.0.135 7  
Trying 192.168.0.135...  
Connected to 192.168.0.135.  
Escape character is '^]'.  
hello  
hello  
^]  
telnet> q  
$
```

One port echoes whatever is typed, one discards everything it gets, and the third spews out bits constantly. Type `wifi stop` to disable WiFi on the Arduino board.

4.12.4 Charge control on PineTime

This tutorial shows you how to get the charge control status on PineTime smartwatch.

- *Prerequisites*
- *Charger hardware*
- *SGM4056 Driver*
- *Charge control*
- *Charger interrupt*
- *Conclusion*

Prerequisites

Ensure that you meet the following prerequisites before continuing with this tutorial:

- Follow [Blinky on PineTime tutorial](#) to create a project with a basic application. You will extend that application in this tutorial.
- Make sure you have the charger input available. This can either be a fully assembled PineTime (but this prevent you from accessing the SWD pins) or by mounting a wire to the 5V charger pad.

Charger hardware

First a few words about the PineTime hardware. The PineTime smartwatch uses a SGM4056 charger chip. The chip gets its power from the USB port via the charging pads at the back of the watch. The charger takes care of battery maintenance by providing the correct voltage and current during the charging process.

The charger is connected to the main processor via two GPIO pins. This way the charger can report its current charging state:

- no source connected,
- charging or
- source connected but not charging.

This tutorial will show you how to obtain this status.

SGM4056 Driver

Communication with the charger is done by the *SGM4056 Driver*. This abstracts the hardware and provides a simple interface to the charger. The *PineTime BSP* already initializes the driver, so we can use it directly in our application. Let's extend the application with the following code:

```
#include "sgm4056/sgm4056.h"
#include "console/console.h"

...

int
main(int argc, char **argv)
{
    int rc;
    struct sgm4056_dev *charger;
    charge_control_status_t charger_status;

    ...

    sysinit();

    g_led_pin = LED_BLINK_PIN;
    hal_gpio_init_out(g_led_pin, 1);

    /* Open charger device */
    charger = (struct sgm4056_dev *) os_dev_open("charger", 0, 0);
    assert(charger);

    while (1) {
        ++g_task1_loops;

        /* Wait one second */
        os_time_delay(OS_TICKS_PER_SEC);

        /* Toggle the LED */
        hal_gpio_toggle(g_led_pin);
```

(continues on next page)

(continued from previous page)

```

/* Get charger state */
rc = sgm4056_get_charger_status(charger, &charger_status);
assert(rc == 0);

/* Print charger state */
console_printf("Charger state = %i\n", charger_status);
console_flush();
}

assert(0);

return rc;
}

```

First we added a include file for the `sgm4056` driver and the console interface for output.

We define a pointer to a `sgm4056_dev` charger device and a variable for the actual charger status.

Then we open the charger device using `os_dev_open`. This will get the driver instance that was initialized by the BSP.

In the while loop we ask the driver to get the charger state and print it to the console as an number.

Let's run this code on the device and watch the output of the console:

```
$ newt run blinky-pinetime 0
```

```

Charger state = 2
Charger state = 2
Charger state = 2
...

```

Warning

Currently the PineTime BSP doesn't support the serial console properly. Therefore you need to setup ARM semi-hosting manually in the application to make these instructions work. These step are beyond the scope of this tutorial.

If you connect or disconnect the charger input, you will see the number changes. However it is not yet clear what the number actually means. Let's make that output more useful:

```

...
char * get_charger_status_string(charge_control_status_t status) {
    static char * no_source_string = "no source detected";
    static char * charging_string = "charging";
    static char * complete_string = "charge completed";
    switch (status)
    {
        case CHARGE_CONTROL_STATUS_NO_SOURCE:
            return no_source_string;

        case CHARGE_CONTROL_STATUS_CHARGING:
            return charging_string;
    }
}
```

(continues on next page)

(continued from previous page)

```

case CHARGE_CONTROL_STATUS_CHARGE_COMPLETE:
    return complete_string;

default:
    return NULL;
}
}

main(int argc, char **argv)
{
...
/* Print charger state */
console_printf("Charger state = %s\n", get_charger_status_string(charger_status));
console_flush();
...
}

```

This adds a function for converting the status enum to a text and then it uses that to output a text representation of the state.

Let's run this improved code and connect the charger:

```
$ newt run blinky-pinetime 0
```

```

Charger state = no source detected
Charger state = no source detected
Charger state = no source detected
Charger state = charging
Charger state = charging
Charger state = charging

```

Great, that is more like it. This code can be used to make a great smartwatch application. However I think we can do better.

Charge control

The code of the last section works great, however it is very specific to the SGM4056 driver. Luckily we can fix that using the *Charge Control interface*. This is enabled by default for the SGM4056 driver, so we don't need to do any configuration for using this new interface.

Charge control works with callbacks for reporting the status. Let's start with adding our callback to the application.

```

#include "charge-control/charge_control.h"

...
static int
charger_data_callback(struct charge_control *chg_ctrl, void *arg,

```

(continues on next page)

(continued from previous page)

```

void *data, charge_control_type_t type)
{
    if (type == CHARGE_CONTROL_TYPE_STATUS) {
        charge_control_status_t charger_status = *(charge_control_status_t*)(data);

        console_printf("Charger state = %s\n", get_charger_status_string(charger_
status));
        console_flush();
    }
    return 0;
}
...

```

First we include the `charge_control.h` header for the correct types. Then we define the callback, which is of the type `charge_control_data_func_t`.

The first argument is the a pointer to `charge_control`. This is a representation of the charger. The second argument is a arg pointer. We will find out later where these two come from.

The third and fourth argument are a pointer to the actual data and a indication of the type of data. This callback can only handle status data of type `CHARGE_CONTROL_TYPE_STATUS`. After checking that, we convert the data to the correct type and print it like in the previous section.

Now we need to change the `main` function to actually call the callback:

```

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

    ...

    /* Open charger device */
    charger = charge_control_mgr_find_next_bytype(CHARGE_CONTROL_TYPE_STATUS, NULL);
    assert(charger);

    while (1) {
        ...

        /* Get charger state */
        rc = charge_control_read(charger, CHARGE_CONTROL_TYPE_STATUS,
            charger_data_callback, NULL, OS_TIMEOUT_NEVER);
        assert(rc == 0);
    }
    assert(0);

    return rc;
}

```

There are a few important changes:

- The type of `charger` has changed. We now use the generic type `charge_control`, which will work for all charger drivers. Notice that this is the same type as the first argument as the callback.

- We don't open a OS device anymore, instead we call the Charge Control Manager and ask for the first charger that supports CHARGE_CONTROL_TYPE_STATUS. Notice that this is the same type as in the callback function.
- Then we execute a read on the charger, for data of type CHARGE_CONTROL_TYPE_STATUS. When finished we want to it to call our callback with the argument NULL and we disable the timeout.

When you run this code, you will get the same results as the previous run, however this code will work with any charger.

Charger interrupt

One of the advantages of charge control is that it supports interrupt-driven notifications. This reduces polling and therefore reduces power usage. This requires a few small changes to our application:

```
...
struct charge_control_listener charger_listener = {
    .ccl_type = CHARGE_CONTROL_TYPE_STATUS,
    .ccl_func = charger_data_callback,
};

...

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

    ...

    /* Open charger device */
    charger = charge_control_mgr_find_next_bytype(CHARGE_CONTROL_TYPE_STATUS, NULL);
    assert(charger);

    /* Set polling rate */
    rc = charge_control_set_poll_rate_ms("charger", 10000);
    assert(rc == 0);

    /* Register charger callback */
    rc = charge_control_register_listener(charger, &charger_listener);
    assert(rc == 0);

    while (1) {
        /* No charger code needed here */
        os_eventq_run(os_eventq_dflt_get());
    }
    assert(0);

    return rc;
}
```

First we need to define a `charge_control_listener` structure, this points to the callback function and indicates the type of data we are interested in. It could also define the argument, but in this example we are not interested in that.

Then we need to set a polling rate, which we can set high as most changes are reported by interrupt. Lastly we register the listener to actually receive the callbacks.

There is no charger code needed in the while loop. However we need the event queue to be handled as charge control will use events to do the polling and interrupt handling. Note that you need to remove the `os_time_delay` to make the events work properly.

Run this code and you see that the charger state is only show every ten seconds. But when you connect the charger you see the output directly. This shows the combination of polling and interrupt-based data acquisition.

Conclusion

You now have an efficient charger status reading application. It will work with any charger driver, not just the one in the PineTime. It uses interrupts to be notified of changes quickly. The next step is to integrate this into your own project.

4.12.5 Rust Blinky application

This tutorial shows you how to convert the Blinky application to Rust. This includes integrating Cargo into newt builder.

- *Prerequisites*
- *Initialize Rust*
- *Setup the basics*
- *Converting sysinit*
- *Doing GPIO and delays*
- *Cargo build*
- *Newt integration*
- *Conclusion*

Prerequisites

Ensure that you meet the following prerequisites before continuing with this tutorial:

- You have basic knowledge about [Rust](#).
- You have basic knowledge about [Rust Embedded](#).
- You have rust installed using [rustup](#).
- Follow [Blinky tutorial](#) to create a project with a basic application. You will extend that application in this tutorial.

Initialize Rust

The first step is to initialize a crate for developing your application in. You can do this in the existing `blinky` directory.

```
$ cargo init --lib apps/blinky
Created library package
$ tree apps/blinky
apps/blinky
├── Cargo.toml
└── pkg.yml
```

(continues on next page)

(continued from previous page)

```

└── src
    ├── lib.rs
    └── main.c

```

1 directory, 4 files

This creates a Cargo.toml configuration file and src/lib.rs. We will use these files to place our application in. You may notice that the Rust application is not configured as an app, but as an lib. This is needed later to allow linking to the rest of mynewt.

Setup the basics

Now we want to actually convert the application code to Rust. Let's open *src/lib.rs* and remove the contents. Then start with some basic setup:

```

#![no_std]

extern crate panic_halt;

#[no_mangle]
pub extern "C" fn main() {

    loop {
    }
}

```

The first line states that this program doesn't use the standard library. This means that only the core library is linked to the program. See [rust-embedded book](#) for more information.

The next line specifies the panic handler. Panicking is an important feature of Rust. To ease this tutorial we choose to halt the processor on panic. See [rust-embedded book](#) for alternatives.

Then we have the `main` function. It contains a endless loop as it should never return, but doesn't do anything useful yet. It is marked `no_mangle` and `extern "C"` to make sure it can be called from the mynewt C code.

Converting sysinit

The next step is to do the sysinit. This is implemented as a C macro, which is incompatible with Rust. This could be solved by using a C library, but for this tutorial we will simply execute the macro manually.

```

#![no_std]

extern "C" {
    fn sysinit_start();
    fn sysinit_app();
    fn sysinit_end();
}

extern crate panic_halt;

#[no_mangle]
pub extern "C" fn main() {

```

(continues on next page)

(continued from previous page)

```
/* Initialize all packages. */
unsafe { sysinit_start(); }
unsafe { sysinit_app(); }
unsafe { sysinit_end(); }

loop {
}

}
```

First we manually define the three sysinit functions. This is similar to the C header file. Then we execute the sysinit as the macro would do.

We need the `unsafe` indication because the C code doesn't have the same memory guarantees as Rust. Normally we need to build a safe Rust wrapper, but that is out of scope for this tutorial.

Doing GPIO and delays

Now it is time to do some GPIO and add a delay. Again we define the functions and then use them in and around the loop. We need some constants that are normally defined by the BSP or MCU. These constants need to move to a better place later.

```
#![no_std]

extern "C" {
    fn sysinit_start();
    fn sysinit_app();
    fn sysinit_end();
    fn hal_gpio_init_out(pin: i32, val: i32) -> i32;
    fn hal_gpio_toggle(pin: i32);
    fn os_time_delay(osticks: u32);
}

extern crate panic_halt;

const OS_TICKS_PER_SEC: u32 = 128;

const LED_BLINK_PIN: i32 = 23;

#[no_mangle]
pub extern "C" fn main() {
    /* Initialize all packages. */
    unsafe { sysinit_start(); }
    unsafe { sysinit_app(); }
    unsafe { sysinit_end(); }

    unsafe { hal_gpio_init_out(LED_BLINK_PIN, 1); }

    loop {
        /* Wait one second */
        unsafe { os_time_delay(OS_TICKS_PER_SEC); }

        /* Toggle the LED */
    }
}
```

(continues on next page)

(continued from previous page)

```

unsafe { hal_gpio_toggle(LED_BLINK_PIN); }
}
}

```

Cargo build

Now that the application is converted we need to build it and link it the rest of mynewt. We start with Cargo.toml:

```

[package]
name = "rust-klok"
version = "0.1.0"
authors = ["Casper Meijn <casper@meijn.net>"]
edition = "2018"

[dependencies]
panic-halt = "0.2.0"

[lib]
crate-type = ["staticlib"]

```

This adds the `panic-halt` dependency, which is needed for the panic handler as mentioned earlier. It also configures the crate as staticlib, which causes the application to be build as .a-library. This will be needed in a later step.

Next we need a script for running cargo and moving the library to the correct place. Create a new file named `apps/blinky/cargo_build.sh` with the following contents:

```

#!/bin/bash

set -eu

if [[ ${MYNEWT_VAL_ARCH_NAME} == '"cortex_m0"' ]]; then
    TARGET="thumbv6m-none-eabi"
elif [[ ${MYNEWT_VAL_ARCH_NAME} == '"cortex_m3"' ]]; then
    TARGET="thumbv7m-none-eabi"
elif [[ ${MYNEWT_VAL_ARCH_NAME} == '"cortex_m4"' || ${MYNEWT_VAL_ARCH_NAME} == '"cortex_m7"' ]]; then
    if [[ ${MYNEWT_VAL_HARDFLOAT} -eq 1 ]]; then
        TARGET="thumbv7em-none-eabihf"
    else
        TARGET="thumbv7em-none-eabi"
    fi
else
    echo "The ARCH_NAME ${MYNEWT_VAL_ARCH_NAME} is not supported"
    exit 1
fi

cargo build --target="${TARGET}" --target-dir="${MYNEWT_PKG_BIN_DIR}"
cp "${MYNEWT_PKG_BIN_DIR}"/${TARGET}/debug/*.a "${MYNEWT_PKG_BIN_ARCHIVE}"

```

The script first sets the name of the target as the Rust compiler knows it. Sadly this is not the same as the names mynewt uses. You need to choose the same type of compiler as mynewt uses. The following targets are available depending on the MCU type:

- `thumbv6m-none-eabi` - use this for Cortex-M0 and Cortex-M0+.
- `thumbv7m-none-eabi` - use this for Cortex-M3.
- `thumbv7em-none-eabi` - use this for Cortex-M4 and Cortex-M7.
- `thumbv7em-none-eabihf` - use this for Cortex-M4 and Cortex-M7 with the HARDFLOAT syscfg enabled.

Then it runs `cargo build` with the target directory set to a path that `newt` provides. Lastly it copies the generated library to the correct path.

Don't forget to mark the script as executable:

```
$ chmod +x apps/blinky/cargo_build.sh
```

Newt integration

To automatically run `cargo` we need to add the following to `pkg.yml`:

```
...  
  
pkg.pre_build_cmds:  
  './cargo_build.sh': 1  
  
pkg.lfags:  
  - '-Wl,--allow-multiple-definition'
```

The first section tells the mynewt build system to run `cargo_build.sh` as part of `newt build`. The second section tells the linker to ignore double function definitions. This is needed as the Rust compiler adds some functions that are also in `baselibc`.

Now we are ready to build a firmware! Remove `main.c` and start the build:

```
$ rm apps/blinky/src/main.c  
$ newt build nrf52_blinky
```

If this command complains about a target may not be installed, then you need to install it. You need the same toolchain as configured earlier for the `TARGET` variable:

```
$ rustup target add <your-target>
```

Conclusion

You now have a firmware where the application is written in Rust. It is nicely integrated into `newt` builder. However it still needs some work: it misses safe Rust wrappers for the mynewt libraries and there are some magic constants that need to be moved to a better location.

THIRD-PARTY RESOURCES

This page lists links to information, tutorial, articles, and posts on Apache Mynewt hosted outside the ASF infrastructure.

- *Helpful Tutorials and Tools*

5.1 Helpful Tutorials and Tools

- [Sensor app and network tutorial](#): A detailed step-by-step guide on how to build your IoT Sensor Network using STM32 Blue Pill, nRF24L01, ESP8266, thethings.io, and Apache Mynewt.
- [Ubuntu VirtualBox with Source Code and Build Tools](#): An Ubuntu VirtualBox with the complete Mynewt 1.6.0 source code + build tools for STM32 Blue Pill to go with the tutorial above.

OS USER GUIDE

This guide provides comprehensive information about Mynewt OS, the real-time operating system for embedded systems. It is intended both for an embedded real-time software developer as well as higher-level applications developer wanting to understand the features and benefits of the system. Mynewt OS is highly composable and flexible. Only those features actually used by the application are compiled into the run-time image. Hence, the actual size of Mynewt is completely determined by the application. The guide covers all the features and services available in the OS and contains several examples showing how they can be used. **Send us an email on the dev@ mailing list if you have comments or suggestions!** If you haven't joined the mailing list, you will find the links [here](#).

6.1 Kernel

6.1.1 Apache Mynewt Operating System Kernel

The Mynewt Core OS is a multitasking, preemptive real-time operating system combining a scheduler with typical RTOS features such as mutexes, semaphores, memory pools, etc. The Mynewt Core OS also provides a number of useful utilities such as a task watchdog, networking stack memory buffers and time management API. Each of these features is described in detail in its own section of the manual.

A multitasking, preemptive operating system is one in which a number of different tasks can be instantiated and assigned a priority, with higher priority tasks running before lower priority tasks. Furthermore, if a lower priority task is running and a higher priority task wants to run, the lower priority task is halted and the higher priority task is allowed to run. In other words, the lower priority task is preempted by the higher priority task.

Why use an OS?

You may ask yourself “why do I need a multitasking preemptive OS”? The answer may indeed be that you do not. Some applications are simple and only require a polling loop. Others are more complex and may require that certain jobs are executed in a timely manner or before other jobs are executed. If you have a simple polling loop, you cannot move on to service a job until the current job is done being serviced. With the Mynewt OS, the application developer need not worry about certain jobs taking too long or not executing in a timely fashion; the OS provides mechanisms to deal with these situations. Another benefit of using an OS is that it helps shield application developers from other application code being written; the developer does not have to worry (or has to worry less) about other application code behaving badly and causing undesirable behavior or preventing their code from executing properly. Other benefits of using an OS (and the Mynewt OS in particular) is that it also provides features that the developer would otherwise need to create on his/her own.

Core OS Features

- *Scheduler/context switching*
- *Time*
- *Tasks*
- *Event queues/callouts*
- *Semaphores*
- *Mutexes*
- *Memory pools*
- *Heap*
- *Mbufs*
- *Sanity*
- *Callouts*
- *Porting OS to other platforms*

Basic OS Application Creation

Creating an application using the Mynewt Core OS is a relatively simple task. The main steps are:

1. Install the basic Newt Tool structure (build structure) for your application.
2. Create your BSP (Board Support Package).
3. In your application `main()` function, call the `sysinit()` function to initialize the system and packages, perform application specific initialization, then wait and dispatch events from the OS default event queue in an infinite loop. (See [System Initialization and System Shutdown](#) for more details.)

Initializing application modules and tasks can get somewhat complicated with RTOS's similar to Mynewt. Care must be taken that the API provided by a task are initialized prior to being called by another (higher priority) task.

For example, take a simple application with two tasks (tasks 1 and 2, with task 1 higher priority than task 2). Task 2 provides an API which has a semaphore lock and this semaphore is initialized by task 2 when the task handler for task 2 is called. Task 1 is expected to call this API.

Consider the sequence of events when the OS is started. The scheduler starts running and picks the highest priority task (task 1 in this case). The task handler function for task 1 is called and will keep running until it yields execution. Before yielding, code in the task 1 handler function calls the API provided by task 2. The semaphore being accessed in the task 2 API has yet to be initialized since the task 2 handler function has not had a chance to run! This will lead to undesirable behavior and will need to be addressed by the application developer. Note that the Mynewt OS does guard against internal API being called before the OS has started (they will return error) but it does not safeguard application defined objects from access prior to initialization.

Example

One way to avoid initialization issues like the one described above is for the application to initialize the objects that are accessed by multiple tasks before it initializes the tasks with the OS.

The code example shown below illustrates this concept. The application initializes system and packages, calls an application specific “task initialization” function, and dispatches events from the default event queue. The application task initialization function is responsible for initializing all the data objects that each task exposes to the other tasks. The tasks themselves are also initialized at this time (by calling `os_task_init()`).

In the example, each task works in a ping-pong like fashion: task 1 wakes up, adds a token to semaphore 1 and then waits for a token from semaphore 2. Task 2 waits for a token on semaphore 1 and once it gets it, adds a token to semaphore 2. Notice that the semaphores are initialized by the application specific task initialization functions and not inside the task handler functions. If task 2 (being lower in priority than task 1) had called `os_sem_init()` for `task2_sem` inside `task2_handler()`, task 1 would have called `os_sem_pend()` using `task2_sem` before `task2_sem` was initialized.

```

struct os_sem task1_sem;
struct os_sem task2_sem;

/* Task 1 handler function */
void
task1_handler(void *arg)
{
    while (1) {
        /* Release semaphore to task 2 */
        os_sem_release(&task1_sem);

        /* Wait for semaphore from task 2 */
        os_sem_pend(&task2_sem, OS_TIMEOUT_NEVER);
    }
}

/* Task 2 handler function */
void
task2_handler(void *arg)
{
    struct os_task *t;

    while (1) {
        /* Wait for semaphore from task1 */
        os_sem_pend(&task1_sem, OS_TIMEOUT_NEVER);

        /* Release task2 semaphore */
        os_sem_release(&task2_sem);
    }
}

/* Initialize task 1 exposed data objects */
void
task1_init(void)
{
    /* Initialize task1 semaphore */

```

(continues on next page)

(continued from previous page)

```

        os_sem_init(&task1_sem, 0);
    }

/* Initialize task 2 exposed data objects */
void
task2_init(void)
{
    /* Initialize task1 semaphore */
    os_sem_init(&task2_sem, 0);
}

/*
 * init_app_tasks
 *
 * This function performs initializations that are required before tasks run.
 *
 * @return int 0 success; error otherwise.
 */
static int
init_app_tasks(void)
{
    /*
     * Call task specific initialization functions to initialize any shared objects
     * before initializing the tasks with the OS.
     */
    task1_init();
    task2_init();

    /*
     * Initialize tasks 1 and 2 with the OS.
     */
    os_task_init(&task1, "task1", task1_handler, NULL, TASK1_PRIO,
                 OS_WAIT_FOREVER, task1_stack, TASK1_STACK_SIZE);

    os_task_init(&task2, "task2", task2_handler, NULL, TASK2_PRIO,
                 OS_WAIT_FOREVER, task2_stack, TASK2_STACK_SIZE);

    return 0;
}

/*
 * main
 *
 * The main function for the application. This function initializes the system and
 * packages,
 * calls the application specific task initialization function, then waits and
 * dispatches
 * events from the OS default event queue in an infinite loop.
 */
int
main(int argc, char **arg)
{

```

(continues on next page)

(continued from previous page)

```

/* Perform system and package initialization */
sysinit();

/* Initialize application specific tasks */
init_app_tasks();

while (1) {
    os_eventq_run(os_eventq_dflt_get());
}
/* main never returns */
}

```

6.1.2 Scheduler

Scheduler's job is to maintain the list of tasks and decide which one should be running next.

Description

Task states can be *running*, *ready to run* or *sleeping*.

When task is *running*, CPU is executing in that task's context. The program counter (PC) is pointing to instructions task wants to execute and stack pointer (SP) is pointing to task's stack.

Task which is *ready to run* wants to get on the CPU to do its work.

Task which is *sleeping* has no more work to do. It's waiting for someone else to wake it up.

Scheduler algorithm is simple: from among the tasks which are ready to run, pick the the one with highest priority (lowest numeric value in task's t_prio field), and make its state *running*.

Tasks which are either *running* or *ready to run* are kept in linked list g_os_run_list. This list is ordered by priority.

Tasks which are *sleeping* are kept in linked list g_os_sleep_list.

Scheduler has a CPU architecture specific component; this code is responsible for swapping in the task which should be *running*. This process is called context switch. During context switching the state of the CPU (e.g. registers) for the currently *running* task is stored and the new task is swapped in.

API

struct *os_task* *os_sched_get_current_task(void)

Returns the currently running task.

Note that this task may or may not be the highest priority task ready to run.

Returns

The currently running task.

void os_sched_set_current_task(struct *os_task* *t)

Sets the currently running task to 't'.

Note that this function simply sets the global variable holding the currently running task. It does not perform a context switch or change the os run or sleep list.

Parameters

- **t** – Pointer to currently running task.

```
struct os_task *os_sched_next_task(void)
```

Returns the task that we should be running.

This is the task at the head of the run list.

 **Note**

If you want to guarantee that the os run list does not change after calling this function, you have to call it with interrupts disabled.

Returns

The task at the head of the list

```
void os_sched_suspend(void)
```

Suspend task scheduling.

Function suspends the scheduler. Suspending the scheduler prevents a context switch but leaves interrupts enabled. Call to [os_sched_resume\(\)](#) enables task scheduling again. Calls to [os_sched_suspend\(\)](#) can be nested. The same number of calls must be made to [os_sched_resume\(\)](#) as have previously been made to [os_sched_suspend\(\)](#) before task scheduling work again.

```
int os_sched_resume(void)
```

Resume task scheduling.

Resumes the scheduler after it was suspended with [os_sched_suspend\(\)](#).

Returns

0 when scheduling resumed; non-0 when scheduling is still locked and more calls to [os_sched_resume\(\)](#) are needed

```
void os_sched(struct os_task *next_t)
```

Performs context switch if needed.

If next_t is set, that task will be made running. If next_t is NULL, highest priority ready to run is swapped in. This function can be called when new tasks were made ready to run or if the current task is moved to sleeping state.

This function will call the architecture specific routine to swap in the new task.

```
// example
os_error_t
os_mutex_release(struct os_mutex *mu)
{
    ...
    OS_EXIT_CRITICAL(sr);

    // Re-schedule if needed
    if (resched) {
        os_sched(rdy);
    }

    return OS_OK;
}
```

(continues on next page)

(continued from previous page)

}

Note

Interrupts must be disabled when calling this.

Parameters

- **next_t** – Pointer to task which must run next (optional)

6.1.3 Task

A task, along with the scheduler, forms the basis of the Mynewt OS. A task consists of two basic elements: a task stack and a task function. The task function is basically a forever loop, waiting for some “event” to wake it up. There are two methods used to signal a task that it has work to do: event queues and semaphores (see the appropriate manual sections for descriptions of these features).

The Mynewt OS is a multi-tasking, preemptive OS. Every task is assigned a task priority (from 0 to 255), with 0 being the highest priority task. If a higher priority task than the current task wants to run, the scheduler preempts the currently running task and switches context to the higher priority task. This is just a fancy way of saying that the processor stack pointer now points to the stack of the higher priority task and the task resumes execution where it left off.

Tasks run to completion unless they are preempted by a higher priority task. The developer must insure that tasks eventually “sleep”; otherwise lower priority tasks will never get a chance to run (actually, any task lower in priority than the task that never sleeps). A task will be put to sleep in the following cases: it puts itself to sleep using `os_time_delay()`, it waits on an event queue which is empty or attempts to obtain a mutex or a semaphore that is currently owned by another task.

Note that other sections of the manual describe these OS features in more detail.

Description

In order to create a task two data structures need to be defined: the task object (struct `os_task`) and its associated stack. Determining the stack size can be a bit tricky; generally developers should not declare large local variables on the stack so that task stacks can be of limited size. However, all applications are different and the developer must choose the stack size accordingly. NOTE: be careful when declaring your stack! The stack is in units of `os_stack_t` sized elements (generally 32-bits). Looking at the example given below and assuming `os_stack_t` is defined to be a 32-bit unsigned value, “my_task_stack” will use 256 bytes.

A task must also have an associated “task function”. This is the function that will be called when the task is first run. This task function should never return!

In order to inform the Mynewt OS of the new task and to have it added to the scheduler, the `os_task_init()` function is called. Once `os_task_init()` is called, the task is made ready to run and is added to the active task list. Note that a task can be initialized (started) before or after the os has started (i.e. before `os_start()` is called) but must be initialized after the os has been initialized (i.e. `os_init()` has been called). In most of the examples and current Mynewt projects, the os is initialized, tasks are initialized, and the os is started. Once the os has started, the highest priority task will be the first task set to run.

Information about a task can be obtained using the `os_task_info_get_next()` API. Developers can walk the list of tasks to obtain information on all created tasks. This information is of type `os_task_info`.

The following is a very simple example showing a single application task. This task simply toggles an LED at a one second interval.

```
/* Create a simple "project" with a task that blinks a LED every second */

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

/* This is the task function */
void my_task_func(void *arg) {
    /* Set the led pin as an output */
    hal_gpio_init_out(LED_BLINK_PIN, 1);

    /* The task is a forever loop that does not return */
    while (1) {
        /* Wait one second */
        os_time_delay(1000);

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

/* This is the main function for the project */
int main(int argc, char **argv)
{
    /* Perform system and package initialization */
    sysinit();

    /* 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);

    /* Process events from the default event queue. */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }
    /* main never returns */
}
```

API

enum os_task_state

Task states.

Values:

enumerator **OS_TASK_READY**

Task is ready to run.

enumerator **OS_TASK_SLEEP**

Task is sleeping.

typedef enum *os_task_state* os_task_state_t

Task states.

typedef void (*os_task_func_t)(void *arg)

Type definition for a task callback function.

Param arg

Argument passed to the task function.

int 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)

Initialize a task.

This function initializes the task structure pointed to by t, clearing and setting it's stack pointer, provides sane defaults and sets the task as ready to run, and inserts it into the operating system scheduler.

Parameters

- **t** – The task to initialize
- **name** – The name of the task to initialize
- **func** – The task function to call
- **arg** – The argument to pass to this task function
- **prio** – The priority at which to run this task
- **sanity_itvl** – The time at which this task should check in with the sanity task. OS_WAIT_FOREVER means never check in here.
- **stack_bottom** – A pointer to the bottom of a task's stack
- **stack_size** – The overall size of the task's stack.

Returns

0 on success; non-zero on failure.

int os_task_remove(struct *os_task* *t)

Removes specified task.

Note

This interface is currently experimental and not ready for common use

Parameters

- **t** – The task to remove

Returns

0 on success; non-zero on failure

`os_stack_t *os_task_stacktop_get(struct os_task *t)`

Return a pointer to the top of the stack for a given task.

Parameters

- **t** – The task to get the top of the stack from

Returns

A pointer to the top of the stack

`uint8_t os_task_count(void)`

Return the number of tasks initialized.

Returns

The number of tasks initialized

`struct os_task *os_task_info_get_next(const struct os_task *prev, struct os_task_info *oti)`

Iterate through tasks, and return the following information about them:

- Priority
- Task ID
- State (READY, SLEEP)
- Total Stack Usage
- Stack Size
- Context Switch Count
- Runtime
- Last & Next Sanity checkin
- Task Name

To get the first task in the list, call `os_task_info_get_next()` with a NULL pointer in the prev argument, and `os_task_info_get_next()` will return a pointer to the task structure, and fill out the `os_task_info` structure pointed to by oti.

To get the next task in the list, provide the task structure returned by the previous call to `os_task_info_get_next()`, and `os_task_info_get_next()` will fill out the task structure pointed to by oti again, and return the next task in the list.

Parameters

- **prev** – The previous task returned by `os_task_info_get_next()`, or NULL to begin iteration.
- **oti** – The OS task info structure to fill out.

Returns

A pointer to the OS task that has been read; NULL when finished iterating through all tasks.

```
void os_task_info_get(const struct os_task *task, struct os_task_info *oti)
```

Get following info about specified task.

- Priority
- Task ID
- State (READY, SLEEP)
- Total Stack Usage
- Stack Size
- Context Switch Count
- Runtime
- Last & Next Sanity checkin
- Task Name

Parameters

- **task** – The task to get info about
- **oti** – The OS task info structure to fill out.

```
OS_TASK_STACK_DEFINE_NOSTATIC(__name, __size)
```

Defines a non-static task stack.

Parameters

- **__name** – The name of the stack.
- **__size** – The size of the stack in bytes.

```
OS_TASK_STACK_DEFINE(__name, __size)
```

Defines a static task stack.

Parameters

- **__name** – The name of the stack.
- **__size** – The size of the stack in bytes.

```
OS_TASK_PRI_HIGHEST
```

Highest priority task.

```
OS_TASK_PRI_LOWEST
```

Lowest priority task.

```
OS_TASK_FLAG_NO_TIMEOUT
```

Task waiting forever.

```
OS_TASK_FLAG_SEM_WAIT
```

Task waiting on a semaphore.

OS_TASK_FLAG_MUTEX_WAIT

Task waiting on a mutex.

OS_TASK_FLAG_EVQ_WAIT

Task waiting on a event queue.

OS_TASK_MAX_NAME_LEN

Maximum length of a task name.

struct os_task_obj

#include <os_task.h> Generic “object” structure.

All objects that a task can wait on must have a *SLIST_HEAD(, os_task)* head_name as the first element in the object structure. The element ‘head_name’ can be any name. See os_mutex.h or os_sem.h for an example.

Public Functions**SLIST_HEAD (, os_task) obj_head**

Head of the list of waiting tasks.

struct os_task

#include <os_task.h> Structure containing information about a running task.

Public Functions**STAILQ_ENTRY (os_task) t_os_task_list**

Entry for a singly-linked task list.

TAILQ_ENTRY (os_task) t_os_list

Entry for a doubly-linked list of tasks managed by the scheduler.

SLIST_ENTRY (os_task) t_obj_list

Entry for a singly-linked object list.

Public Members**os_stack_t *t_stackptr**

Current stack pointer for this task.

os_stack_t *t_stackbottom

Pointer to bottom of this task’s stack.

uint16_t t_stacksize

Size of this task’s stack.

```
uint8_t t_taskid
    Task ID.

uint8_t t_prio
    Task Priority.

uint8_t t_state
    Task state, either READY or SLEEP.

uint8_t t_flags
    Task flags, bitmask.

uint8_t t_lockcnt
    Task lock count.

uint8_t t_pad
    Padding for alignment.

const char *t_name
    Task name.

os_task_func_t t_func
    Task function that executes.

void *t_arg
    Argument to pass to task function when called.

union os_task
    Current object task is waiting on, either a semaphore or mutex.

    struct os_sanity_check t_sanity_check
        Default sanity check for this task.

    os_time_t t_next_wakeup
        Next scheduled wakeup if this task is sleeping.

    os_time_t t_run_time
        Total task run time.

    uint32_t t_ctx_sw_cnt
        Total number of times this task has been context switched during execution.

struct os_task_info
    #include <os_task.h> Information about an individual task, returned for management APIs.
```

Public Members

`uint8_t oti_prio`

Task priority.

`uint8_t oti_taskid`

Task identifier.

`uint8_t oti_state`

Task state, either READY or SLEEP.

`uint16_t oti_stkusage`

Task stack usage.

`uint16_t oti_stksize`

Task stack size.

`uint32_t oti_cswcnt`

Task context switch count.

`uint32_t oti_runtim`

Task runtime.

`os_time_t oti_last_checkin`

Last time this task checked in with sanity.

`os_time_t oti_next_checkin`

Next time this task is scheduled to check-in with sanity.

`char oti_name[OS_TASK_MAX_NAME_LEN]`

Name of this task.

6.1.4 Mutex

Mutex is short for “mutual exclusion”; a mutex provides mutually exclusive access to a shared resource. A mutex provides *priority inheritance* in order to prevent *priority inversion*. Priority inversion occurs when a higher priority task is waiting on a resource owned by a lower priority task. Using a mutex, the lower priority task will inherit the highest priority of any task waiting on the mutex.

Description

The first order of business when using a mutex is to declare the mutex globally. The mutex needs to be initialized before it is used (see the examples). It is generally a good idea to initialize the mutex before tasks start running in order to avoid a task possibly using the mutex before it is initialized.

When a task wants exclusive access to a shared resource it needs to obtain the mutex by calling `os_mutex_pend()`. If the mutex is currently owned by a different task (a lower priority task), the requesting task will be put to sleep and the owners priority will be elevated to the priority of the requesting task. Note that multiple tasks can request ownership and the current owner is elevated to the highest priority of any task waiting on the mutex. When the task is done using the shared resource, it needs to release the mutex by called `os_mutex_release()`. There needs to be one release per call to pend. Note that nested calls to `os_mutex_pend()` are allowed but there needs to be one release per pend.

The following example will illustrate how priority inheritance works. In this example, the task number is the same as its priority. Remember that the lower the number, the higher the priority (i.e. priority 0 is higher priority than priority 1). Suppose that task 5 gets ownership of a mutex but is preempted by task 4. Task 4 attempts to gain ownership of the mutex but cannot as it is owned by task 5. Task 4 is put to sleep and task 5 is temporarily raised to priority 4. Before task 5 can release the mutex, task 3 runs and attempts to acquire the mutex. At this point, both task 3 and task 4 are waiting on the mutex (sleeping). Task 5 now runs at priority 3 (the highest priority of all the tasks waiting on the mutex). When task 5 finally releases the mutex it will be preempted as two higher priority tasks are waiting for it.

Note that when multiple tasks are waiting on a mutex owned by another task, once the mutex is released the highest priority task waiting on the mutex is run.

API

`os_error_t os_mutex_init(struct os_mutex *mu)`

Create a mutex and initialize it.

Parameters

- `mu` – Pointer to mutex

Returns

`os_error_t OS_INVALID_PARM` Mutex passed in was NULL. `OS_OK` no error.

`os_error_t os_mutex_release(struct os_mutex *mu)`

Release a mutex.

Parameters

- `mu` – Pointer to the mutex to be released

Returns

`os_error_t OS_INVALID_PARM` Mutex passed in was NULL. `OS_BAD_MUTEX` Mutex was not granted to current task (not owner). `OS_OK` No error

`os_error_t os_mutex_pend(struct os_mutex *mu, os_time_t timeout)`

Pend (wait) for a mutex.

Parameters

- `mu` – Pointer to mutex.
- `timeout` – Timeout, in os ticks. A timeout of 0 means do not wait if not available. A timeout of `OS_TIMEOUT_NEVER` means wait forever.

Returns

`os_error_t OS_INVALID_PARM` Mutex passed in was NULL. `OS_TIMEOUT` Mutex was owned by another task and `timeout=0` `OS_OK` no error.

```
static inline uint16_t os_mutex_get_level(struct os_mutex *mu)
```

Get mutex lock count.

It can also be called from interrupt context to check if given mutex is taken.

 **Note**

Function should be called from task owning the mutex (one that successfully called os_mutex_pend). Calling function from other task that does not own the mutex will return value that has little value to the caller since value can change at any time by other task.

Parameters

- **mu** – Pointer to mutex.

Returns

number of times lock was called from current task

```
struct os_mutex
```

```
#include <os_mutex.h> OS mutex structure.
```

Public Functions

```
SLIST_HEAD (, os_task) mu_head
```

Mutex head.

Public Members

```
uint8_t _pad
```

Padding for alignment.

```
uint8_t mu_prio
```

Mutex owner's default priority.

```
uint16_t mu_level
```

Mutex call nesting level.

```
struct os_task *mu_owner
```

Task that owns the mutex.

6.1.5 Semaphore

A semaphore is a structure used for gaining exclusive access (much like a mutex), synchronizing task operations and/or use in a “producer/consumer” roles. Semaphores like the ones used by the myNewt OS are called “counting” semaphores as they are allowed to have more than one token (explained below).

Description

A semaphore is a fairly simple construct consisting of a queue for waiting tasks and the number of tokens currently owned by the semaphore. A semaphore can be obtained as long as there are tokens in the semaphore. Any task can add tokens to the semaphore and any task can request the semaphore, thereby removing tokens. When creating the semaphore, the initial number of tokens can be set as well.

When used for exclusive access to a shared resource the semaphore only needs a single token. In this case, a single task “creates” the semaphore by calling `os_sem_init()` with a value of one (1) for the token. When a task desires exclusive access to the shared resource it requests the semaphore by calling `os_sem_pend()`. If there is a token the requesting task will acquire the semaphore and continue operation. If no tokens are available the task will be put to sleep until there is a token. A common “problem” with using a semaphore for exclusive access is called *priority inversion*. Consider the following scenario: a high and low priority task both share a resource which is locked using a semaphore. If the low priority task obtains the semaphore and then the high priority task requests the semaphore, the high priority task is now blocked until the low priority task releases the semaphore. Now suppose that there are tasks between the low priority task and the high priority task that want to run. These tasks will preempt the low priority task which owns the semaphore. Thus, the high priority task is blocked waiting for the low priority task to finish using the semaphore but the low priority task cannot run since other tasks are running. Thus, the high priority tasks is “inverted” in priority; in effect running at a much lower priority as normally it would preempt the other (lower priority) tasks. If this is an issue a mutex should be used instead of a semaphore.

Semaphores can also be used for task synchronization. A simple example of this would be the following. A task creates a semaphore and initializes it with no tokens. The task then waits on the semaphore, and since there are no tokens, the task is put to sleep. When other tasks want to wake up the sleeping task they simply add a token by calling `os_sem_release()`. This will cause the sleeping task to wake up (instantly if no other higher priority tasks want to run).

The other common use of a counting semaphore is in what is commonly called a “producer/consumer” relationship. The producer adds tokens (by calling `os_sem_release()`) and the consumer consumes them by calling `os_sem_pend()`. In this relationship, the producer has work for the consumer to do. Each token added to the semaphore will cause the consumer to do whatever work is required. A simple example could be the following: every time a button is pressed there is some work to do (ring a bell). Each button press causes the producer to add a token. Each token consumed rings the bell. There will exactly the same number of bell rings as there are button presses. In other words, each call to `os_sem_pend()` subtracts exactly one token and each call to `os_sem_release()` adds exactly one token.

API

`os_error_t os_sem_init(struct os_sem *sem, uint16_t tokens)`

Initialize a semaphore.

Parameters

- **sem** – Pointer to semaphore
- **tokens** – # of tokens the semaphore should contain initially.

Returns

`os_error_t OS_INVALID_PARM` Semaphore passed in was NULL. `OS_OK` no error.

`os_error_t os_sem_release(struct os_sem *sem)`

Release a semaphore.

Parameters

- **sem** – Pointer to the semaphore to be released

Returns

`os_error_t OS_INVALID_PARM` Semaphore passed in was NULL. `OS_OK` No error

`os_error_t os_sem_pend(struct os_sem *sem, os_time_t timeout)`

Pend (wait) for a semaphore.

Parameters

- **sem** – Pointer to semaphore.
- **timeout** – Timeout, in os ticks. A timeout of 0 means do not wait if not available. A timeout of `OS_TIMEOUT_NEVER` means wait forever.

Returns

`os_error_t OS_INVALID_PARM` Semaphore passed in was NULL. `OS_TIMEOUT` Semaphore was owned by another task and `timeout=0` `OS_OK` no error.

`static inline uint16_t os_sem_get_count(struct os_sem *sem)`

Get current semaphore's count.

`struct os_sem`

`#include <os_sem.h>` Structure representing an OS semaphore.

Public Functions

`SLIST_HEAD (, os_task) sem_head`

Semaphore head.

Public Members

`uint16_t _pad`

Padding for alignment.

`uint16_t sem_tokens`

Number of tokens.

6.1.6 Event Queues

An event queue allows a task to serialize incoming events and simplify event processing. Events are stored in a queue and a task removes and processes an event from the queue. An event is processed in the context of this task. Events may be generated by OS callouts, interrupt handlers, and other tasks.

Description

Mynewt's event queue model uses callback functions to process events. Each event is associated with a callback function that is called to process the event. This model enables a library package, that uses events in its implementation but does not have real-time timing requirements, to use an application event queue instead of creating a dedicated event queue and task to process its events. The callback function executes in the context of the task that the application creates to manage the event queue. This model reduces an application's memory requirement because memory must be allocated for the task's stack when a task is created. A package that has real-time timing requirements and must run at a specific task priority should create a dedicated event queue and task to process its events.

In the Mynewt model, a package defines its events and implements the callback functions for the events. A package that does not have real-time timing requirements should use Mynewt's default event queue for its events. The callback function for an event from the Mynewt default event queue is executed in the context of the application main task. A package can, optionally, export a function that allows an application to specify the event queue for the package to use. The application task handler that manages the event queue simply pulls events from the event queue and executes the event's callback function in its context.

A common way that Mynewt applications or packages process events from an event queue is to have a task that executes in an infinite loop and calls the `os_eventq_get()` function to dequeue and return the event from the head of the event queue. The task then calls the event callback function to process the event. The `os_eventq_get()` function puts the task in to the `sleeping` state when there are no events on the queue. (See *Scheduler* for more information on task execution states.) Other tasks (or interrupts) call the `os_eventq_put()` function to add an event to the queue. The `os_eventq_put()` function determines whether a task is blocked waiting for an event on the queue and puts the task into the `ready-to-run` state.

A task can use the `os_eventq_run()` wrapper function that calls the `os_eventq_get()` function to dequeue an event from the queue and then calls the event callback function to process the event. Note:

- Only one task should consume or block waiting for events from an event queue.
- The OS callout subsystem uses events for timer expiration notification.

Example

Here is an example of using an event from the BLE host:

```
static void ble_hs_event_tx_notify(struct os_event *ev);

/** OS event - triggers tx of pending notifications and indications. */
static struct os_event ble_hs_ev_tx_notifications = {
    .ev_cb = ble_hs_event_tx_notify,
};
```

API

`typedef void os_event_fn(struct os_event *ev)`

Callback function type for handling events.

`void os_eventq_init(struct os_eventq *evq)`

Initialize the event queue.

Parameters

- `evq` – The event queue to initialize

int **os_eventq_inited**(const struct *os_eventq* *evq)

Check whether the event queue is initialized.

Parameters

- **evq** – The event queue to check

Returns

Non-zero if the event queue is initialized; zero otherwise.

void **os_eventq_put**(struct *os_eventq* *evq, struct *os_event* *ev)

Put an event on the event queue.

Parameters

- **evq** – The event queue to put an event on
- **ev** – The event to put on the queue

struct *os_event* ***os_eventq_get_no_wait**(struct *os_eventq* *evq)

Poll an event from the event queue and return it immediately.

If no event is available, don't block, just return NULL.

Parameters

- **evq** – The event queue to pull an event from

Returns

The event from the queue; NULL if none available.

struct *os_event* ***os_eventq_get**(struct *os_eventq* *evq)

Pull a single item from an event queue.

This function blocks until there is an item on the event queue to read.

Parameters

- **evq** – The event queue to pull an event from

Returns

The event from the queue

void **os_eventq_run**(struct *os_eventq* *evq)

Pull a single item off the event queue and call its event callback.

Parameters

- **evq** – The event queue to pull the item off.

struct *os_event* ***os_eventq_poll**(struct *os_eventq* **evq, int nevqs, *os_time_t* timo)

Poll the list of event queues specified by the evq parameter (size nevqs), and return the “first” event available on any of the queues.

Event queues are searched in the order that they are passed in the array.

Parameters

- **evq** – Array of event queues
- **nevqs** – Number of event queues in evq
- **timo** – Timeout, forever if OS_WAIT_FOREVER is passed to poll.

Returns

The first available event; NULL if no events available

```
void os_eventq_remove(struct os_eventq *evq, struct os_event *ev)
```

Remove an event from the queue.

Parameters

- **evq** – The event queue to remove the event from
- **ev** – The event to remove from the queue

```
struct os_eventq *os_eventq_dflt_get(void)
```

Retrieves the default event queue processed by OS main task.

Returns

The default event queue.

```
static inline void os_eventq_mon_start(struct os_eventq *evq, int cnt, struct os_eventq_mon *mon)
```

Instrument OS eventq to monitor time spent handling events.

Parameters

- **evq** – The event queue to start monitoring
- **cnt** – How many elements can be used in monitoring.
- **mon** – Pointer to data where monitoring data is collected. Must hold cnt number of elements.

```
static inline void os_eventq_mon_stop(struct os_eventq *evq)
```

Stop OS eventq monitoring.

Parameters

- **evq** – The event queue this operation applies to

OS_EVENT_QUEUED(__ev)

Return whether or not the given event is queued.

```
struct os_event
```

#include <os_eventq.h> Structure representing an OS event.

OS events get placed onto the event queues and are consumed by tasks.

Public Functions

STAILQ_ENTRY (os_event) ev_next

Next event in the queue.

Public Members

uint8_t ev_queued

Whether this OS event is queued on an event queue.

os_event_fn *ev_cb

Callback to call when the event is taken off of an event queue.

APIs, except for *os_eventq_run()*, assume this callback will be called by the user.

```
void *ev_arg
```

Argument to pass to the event queue callback.

```
struct os_eventq_mon
```

```
#include <os_eventq.h>
```

Structure keeping track of time spent inside event callback.

This is stored per eventq, and is updated inside [*os_eventq_run\(\)*](#). Tick unit is os_cputime.

```
struct os_eventq
```

```
#include <os_eventq.h>
```

Structure representing an event queue.

Public Functions

```
STAILQ_HEAD (, os_event) evq_list
```

Event queue list.

Public Members

```
struct os_task *evq_owner
```

Pointer to task that “owns” this event queue.

```
struct os_task *evq_task
```

Pointer to the task that is sleeping on this event queue, either NULL, or the owner task.

```
struct os_event *evq_prev
```

Most recently processed event.

6.1.7 Callout

Callouts are Apache Mynewt OS timers.

Description

Callout is a way of setting up an OS timer. When the timer fires, it is delivered as an event to task’s event queue.

User would initialize their callout structure `struct os_callout` using [*os_callout_init\(\)*](#) and then arm it with [*os_callout_reset\(\)*](#).

If user wants to cancel the timer before it expires, they can either use [*os_callout_reset\(\)*](#) to arm it for later expiry, or stop it altogether by calling [*os_callout_stop\(\)*](#).

Time unit when arming the timer is OS ticks. This rate of this ticker depends on the platform this is running on. You should use OS define OS_TICKS_PER_SEC to convert wallclock time to OS ticks.

Callout timer fires out just once. For periodic timer type of operation you need to rearm it once it fires.

API

`void os_callout_init(struct os_callout *c, struct os_eventq *evq, os_event_fn *ev_cb, void *ev_arg)`

Initialize a callout.

Callouts are used to schedule events in the future onto a task's event queue. Callout timers are scheduled using the `os_callout_reset()` function. When the timer expires, an event is posted to the event queue specified in `os_callout_init()`. The event argument given here is posted in the `ev_arg` field of that event.

Parameters

- **c** – The callout to initialize
- **evq** – The event queue to post an OS_EVENT_T_TIMER event to
- **ev_cb** – The function to call on this callout for the host task used to provide multiple timer events to a task (this can be NULL).
- **ev_arg** – The argument to provide to the event when posting the timer.

`void os_callout_stop(struct os_callout *c)`

Stop the callout from firing off, any pending events will be cleared.

Parameters

- **c** – The callout to stop

`int os_callout_reset(struct os_callout *c, os_time_t ticks)`

Reset the callout to fire off in 'ticks' ticks.

Parameters

- **c** – The callout to reset
- **ticks** – The number of ticks to wait before posting an event

Returns

0 on success; non-zero on failure

`os_time_t os_callout_remaining_ticks(struct os_callout *c, os_time_t now)`

Returns the number of ticks which remains to callout.

Parameters

- **c** – The callout to check
- **now** – The current time in OS ticks

Returns

Number of ticks to first pending callout

`static inline int os_callout_queued(struct os_callout *c)`

Returns whether the callout is pending or not.

Parameters

- **c** – The callout to check

Returns

1 if queued; 0 if not queued.

`struct os_callout`

`#include <os_callout.h>` Structure containing the definition of a callout, initialized by `os_callout_init()` and passed to callout functions.

Public Functions

`TAILQ_ENTRY (os_callout) c_next`

Next callout in the list.

Public Members

`struct os_event c_ev`

Event to post when the callout expires.

`struct os_eventq *c_evq`

Pointer to the event queue to post the event to.

`os_time_t c_ticks`

Number of ticks in the future to expire the callout.

6.1.8 Heap

API for doing dynamic memory allocation.

Description

This provides malloc()/free() functionality with locking. The shared resource heap needs to be protected from concurrent access when OS has been started. `os_malloc()` function grabs a mutex before calling malloc().

API

`void *os_malloc(size_t size)`

Operating system level malloc().

This ensures that a safe malloc occurs within the context of the OS. Depending on platform, the OS may rely on libc's malloc() implementation, which is not guaranteed to be thread-safe. This malloc() will always be thread-safe.

Parameters

- `size` – The number of bytes to allocate

Returns

A pointer to the memory region allocated.

`void os_free(void *mem)`

Operating system level free().

See description of `os_malloc()` for reasoning.

Free's memory allocated by malloc.

Parameters

- `mem` – The memory to free.

```
void *os_realloc(void *ptr, size_t size)
```

Operating system level realloc().

See description of [os_malloc\(\)](#) for reasoning.

Reallocates the memory at ptr, to be size contiguous bytes.

Parameters

- **ptr** – A pointer to the memory to allocate
- **size** – The number of contiguous bytes to allocate at that location

Returns

A pointer to memory of size, or NULL on failure to allocate

6.1.9 Memory Pools

A memory pool is a collection of fixed sized elements called memory blocks. Generally, memory pools are used when the developer wants to allocate a certain amount of memory to a given feature. Unlike the heap, where a code module is at the mercy of other code modules to insure there is sufficient memory, memory pools can insure sufficient memory allocation.

Description

In order to create a memory pool the developer needs to do a few things. The first task is to define the memory pool itself. This is a data structure which contains information about the pool itself (i.e. number of blocks, size of the blocks, etc).

```
struct os_mempool my_pool;
```

The next order of business is to allocate the memory used by the memory pool. This memory can either be statically allocated (i.e. a global variable) or dynamically allocated (i.e. from the heap). When determining the amount of memory required for the memory pool, simply multiplying the number of blocks by the size of each block is not sufficient as the OS may have alignment requirements. The alignment size definition is named OS_ALIGNMENT and can be found in os_arch.h as it is architecture specific. The memory block alignment is usually for efficiency but may be due to other reasons. Generally, blocks are aligned on 32-bit boundaries. Note that memory blocks must also be of sufficient size to hold a list pointer as this is needed to chain memory blocks on the free list.

In order to simplify this for the user two macros have been provided: c:macro:OS_MEMPOOL_BYTES(*n, blksize*) and OS_MEMPOOL_SIZE(*n, blksize*). The first macro returns the number of bytes needed for the memory pool while the second returns the number of `os_membuf_t` elements required by the memory pool. The `os_membuf_t` type is used to guarantee that the memory buffer used by the memory pool is aligned on the correct boundary.

Here are some examples. Note that if a custom malloc implementation is used it must guarantee that the memory buffer used by the pool is allocated on the correct boundary (i.e. OS_ALIGNMENT).

```
void *my_memory_buffer;
my_memory_buffer = malloc(OS_MEMPOOL_BYTES(NUM_BLOCKS, BLOCK_SIZE));
```

```
os_membuf_t my_memory_buffer[OS_MEMPOOL_SIZE(NUM_BLOCKS, BLOCK_SIZE)];
```

Now that the memory pool has been defined as well as the memory required for the memory blocks which make up the pool the user needs to initialize the memory pool by calling `os_mempool_init()`.

```
os_mempool_init(&my_pool, NUM_BLOCKS, BLOCK_SIZE, my_memory_buffer,  
                 "MyPool");
```

Once the memory pool has been initialized the developer can allocate memory blocks from the pool by calling `os_memblock_get()`. When the memory block is no longer needed the memory can be freed by calling `os_memblock_put()`.

API

typedef os_error_t **os_mempool_put_fn**(struct *os_mempool_ext* *ome, void *data, void *arg)

Block put callback function.

If configured, this callback gets executed whenever a block is freed to the corresponding extended mempool.

Note: The `os_memblock_put()` function calls this callback instead of freeing the block itself. Therefore, it is the callback's responsibility to free the block via a call to `os_memblock_put_from_cb()`.

Param ome

The extended mempool that a block is being freed back to.

Param data

The block being freed.

Param arg

Optional argument configured along with the callback.

Return

Indicates whether the block was successfully freed. A non-zero value should only be returned if the block was not successfully released back to its pool.

struct *os_mempool* ***os_mempool_info_get_next**(struct *os_mempool* *mp, struct *os_mempool_info* *omi)

Get information about the next system memory pool.

Parameters

- **mp** – The current memory pool, or NULL if starting iteration.
- **omi** – A pointer to the structure to return memory pool information into.

Returns

The next memory pool in the list to get information about; NULL when at the last memory pool.

struct *os_mempool* ***os_mempool_get**(const char *mempool_name, struct *os_mempool_info* *info)

Get information system memory pool by name.

Parameters

- **mempool_name** – The name of mempool.
- **info** – A pointer to the structure to return memory pool information into, can be NULL.

Returns

The memory pool found; NULL when there is no such memory pool.

os_error_t **os_mempool_init**(struct *os_mempool* *mp, uint16_t blocks, uint32_t block_size, void *membuf, char *name)

Initialize a memory pool.

Parameters

- **mp** – Pointer to a pointer to a mempool

- **blocks** – The number of blocks in the pool
- **block_size** – The size of the block, in bytes.
- **mdbuf** – Pointer to memory to contain blocks.
- **name** – Name of the pool.

Returns

0 on success; Non-zero error code on failure.

```
os_error_t os_mempool_ext_init(struct os_mempool_ext *mpe, uint16_t blocks, uint32_t block_size, void
                                *dbuf, char *name)
```

Initializes an extended memory pool.

Extended attributes (e.g., callbacks) are not specified when this function is called; they are assigned manually after initialization.

Parameters

- **mpe** – The extended memory pool to initialize.
- **blocks** – The number of blocks in the pool.
- **block_size** – The size of each block, in bytes.
- **dbuf** – Pointer to memory to contain blocks.
- **name** – Name of the pool.

Returns

0 on success; Non-zero error code on failure.

```
os_error_t os_mempool_unregister(struct os_mempool *mp)
```

Removes the specified mempool from the list of initialized mempools.

Parameters

- **mp** – The mempool to unregister.

Returns

0 on success; OS_INVALID_PARM if the mempool is not registered.

```
os_error_t os_mempool_clear(struct os_mempool *mp)
```

Clears a memory pool.

Parameters

- **mp** – The mempool to clear.

Returns

0 on success; Non-zero error code on failure.

```
bool os_mempool_is_sane(const struct os_mempool *mp)
```

Performs an integrity check of the specified mempool.

This function attempts to detect memory corruption in the specified memory pool.

Parameters

- **mp** – The mempool to check.

Returns

true if the memory pool passes the integrity check; false if the memory pool is corrupt.

`int os_memblock_from(const struct os_mempool *mp, const void *block_addr)`

Checks if a memory block was allocated from the specified mempool.

Parameters

- **mp** – The mempool to check as parent.
- **block_addr** – The memory block to check as child.

Returns

0 if the block does not belong to the mempool; 1 if the block does belong to the mempool.

`void *os_memblock_get(struct os_mempool *mp)`

Get a memory block from a memory pool.

Parameters

- **mp** – Pointer to the memory pool

Returns

Pointer to block if available; NULL otherwise

`os_error_t os_memblock_put_from_cb(struct os_mempool *mp, void *block_addr)`

Puts the memory block back into the pool, ignoring the put callback, if any.

This function should only be called from a put callback to free a block without causing infinite recursion.

Parameters

- **mp** – Pointer to memory pool
- **block_addr** – Pointer to memory block

Returns

0 on success; Non-zero error code on failure.

`os_error_t os_memblock_put(struct os_mempool *mp, void *block_addr)`

Puts the memory block back into the pool.

Parameters

- **mp** – Pointer to memory pool
- **block_addr** – Pointer to memory block

Returns

0 on success; Non-zero error code on failure.

OS_MEMPOOL_F_EXT

Indicates an extended mempool.

Address can be safely cast to (`struct os_mempool_ext *`).

OS_MEMPOOL_INFO_NAME_LEN

Length of the name of memory pool.

OS_MEMPOOL_BLOCK_SZ(sz)

Leave extra 4 bytes of guard area at the end.

OS_MEMPOOL_SIZE(n, blksize)

The total size of a memory pool, including alignment.

OS_MEMPOOL_BYTES(n, blksize)

Calculates the number of bytes required to initialize a memory pool.

struct **os_memblock**

#include <os_mempool.h> A memory block structure.

This simply contains a pointer to the free list chain and is only used when the block is on the free list. When the block has been removed from the free list the entire memory block is usable by the caller.

Public Functions

SLIST_ENTRY (os_memblock) mb_next

Next memory block in the list.

struct **os_mempool**

#include <os_mempool.h> Memory pool.

Public Functions

STAILQ_ENTRY (os_mempool) mp_list

Next memory pool in the list.

SLIST_HEAD (, os_memblock)

Head of the list of memory blocks.

Public Members

uint32_t mp_block_size

Size of the memory blocks, in bytes.

uint16_t mp_num_blocks

The number of memory blocks.

uint16_t mp_num_free

The number of free blocks left.

uint16_t mp_min_free

The lowest number of free blocks seen.

uint8_t mp_flags

Bitmap of OS_MEMPOOL_F_[...] values.

uintptr_t mp_membuf_addr

Address of memory buffer used by pool.

```
char *name  
Name for memory block.
```

```
struct os_mempool_ext  
#include <os_mempool.h> Extended memory pool.
```

Public Members

```
struct os_mempool mpe_mp
```

Standard memory pool.

```
os_mempool_put_fn *mpe_put_cb
```

Callback that is executed immediately when a block is freed.

```
void *mpe_put_arg
```

Optional argument passed to the callback function.

```
struct os_mempool_info
```

```
#include <os_mempool.h> Information describing a memory pool, used to return OS information to the management layer.
```

Public Members

```
int omi_block_size
```

Size of the memory blocks in the pool.

```
int omi_num_blocks
```

Number of memory blocks in the pool.

```
int omi_num_free
```

Number of free memory blocks.

```
int omi_min_free
```

Minimum number of free memory blocks ever.

```
char omi_name[OS_MEMPOOL_INFO_NAME_LEN]
```

Name of the memory pool.

6.1.10 Mbufs

The mbuf (short for memory buffer) is a common concept in networking stacks. The mbuf is used to hold packet data as it traverses the stack. The mbuf also generally stores header information or other networking stack information that is carried around with the packet. The mbuf and its associated library of functions were developed to make common networking stack operations (like stripping and adding protocol headers) efficient and as copy-free as possible.

In its simplest form, an mbuf is a memory block with some space reserved for internal information and a pointer which is used to “chain” memory blocks together in order to create a “packet”. This is a very important aspect of the mbuf: the ability to chain mbufs together to create larger “packets” (chains of mbufs).

Why use mbufs?

The main reason is to conserve memory. Consider a networking protocol that generally sends small packets but occasionally sends large ones. The Bluetooth Low Energy (BLE) protocol is one such example. A flat buffer would need to be sized so that the maximum packet size could be contained by the buffer. With the mbuf, a number of mbufs can be chained together so that the occasional large packet can be handled while leaving more packet buffers available to the networking stack for smaller packets.

Packet Header mbuf

Not all mbufs are created equal. The first mbuf in a chain of mbufs is a special mbuf called a “packet header mbuf”. The reason that this mbuf is special is that it contains the length of all the data contained by the chain of mbufs (the packet length, in other words). The packet header mbuf may also contain a user defined structure (called a “user header”) so that networking protocol specific information can be conveyed to various layers of the networking stack. Any mbufs that are part of the packet (i.e. in the mbuf chain but not the first one) are “normal” (i.e. non-packet header) mbufs. A normal mbuf does not have any packet header or user packet header structures in them; they only contain the basic mbuf header (`struct os_mbuf`). Figure 1 illustrates these two types of mbufs. Note that the numbers/text in parentheses denote the size of the structures/elements (in bytes) and that MBLLEN is the memory block length of the memory pool used by the mbuf pool.

Normal mbuf

Now let’s take a deeper dive into the mbuf structure. Figure 2 illustrates a normal mbuf and breaks out the various fields in the `c:type:os_mbuf` structure.

- The `om_data` field is a pointer to where the data starts inside the data buffer. Typically, mbufs that are allocated from the mbuf pool (discussed later) have their `om_data` pointer set to the start of the data buffer but there are cases where this may not be desirable (added a protocol header to a packet, for example).
- The `om_flags` field is a set of flags used internally by the mbuf library. Currently, no flags have been defined.
- The `om_pkthdr_len` field is the total length of all packet headers in the mbuf. For normal mbufs this is set to 0 as there is no packet or user packet headers. For packet header mbufs, this would be set to the length of the packet header structure (16) plus the size of the user packet header (if any). Note that it is this field which differentiates packet header mbufs from normal mbufs (i.e. if `om_pkthdr_len` is zero, this is a normal mbuf; otherwise it is a packet header mbuf).
- The `om_len` field contains the amount of user data in the data buffer. When initially allocated, this field is 0 as there is no user data in the mbuf.
- The `omp_pool` field is a pointer to the pool from which this mbuf has been allocated. This is used internally by the mbuf library.
- The `omp_next` field is a linked list element which is used to chain mbufs.

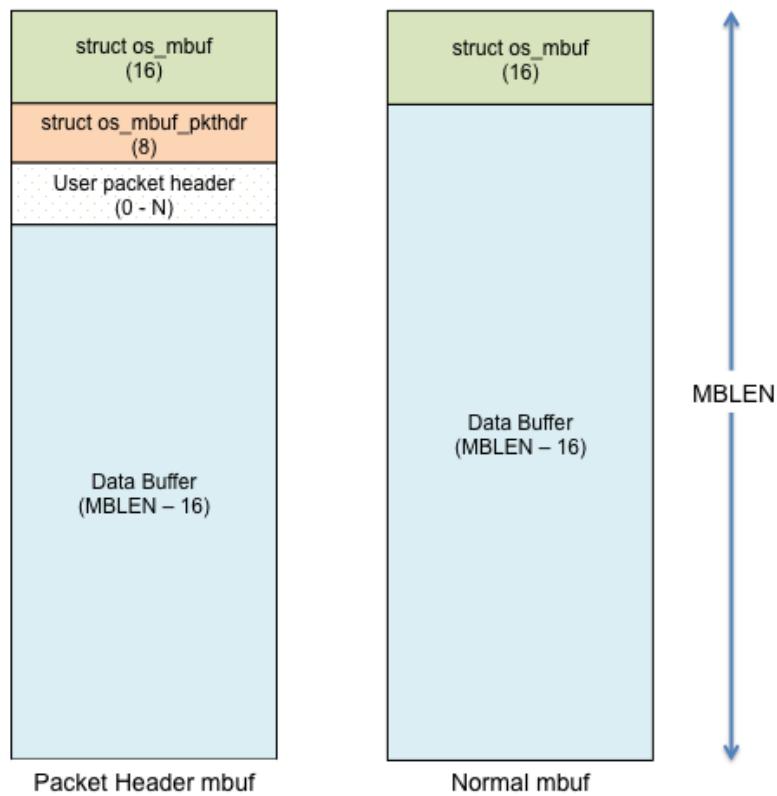


Figure 1 Packet header and normal mbufs

Fig. 1: Packet header mbuf

Figure 2 also shows a normal mbuf with actual values in the `os_mbuf` structure. This mbuf starts at address 0x1000 and is 256 bytes in total length. In this example, the user has copied 33 bytes into the data buffer starting at address 0x1010 (this is where `om_data` points). Note that the packet header length in this mbuf is 0 as it is not a packet header mbuf.

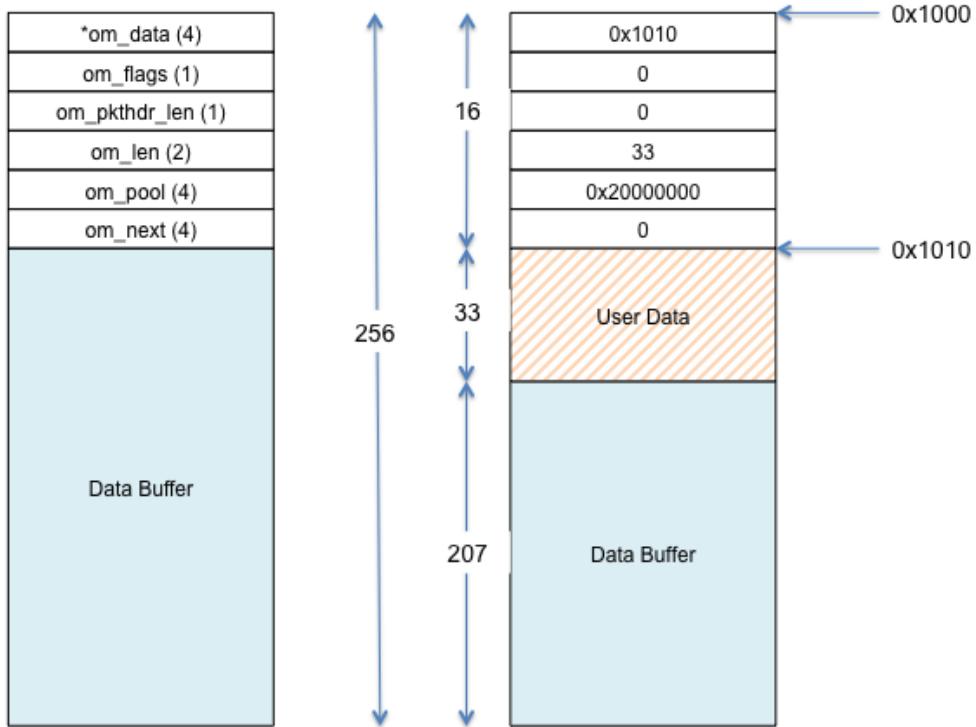


Figure 2 `os_mbuf` structure details

Fig. 2: OS mbuf structure

Figure 3 illustrates the packet header mbuf along with some chained mbus (i.e a “packet”). In this example, the user header structure is defined to be 8 bytes. Note that in figure 3 we show a number of different mbus with varying `om_data` pointers and lengths since we want to show various examples of valid mbus. For all the mbus (both packet header and normal ones) the total length of the memory block is 128 bytes.

Mbuf pools

Mbus are collected into “mbuf pools” much like memory blocks. The mbuf pool itself contains a pointer to a memory pool. The memory blocks in this memory pool are the actual mbus; both normal and packet header mbus. Thus, the memory block (and corresponding memory pool) must be sized correctly. In other words, the memory blocks which make up the memory pool used by the mbuf pool must be at least: `sizeof(struct os_mbuf) + sizeof(struct os_mbuf_pkthdr) + sizeof(struct user_defined_header) + desired minimum data buffer length`. For example, if the developer wants mbus to contain at least 64 bytes of user data and they have a user header of 12 bytes, the size of the memory block would be (at least): $64 + 12 + 16 + 8$, or 100 bytes. Yes, this is a fair amount of overhead. However, the flexibility provided by the mbuf library usually outweighs overhead concerns.

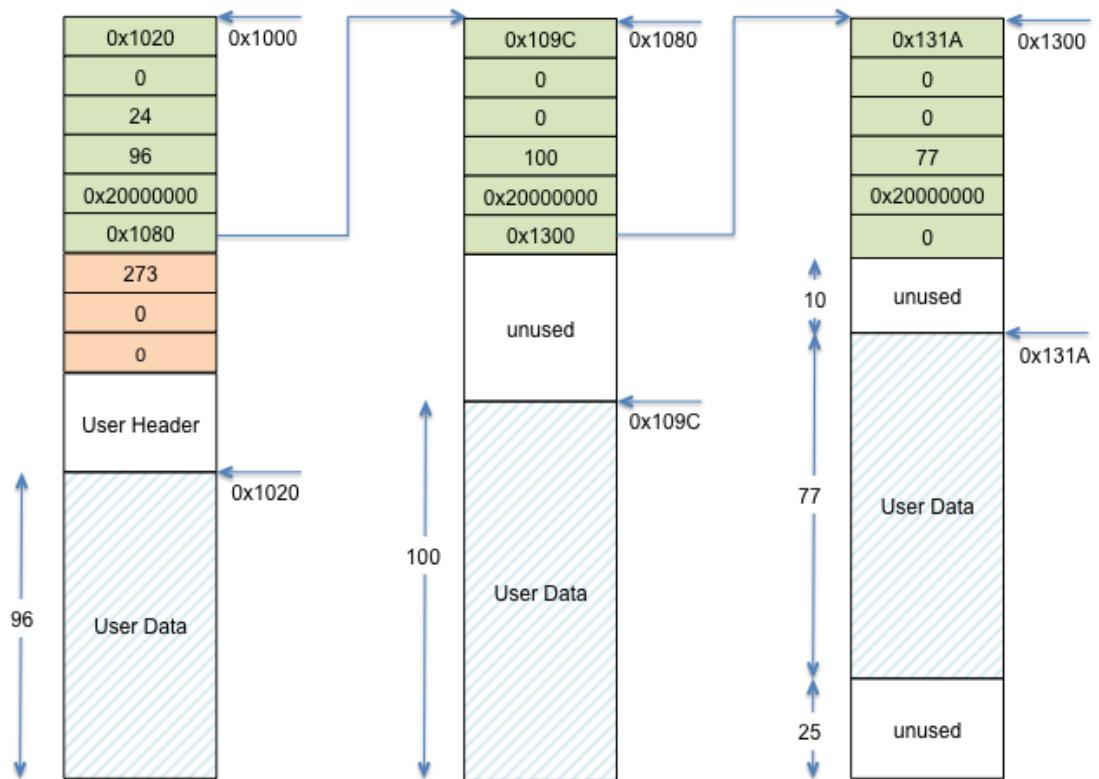


Figure 3 Packet example

Fig. 3: Packet

Create mbuf pool

Creating an mbuf pool is fairly simple: create a memory pool and then create the mbuf pool using that memory pool. Once the developer has determined the size of the user data needed per mbuf (this is based on the application/networking stack and is outside the scope of this discussion) and the size of the user header (if any), the memory blocks can be sized. In the example shown below, the application requires 64 bytes of user data per mbuf and also allocates a user header (called struct `user_hdr`). Note that we do not show the user header data structure as there really is no need; all we need to do is to account for it when creating the memory pool. In the example, we use the macro `MBUF_PKTHDR_OVERHEAD` to denote the amount of packet header overhead per mbuf and `MBUF_MEMBLOCK_OVERHEAD` to denote the total amount of overhead required per memory block. The macro `MBUF_BUF_SIZE` is used to denote the amount of payload that the application requires (aligned on a 32-bit boundary in this case). All this leads to the total memory block size required, denoted by the macro `MBUF_MEMBLOCK_SIZE`.

```
#define MBUF_PKTHDR_OVERHEAD    sizeof(struct os_mbuf_pkthdr) + sizeof(struct user_hdr)
#define MBUF_MEMBLOCK_OVERHEAD   sizeof(struct os_mbuf) + MBUF_PKTHDR_OVERHEAD

#define MBUF_NUM_MBUFS      (32)
#define MBUF_PAYLOAD_SIZE   (64)
#define MBUF_BUF_SIZE        OS_ALIGN(MBUF_PAYLOAD_SIZE, 4)
#define MBUF_MEMBLOCK_SIZE   (MBUF_BUF_SIZE + MBUF_MEMBLOCK_OVERHEAD)
#define MBUF_MEMPOOL_SIZE    OS_MEMPOOL_SIZE(MBUF_NUM_MBUFS, MBUF_MEMBLOCK_SIZE)

struct os_mbuf_pool g_mbuf_pool;
struct os_mempool g_mbuf_mempool;
os_membuf_t g_mbuf_buffer[MBUF_MEMPOOL_SIZE];

void
create_mbuf_pool(void)
{
    int rc;

    rc = os_mempool_init(&g_mbuf_mempool, MBUF_NUM_MBUFS,
                         MBUF_MEMBLOCK_SIZE, &g_mbuf_buffer[0], "mbuf_pool");
    assert(rc == 0);

    rc = os_mbuf_pool_init(&g_mbuf_pool, &g_mbuf_mempool, MBUF_MEMBLOCK_SIZE,
                           MBUF_NUM_MBUFS);
    assert(rc == 0);
}
```

Msys

Msys stands for “system mbufs” and is a set of API built on top of the mbuf code. The basic idea behind msys is the following. The developer can create different size mbuf pools and register them with msys. The application then allocates mbufs using the msys API (as opposed to the mbuf API). The msys code will choose the mbuf pool with the smallest mbufs that can accommodate the requested size.

Let us walk through an example where the user registers three mbuf pools with msys: one with 32 byte mbufs, one with 256 and one with 2048. If the user requests an mbuf with 10 bytes, the 32-byte mbuf pool is used. If the request is for 33 bytes the 256 byte mbuf pool is used. If an mbuf data size is requested that is larger than any of the pools (say, 4000 bytes) the largest pool is used. While this behaviour may not be optimal in all cases that is the currently implemented behaviour. All this means is that the user is not guaranteed that a single mbuf can hold the requested data.

The msys code will not allocate an mbuf from a larger pool if the chosen mbuf pool is empty. Similarly, the msys code will not chain together a number of smaller mbufs to accommodate the requested size. While this behaviour may change in future implementations the current code will simply return NULL. Using the above example, say the user requests 250 bytes. The msys code chooses the appropriate pool (i.e. the 256 byte mbuf pool) and attempts to allocate an mbuf from that pool. If that pool is empty, NULL is returned even though the 32 and 2048 byte pools are not empty.

Note that no added descriptions on how to use the msys API are presented here (other than in the API descriptions themselves) as the msys API is used in exactly the same manner as the mbuf API. The only difference is that mbuf pools are added to msys by calling `os_msyst_register()`.

Using mbufs

The following examples illustrate typical mbuf usage. There are two basic mbuf allocation API: `c:func:os_mbuf_get()` and `os_mbuf_get_pkthdr()`. The first API obtains a normal mbuf whereas the latter obtains a packet header mbuf. Typically, application developers use `os_mbuf_get_pkthdr()` and rarely, if ever, need to call `os_mbuf_get()` as the rest of the mbuf API (e.g. `os_mbuf_append()`, `os_mbuf_copyinto()`, etc.) typically deal with allocating and chaining mbufs. It is recommended to use the provided API to copy data into/out of mbuf chains and/or manipulate mbufs.

In `example1`, the developer creates a packet and then sends the packet to a networking interface. The code sample also provides an example of copying data out of an mbuf as well as use of the “pullup” api (another very common mbuf api).

```
void
mbuf_usage_example1(uint8_t *mydata, int mydata_length)
{
    int rc;
    struct os_mbuf *om;

    /* get a packet header mbuf */
    om = os_mbuf_get_pkthdr(&g_mbuf_pool, sizeof(struct user_hdr));
    if (om) {
        /*
         * Copy user data into mbuf. NOTE: if mydata_length is greater than the
         * mbuf payload size (64 bytes using above example), mbufs are allocated
         * and chained together to accommodate the total packet length.
         */
        rc = os_mbuf_copyinto(om, 0, mydata, len);
        if (rc) {
            /* Error! Could not allocate enough mbufs for total packet length */
            return -1;
        }

        /* Send packet to networking interface */
        send_pkt(om);
    }
}
```

In `example2` we show use of the pullup api as this illustrates some of the typical pitfalls developers encounter when using mbufs. The first pitfall is one of alignment/padding. Depending on the processor and/or compiler, the `sizeof()` a structure may vary. Thus, the size of `my_protocol_header` may be different inside the packet data of the mbuf than the size of the structure on the stack or as a global variable, for instance. While some networking protocols may align protocol information on convenient processor boundaries many others try to conserve bytes “on the air” (i.e inside the packet data). Typical methods used to deal with this are “packing” the structure (i.e. force compiler to not pad) or

creating protocol headers that do not require padding. `example2` assumes that one of these methods was used when defining the `my_protocol_header` structure.

Another common pitfall occurs around endianness. A network protocol may be little endian or big endian; it all depends on the protocol specification. Processors also have an endianness; this means that the developer has to be careful that the processor endianness and the protocol endianness are handled correctly. In `example2`, some common networking functions are used: `ntohs()` and `ntohl()`. These are shorthand for “network order to host order, short” and “network order to host order, long”. Basically, these functions convert data of a certain size (i.e. 16 bits, 32 bits, etc) to the endianness of the host. Network byte order is big-endian (most significant byte first), so these functions convert big-endian byte order to host order (thus, the implementation of these functions is host dependent). Note that the BLE networking stack “on the air” format is least significant byte first (i.e. little endian), so a “bletoh” function would have to take little endian format and convert to host format.

A long story short: the developer must take care when copying structure data to/from mbufs and flat buffers!

A final note: these examples assume the same mbuf structure and definitions used in the first example.

```
void
mbuf_usage_example2(struct mbuf *rxpkt)
{
    int rc;
    uint8_t packet_data[16];
    struct mbuf *om;
    struct my_protocol_header *phdr;

    /* Make sure that "my_protocol_header" bytes are contiguous in mbuf */
    om = os_mbuf_pullup(&g_mbuf_pool, sizeof(struct my_protocol_header));
    if (!om) {
        /* Not able to pull up data into contiguous area */
        return -1;
    }

    /*
     * Get the protocol information from the packet. In this example we presume that we
     * are interested in protocol types that are equal to MY_PROTOCOL_TYPE, are not zero
     * length, and have had some time in flight.
     */
    phdr = OS_MBUF_DATA(om, struct my_protocol_header *);
    type = ntohs(phdr->prot_type);
    length = ntohs(phdr->prot_length);
    time_in_flight = ntohl(phdr->prot_tif);

    if ((type == MY_PROTOCOL_TYPE) && (length > 0) && (time_in_flight > 0)) {
        rc = os_mbuf_copydata(rxpkt, sizeof(struct my_protocol_header), 16, packet_data);
        if (!rc) {
            /* Success! Perform operations on packet data */
            <... user code here ...>
        }
    }

    /* Free passed in packet (mbuf chain) since we don't need it anymore */
    os_mbuf_free_chain(om);
}
```

Mqueue

The mqueue construct allows a task to wake up when it receives data. Typically, this data is in the form of packets received over a network. A common networking stack operation is to put a packet on a queue and post an event to the task monitoring that queue. When the task handles the event, it processes each packet on the packet queue.

Using Mqueue

The following code sample demonstrates how to use an mqueue. In this example:

- packets are put on a receive queue
- a task processes each packet on the queue (increments a receive counter)

Not shown in the code example is a call `my_task_rx_data_func`. Presumably, some other code will call this API.

```
uint32_t pkts_rxd;
struct os_mqueue rxpkt_q;
struct os_eventq my_task_evq;

/***
 * Removes each packet from the receive queue and processes it.
 */
void
process_rx_data_queue(void)
{
    struct os_mbuf *om;

    while ((om = os_mqueue_get(&rxpkt_q)) != NULL) {
        ++pkts_rxd;
        os_mbuf_free_chain(om);
    }
}

/***
 * Called when a packet is received.
 */
int
my_task_rx_data_func(struct os_mbuf *om)
{
    int rc;

    /* Enqueue the received packet and wake up the listening task. */
    rc = os_mqueue_put(&rxpkt_q, &my_task_evq, om);
    if (rc != 0) {
        return -1;
    }

    return 0;
}

void
my_task_handler(void *arg)
{
```

(continues on next page)

(continued from previous page)

```

struct os_event *ev;
struct os_callout_func *cf;
int rc;

/* Initialize eventq */
os_eventq_init(&my_task_evq);

/* Initialize mqueue */
os_mqueue_init(&rxpkt_q, NULL);

/* Process each event posted to our eventq. When there are no events to
* process, sleep until one arrives.
*/
while (1) {
    os_eventq_run(&my_task_evq);
}
}

```

API

int os_mqueue_init(struct os_mqueue *mq, os_event_fn *ev_cb, void *arg)

Initializes an mqueue.

An mqueue is a queue of mbusfs that ties to a particular task's event queue. Mqueues form a helper API around a common paradigm: wait on an event queue until at least one packet is available, then process a queue of packets.

When mbusfs are available on the queue, an event OS_EVENT_T_MQUEUE_DATA will be posted to the task's mbuf queue.

Parameters

- **mq** – The mqueue to initialize
- **ev_cb** – The callback to associate with the mqueue event. Typically, this callback pulls each packet off the mqueue and processes them.
- **arg** – The argument to associate with the mqueue event.

Returns

0 on success, non-zero on failure.

struct os_mbuf *os_mqueue_get(struct os_mqueue *mq)

Remove and return a single mbuf from the mbuf queue.

Does not block.

Parameters

- **mq** – The mbuf queue to pull an element off of.

Returns

The next mbuf in the queue; NULL if queue has no mbusfs.

int os_mqueue_put(struct os_mqueue *mq, struct os_eventq *evq, struct os_mbuf *om)

Adds a packet (i.e.

packet header mbuf) to an mqueue. The event associated with the mqueue gets posted to the specified eventq.

Parameters

- **mq** – The mbuf queue to append the mbuf to.
- **evq** – The event queue to post an event to.
- **om** – The mbuf to append to the mbuf queue.

Returns

0 on success, non-zero on failure.

```
int os_msyst_register(struct os_mbuf_pool *new_pool)
```

MSYS is a system level mbuf registry.

Allows the system to share packet buffers amongst the various networking stacks that can be running simultaneously.

Mbuf pools are created in the system initialization code, and then when a mbuf is allocated out of msys, it will try and find the best fit based upon estimated mbuf size.

os_msyst_register() registers a mbuf pool with MSYS, and allows MSYS to allocate mbufs out of it.

Parameters

- **new_pool** – The pool to register with MSYS

Returns

0 on success; non-zero on failure

```
struct os_mbuf *os_msyst_get(uint16_t dsize, uint16_t leadingspace)
```

Allocate a mbuf from msys.

Based upon the data size requested, *os_msyst_get()* will choose the mbuf pool that has the best fit.

Parameters

- **dsize** – The estimated size of the data being stored in the mbuf
- **leadingspace** – The amount of leadingspace to allocate in the mbuf

Returns

A freshly allocated mbuf on success; NULL on failure.

```
void os_msyst_reset(void)
```

De-registers all mbuf pools from msys.

```
struct os_mbuf *os_msyst_get_pkthdr(uint16_t dsize, uint16_t user_hdr_len)
```

Allocate a packet header structure from the MSYS pool.

See *os_msyst_register()* for a description of MSYS.

Parameters

- **dsize** – The estimated size of the data being stored in the mbuf
- **user_hdr_len** – The length to allocate for the packet header structure

Returns

A freshly allocated mbuf on success; NULL on failure.

```
int os_msyst_count(void)
```

Count the number of blocks in all the mbuf pools that are allocated.

Returns

total number of blocks allocated in Msys

```
int os_msys_num_free(void)
```

Return the number of free blocks in Msys.

Returns

Number of free blocks available in Msys

```
int os_mbuf_pool_init(struct os_mbuf_pool *omp, struct os_mempool *mp, uint16_t buf_len, uint16_t nbufs)
```

Initialize a pool of mbufs.

Parameters

- **omp** – The mbuf pool to initialize
- **mp** – The memory pool that will hold this mbuf pool
- **buf_len** – The length of the buffer itself.
- **nbufs** – The number of buffers in the pool

Returns

0 on success; error code on failure.

```
struct os_mbuf *os_mbuf_get(struct os_mbuf_pool *omp, uint16_t leadingspace)
```

Get an mbuf from the mbuf pool.

The mbuf is allocated, and initialized prior to being returned.

Parameters

- **omp** – The mbuf pool to return the packet from
- **leadingspace** – The amount of leadingspace to put before the data section by default.

Returns

An initialized mbuf on success; NULL on failure.

```
struct os_mbuf *os_mbuf_get_pkthdr(struct os_mbuf_pool *omp, uint8_t user_pkthdr_len)
```

Allocate a new packet header mbuf out of the *os_mbuf_pool*.

Parameters

- **omp** – The mbuf pool to allocate out of
- **user_pkthdr_len** – The packet header length to reserve for the caller.

Returns

A freshly allocated mbuf on success; NULL on failure.

```
struct os_mbuf *os_mbuf_dup(struct os_mbuf *om)
```

Duplicate a chain of mbufs.

Return the start of the duplicated chain.

Parameters

- **om** – The mbuf chain to duplicate

Returns

A pointer to the new chain of mbufs

```
struct os_mbuf *os_mbuf_off(const struct os_mbuf *om, int off, uint16_t *out_off)
```

Locates the specified absolute offset within an mbuf chain.

The offset can be one past than the total length of the chain, but no greater.

Parameters

- **om** – The start of the mbuf chain to seek within.
- **off** – The absolute address to find.
- **out_off** – On success, this points to the relative offset within the returned mbuf.

Returns

The mbuf containing the specified offset on success. NULL if the specified offset is out of bounds.

```
int os_mbuf_copydata(const struct os_mbuf *om, int off, int len, void *dst)
```

Copy data from an mbuf chain starting “off” bytes from the beginning, continuing for “len” bytes, into the indicated buffer.

Parameters

- **om** – The mbuf chain to copy from
- **off** – The offset into the mbuf chain to begin copying from
- **len** – The length of the data to copy
- **dst** – The destination buffer to copy into

Returns

0 on success; -1 if the mbuf does not contain enough data.

```
uint16_t os_mbuf_len(const struct os_mbuf *om)
```

Calculates the length of an mbuf chain.

Calculates the length of an mbuf chain. If the mbuf contains a packet header, you should use [OS_MBUF_PKTLEN\(\)](#) as a more efficient alternative to this function.

Parameters

- **om** – The mbuf to measure.

Returns

The length, in bytes, of the provided mbuf chain.

```
int os_mbuf_append(struct os_mbuf *om, const void *data, uint16_t len)
```

Append data onto a mbuf.

Parameters

- **om** – The mbuf to append the data onto
- **data** – The data to append onto the mbuf
- **len** – The length of the data to append

Returns

0 on success; an error code on failure

```
int os_mbuf_appendfrom(struct os_mbuf *dst, const struct os_mbuf *src, uint16_t src_off, uint16_t len)
```

Reads data from one mbuf and appends it to another.

On error, the specified data range may be partially appended. Neither mbuf is required to contain an mbuf packet header.

Parameters

- **dst** – The mbuf to append to.
- **src** – The mbuf to copy data from.
- **src_off** – The absolute offset within the source mbuf chain to read from.

- **len** – The number of bytes to append.

Returns

0 on success; OS_EINVAL if the specified range extends beyond the end of the source mbuf chain.

int os_mbuf_free(struct *os_mbuf* *om)

Release a mbuf back to the pool.

Parameters

- **om** – The Mbuf to release back to the pool

Returns

0 on success; -1 on failure

int os_mbuf_free_chain(struct *os_mbuf* *om)

Free a chain of mbus.

Parameters

- **om** – The starting mbuf of the chain to free back into the pool

Returns

0 on success; -1 on failure

void os_mbuf_adj(struct *os_mbuf* *om, int req_len)

Adjust the length of a mbuf, trimming either from the head or the tail of the mbuf.

Parameters

- **om** – The mbuf chain to adjust
- **req_len** – The length to trim from the mbuf. If positive, trims from the head of the mbuf, if negative, trims from the tail of the mbuf.

int os_mbuf_cmpf(const struct *os_mbuf* *om, int off, const void *data, int len)

Performs a memory compare of the specified region of an mbuf chain against a flat buffer.

Parameters

- **om** – The start of the mbuf chain to compare.
- **off** – The offset within the mbuf chain to start the comparison.
- **data** – The flat buffer to compare.
- **len** – The length of the flat buffer.

Returns

0 if both memory regions are identical; A memcmp return code if there is a mismatch; INT_MAX if the mbuf is too short.

int os_mbuf_cmpm(const struct *os_mbuf* *om1, uint16_t offset1, const struct *os_mbuf* *om2, uint16_t offset2, uint16_t len)

Compares the contents of two mbuf chains.

The ranges of the two chains to be compared are specified via the two offset parameters and the len parameter. Neither mbuf chain is required to contain a packet header.

Parameters

- **om1** – The first mbuf chain to compare.
- **offset1** – The absolute offset within om1 at which to start the comparison.

- **om2** – The second mbuf chain to compare.
- **offset2** – The absolute offset within om2 at which to start the comparison.
- **len** – The number of bytes to compare.

Returns

0 if both mbuf segments are identical; A memcmp() return code if the segment contents differ; INT_MAX if a specified range extends beyond the end of its corresponding mbuf chain.

```
struct os_mbuf *os_mbuf_prepend(struct os_mbuf *om, int len)
```

Increases the length of an mbuf chain by adding data to the front.

If there is insufficient room in the leading mbuf, additional mbufs are allocated and prepended as necessary. If this function fails to allocate an mbuf, the entire chain is freed.

The specified mbuf chain does not need to contain a packet header.

Parameters

- **om** – The head of the mbuf chain.
- **len** – The number of bytes to prepend.

Returns

The new head of the chain on success; NULL on failure.

```
struct os_mbuf *os_mbuf_prepend_pullup(struct os_mbuf *om, uint16_t len)
```

Prepends a chunk of empty data to the specified mbuf chain and ensures the chunk is contiguous.

If either operation fails, the specified mbuf chain is freed and NULL is returned.

Parameters

- **om** – The mbuf chain to prepend to.
- **len** – The number of bytes to prepend and pullup.

Returns

The modified mbuf on success; NULL on failure (and the mbuf chain is freed).

```
int os_mbuf_copyinto(struct os_mbuf *om, int off, const void *src, int len)
```

Copies the contents of a flat buffer into an mbuf chain, starting at the specified destination offset.

If the mbuf is too small for the source data, it is extended as necessary. If the destination mbuf contains a packet header, the header length is updated.

Parameters

- **om** – The mbuf chain to copy into.
- **off** – The offset within the chain to copy to.
- **src** – The source buffer to copy from.
- **len** – The number of bytes to copy.

Returns

0 on success; nonzero on failure.

```
void os_mbuf_concat(struct os_mbuf *first, struct os_mbuf *second)
```

Attaches a second mbuf chain onto the end of the first.

If the first chain contains a packet header, the header's length is updated. If the second chain has a packet header, its header is cleared.

Parameters

- **first** – The mbuf chain being attached to.
- **second** – The mbuf chain that gets attached.

`void *os_mbuf_extend(struct os_mbuf *om, uint16_t len)`

Increases the length of an mbuf chain by the specified amount.

If there is not sufficient room in the last buffer, a new buffer is allocated and appended to the chain. It is an error to request more data than can fit in a single buffer.

Parameters

- **om** – The head of the chain to extend.
- **len** – The number of bytes to extend by.

Returns

A pointer to the new data on success; NULL on failure.

`struct os_mbuf *os_mbuf_pullup(struct os_mbuf *om, uint16_t len)`

Rearrange a mbuf chain so that len bytes are contiguous, and in the data area of an mbuf (so that `OS_MBUF_DATA()` will work on a structure of size len.) Returns the resulting mbuf chain on success, free's it and returns NULL on failure.

If there is room, it will add up to “max_protohdr - len” extra bytes to the contiguous region, in an attempt to avoid being called next time.

Parameters

- **om** – The mbuf chain to make contiguous
- **len** – The number of bytes in the chain to make contiguous

Returns

The contiguous mbuf chain on success; NULL on failure.

`struct os_mbuf *os_mbuf_trim_front(struct os_mbuf *om)`

Removes and frees empty mbus from the front of a chain.

If the chain contains a packet header, it is preserved.

Parameters

- **om** – The mbuf chain to trim.

Returns

The head of the trimmed mbuf chain.

`int os_mbuf_widen(struct os_mbuf *om, uint16_t off, uint16_t len)`

Increases the length of an mbuf chain by inserting a gap at the specified offset.

The contents of the gap are indeterminate. If the mbuf chain contains a packet header, its total length is increased accordingly.

This function never frees the provided mbuf chain.

Parameters

- **om** – The mbuf chain to widen.
- **off** – The offset at which to insert the gap.
- **len** – The size of the gap to insert.

Returns

0 on success; SYS_[...] error code on failure.

struct *os_mbuf* ***os_mbuf_pack_chains**(struct *os_mbuf* *m1, struct *os_mbuf* *m2)

Creates a single chained mbuf from m1 and m2 utilizing all the available buffer space in all mbufs in the resulting chain.

In other words, ensures there is no leading space in any mbuf in the resulting chain and trailing space only in the last mbuf in the chain. Mbufs from either chain may be freed if not needed. No mbufs are allocated. Note that mbufs from m2 are added to the end of m1. If m1 has a packet header, it is retained and length updated. If m2 has a packet header it is discarded. If m1 is NULL, NULL is returned and m2 is left untouched.

Parameters

- **m1** – Pointer to first mbuf chain to pack
- **m2** – Pointer to second mbuf chain to pack

Returns

Pointer to resulting mbuf chain

OS_MBUF_F_MASK(*__n***)**

Given a flag number, provide the mask for it.

Parameters

- **__n** – The number of the flag in the mask

OS_MBUF_IS_PKTHDR(*__om***)**

Checks whether a given mbuf is a packet header mbuf.

Parameters

- **__om** – The mbuf to check

OS_MBUF_PKTHDR(*__om***)**

Get a packet header pointer given an mbuf pointer.

OS_MBUF_PKTHDR_TO_MBUF(*__hdr***)**

Given a mbuf packet header pointer, return a pointer to the mbuf.

OS_MBUF_PKTLEN(*__om***)**

Gets the length of an entire mbuf chain.

The specified mbuf must have a packet header.

OS_MBUF_DATA(*__om*, *__type***)**

Access the data of a mbuf, and cast it to type.

Parameters

- **__om** – The mbuf to access, and cast
- **__type** – The type to cast it to

OS_MBUF_USRHDR(*om***)**

Access the “user header” in the head of an mbuf chain.

Parameters

- **om** – Pointer to the head of an mbuf chain.

OS_MBUF_USRHDR_LEN(om)

Retrieves the length of the user header in an mbuf.

Parameters

- **om** – Pointer to the mbuf to query.

OS_MBUF.LEADINGSPACE(__om)

Returns the leading space (space at the beginning) of the mbuf.

Works on both packet header, and regular mbufs, as it accounts for the additional space allocated to the packet header.

Parameters

- **__om** – The mbuf in that pool to get the leading space for

Returns

Amount of leading space available in the mbuf

OS_MBUF.TRAILINGSPACE(__om)

Returns the trailing space (space at the end) of the mbuf.

Works on both packet header and regular mbufs.

Parameters

- **__om** – The mbuf in that pool to get the trailing space for

Returns

Amount of trailing space available in the mbuf

struct os_mbuf_pool

#include <os_mbuf.h> A mbuf pool from which to allocate mbufs.

This contains a pointer to the os mempool to allocate mbufs out of, the total number of elements in the pool, and the amount of “user” data in a non-packet header mbuf. The total pool size, in bytes, should be: `os_mbuf_count * (omp_databuf_len + sizeof(struct os_mbuf))`

Public Functions**STAILQ_ENTRY (os_mbuf_pool) omp_next**

Next mbuf pool in the list.

Public Members**uint16_t omp_databuf_len**

Total length of the databuf in each mbuf.

This is the size of the mempool block, minus the mbuf header

struct os_mempool *omp_pool

The memory pool which to allocate mbufs out of.

struct os_mbuf_pkthdr

#include <os_mbuf.h> A packet header structure that precedes the mbuf packet headers.

Public Functions

STAILQ_ENTRY (os_mbuf_pkthdr) omp_next

Next mbuf packet header in the list.

Public Members

uint16_t omp_len

Overall length of the packet.

uint16_t omp_flags

Flags.

struct os_mbuf

#include <os_mbuf.h> Chained memory buffer.

Public Functions

SLIST_ENTRY (os_mbuf) om_next

Next mbuf in the list.

Public Members

uint8_t *om_data

Current pointer to data in the structure.

uint8_t om_flags

Flags associated with this buffer, see OS_MBUF_F_* definitions.

uint8_t om_pkthdr_len

Length of packet header.

uint16_t om_len

Length of data in this buffer.

struct os_mbuf_pool *om_omp

The mbuf pool this mbuf was allocated out of.

uint8_t om_databuf[0]

Pointer to the beginning of the data, after this buffer.

struct os_mqueue

#include <os_mbuf.h> Structure representing a queue of mbus.

Public Functions

`STAILQ_HEAD (, os_mbuf_pkthdr) mq_head`

A queue of mbuf packet headers.

Public Members

struct `os_event` `mq_ev`

Event to post when new buffers are available on the queue.

6.1.11 CPU Time

The MyNewt cputime module provides high resolution time and timer support.

Description

The cputime API provides high resolution time and timer support. The module must be initialized, using the `os_cputime_init()` function, with the clock frequency to use. The module uses the hal_timer API, defined in `hal/hal_timer.h`, to access the hardware timers. It uses the hardware timer number specified by the `OS_CPUTIME_TIMER_NUM` system configuration setting.

API

`int os_cputime_init(uint32_t clock_freq)`

Initialize the cputime module.

This must be called after `os_init` is called and before any other timer API are used. This should be called only once and should be called before the hardware timer is used.

Parameters

- `clock_freq` – The desired cputime frequency, in hertz (Hz).

Returns

0 on success; -1 on error.

`uint32_t os_cputime_get32(void)`

Returns the low 32 bits of cputime.

Returns

The lower 32 bits of cputime

`uint32_t os_cputime_nsecs_to_ticks(uint32_t nsecs)`

Converts the given number of nanoseconds into cputime ticks.

Not defined if `OS_CPUTIME_FREQ_PWR2` is defined.

Parameters

- `nsecs` – The number of nanoseconds to convert to ticks

Returns

The number of ticks corresponding to ‘nsecs’

`uint32_t os_cputime_ticks_to_nsecs(uint32_t ticks)`

Convert the given number of ticks into nanoseconds.

Not defined if OS_CPUTIME_FREQ_PWR2 is defined.

Parameters

- **ticks** – The number of ticks to convert to nanoseconds.

Returns

The number of nanoseconds corresponding to ‘ticks’

`void os_cputime_delay_nsecs(uint32_t nsecs)`

Wait until ‘nsecs’ nanoseconds has elapsed.

This is a blocking delay. Not defined if OS_CPUTIME_FREQ_PWR2 is defined.

Parameters

- **nsecs** – The number of nanoseconds to wait.

`uint32_t os_cputime_usecs_to_ticks(uint32_t usecs)`

Converts the given number of microseconds into cputime ticks.

Parameters

- **usecs** – The number of microseconds to convert to ticks

Returns

The number of ticks corresponding to ‘usecs’

`uint32_t os_cputime_ticks_to_usecs(uint32_t ticks)`

Convert the given number of ticks into microseconds.

Parameters

- **ticks** – The number of ticks to convert to microseconds.

Returns

The number of microseconds corresponding to ‘ticks’

`void os_cputime_delay_ticks(uint32_t ticks)`

Wait until the number of ticks has elapsed.

This is a blocking delay.

Parameters

- **ticks** – The number of ticks to wait.

`void os_cputime_delay_usecs(uint32_t usecs)`

Wait until ‘usecs’ microseconds has elapsed.

This is a blocking delay.

Parameters

- **usecs** – The number of usecs to wait.

`void os_cputime_timer_init(struct hal_timer *timer, hal_timer_cb fp, void *arg)`

Initialize a CPU timer, using the given HAL timer.

Parameters

- **timer** – The timer to initialize. Cannot be NULL.

- **fp** – The timer callback function. Cannot be NULL.
- **arg** – Pointer to data object to pass to timer.

```
int os_cputime_timer_start(struct hal_timer *timer, uint32_t cputime)
```

Start a cputimer that will expire at ‘cputime’.

If cputime has already passed, the timer callback will still be called (at interrupt context).

 **Note**

This must be called when the timer is stopped.

Parameters

- **timer** – Pointer to timer to start. Cannot be NULL.
- **cputime** – The cputime at which the timer should expire.

Returns

0 on success; EINVAL if timer already started or timer struct invalid

```
int os_cputime_timer_relative(struct hal_timer *timer, uint32_t usecs)
```

Sets a cpu timer that will expire ‘usecs’ microseconds from the current cputime.

 **Note**

This must be called when the timer is stopped.

Parameters

- **timer** – Pointer to timer. Cannot be NULL.
- **usecs** – The number of usecs from now at which the timer will expire.

Returns

0 on success; EINVAL if timer already started or timer struct invalid

```
void os_cputime_timer_stop(struct hal_timer *timer)
```

Stops a cputimer from running.

The timer is removed from the timer queue and interrupts are disabled if no timers are left on the queue. Can be called even if timer is not running.

Parameters

- **timer** – Pointer to cputimer to stop. Cannot be NULL.

6.1.12 OS Time

The system time for the Mynewt OS.

- *Description*
- *Data Structures*
- *Functions*
- *Macros*
- *Special Notes*
- *API*

Description

The Mynewt OS contains an incrementing time that drives the OS scheduler and time delays. The time is a fixed size (e.g. 32 bits) and will eventually wrap back to zero. The time to wrap from zero back to zero is called the **OS time epoch**.

The frequency of the OS time tick is specified in the architecture-specific OS code `os_arch.h` and is named `OS_TICKS_PER_SEC`.

The Mynewt OS also provides APIs for setting and retrieving the wallclock time (also known as local time or time-of-day in other operating systems).

Data Structures

Time is stored in Mynewt as an `os_time_t` value.

Wallclock time is represented using the `struct os_timeval` and `struct os_timezone` tuple.

`struct os_timeval` represents the number of seconds elapsed since 00:00:00 Jan 1, 1970 UTC.

```
struct os_timeval {  
    int64_t tv_sec; /* seconds since Jan 1 1970 UTC */  
    int32_t tv_usec; /* fractional seconds */  
};  
  
struct os_timeval tv = { 1457400000, 0 }; /* 01:20:00 Mar 8 2016 UTC */
```

`struct os_timezone` is used to specify the offset of local time from UTC and whether daylight savings is in effect. Note that `tz_minuteswest` is a positive number if the local time is *behind* UTC and a negative number if the local time is *ahead* of UTC.

```
struct os_timezone {  
    int16_t tz_minuteswest;  
    int16_t tz_dsttime;  
};  
  
/* Pacific Standard Time is 08:00 hours west of UTC */  
struct os_timezone PST = { 480, 0 };
```

(continues on next page)

(continued from previous page)

```
struct os_timezone PDT = { 480, 1 };

/* Indian Standard Time is 05:30 hours east of UTC */
struct os_timezone IST = { -330, 0 };
```

Functions

Function	Description
<code>os_time_advance()</code>	Increments the OS time tick for the system.
<code>os_time_delay()</code>	Put the current task to sleep for the given number of ticks.
<code>os_time_get()</code>	Get the current value of OS time.
<code>os_time_ms_to_ticks()</code>	Converts milliseconds to os ticks.
<code>os_get_uptime_usec()</code>	Gets the time duration since boot.
<code>os_gettimeofday()</code>	Populate the given timeval and timezone structs with current time data.
<code>os_settimeofday()</code>	Set the current time of day to the given time structs.

Macros

Several macros help with the evalution of times with respect to each other.

- `OS_TIME_TICK_LT` – evaluates to true if t1 is before t2 in time.
- `OS_TIME_TICK_GT` – evaluates to true if t1 is after t2 in time
- `OS_TIME_TICK_GEQ` – evaluates to true if t1 is on or after t2 in time.

NOTE: For all of these macros the calculations are done modulo ‘`os_time_t`’.

Ensure that comparison of OS time always uses the macros above (to compensate for the possible wrap of OS time).

The following macros help adding or subtracting time when represented as `struct os_timeval`. All parameters to the following macros are pointers to `struct os_timeval`.

- `os_timeradd` – Add uvp to tvp and store result in vvp.
- `os_timersub` – Subtract uvp from tvp and store result in vvp.

Special Notes

Its important to understand how quickly the time wraps especially when doing time comparison using the macros above (or by any other means).

For example, if a tick is 1 millisecond and `os_time_t` is 32-bits the OS time will wrap back to zero in about 49.7 days or stated another way, the OS time epoch is 49.7 days.

If two times are more than 1/2 the OS time epoch apart, any time comparison will be incorrect. Ensure at design time that comparisons will not occur between times that are more than half the OS time epoch.

API

```
typedef int32_t os_stime_t
```

Signed 32-bit system time type definition.

```
typedef uint32_t os_time_t
```

Unsigned 32-bit system time type definition.

```
typedef void os_time_change_fn(const struct os_time_change_info *info, void *arg)
```

Callback that is executed when the time-of-day is set.

Param info

Describes the time change that just occurred.

Param arg

Optional argument corresponding to listener.

```
os_time_t os_time_get(void)
```

Get the current OS time in ticks.

Returns

OS time in ticks

```
void os_time_advance(int ticks)
```

Move OS time forward ticks.

Parameters

- **ticks** – The number of ticks to move time forward.

```
void os_time_delay(os_time_t osticks)
```

Puts the current task to sleep for the specified number of os ticks.

There is no delay if ticks is 0.

Parameters

- **osticks** – Number of ticks to delay (0 means no delay).

```
int os_settimeofday(struct os_timeval *utctime, struct os_timezone *tz)
```

Set the time of day.

This does not modify os time, but rather just modifies the offset by which we are tracking real time against os time. This function notifies all registered time change listeners.

Parameters

- **utctime** – A timeval representing the UTC time we are setting
- **tz** – The time-zone to apply against the utctime being set.

Returns

0 on success; non-zero on failure.

```
int os_gettimeofday(struct os_timeval *utctime, struct os_timezone *tz)
```

Get the current time of day.

Returns the time of day in UTC into the utctime argument, and returns the timezone (if set) into tz.

Parameters

- **utctime** – The structure to put the UTC time of day into

- **tz** – The structure to put the timezone information into

Returns

0 on success, non-zero on failure

bool os_time_is_set(void)

Indicates whether the time has been set.

Returns

True if time is set; false otherwise.

int64_t os_get_uptime_usec(void)

Get time since boot in microseconds.

Returns

Time since boot in microseconds

void os_get_uptime(struct *os_timeval* *tvp)

Get time since boot as *os_timeval*.

Parameters

- **tvp** – Structure to put the time since boot.

int os_time_ms_to_ticks(uint32_t ms, *os_time_t* *out_ticks)

Converts milliseconds to OS ticks.

Parameters

- **ms** – The milliseconds input.
- **out_ticks** – The OS ticks output.

Returns

0 on success; OS_EINVAL if the result is too large to fit in a uint32_t.

int os_time_ticks_to_ms(*os_time_t* ticks, uint32_t *out_ms)

Converts OS ticks to milliseconds.

Parameters

- **ticks** – The OS ticks input.
- **out_ms** – The milliseconds output.

Returns

0 on success; OS_EINVAL if the result is too large to fit in a uint32_t.

static inline *os_time_t* os_time_ms_to_ticks32(uint32_t ms)

Converts milliseconds to OS ticks.

This function does not check if conversion overflows and should be only used in cases where input is known to be small enough not to overflow.

Parameters

- **ms** – The milliseconds input.

Returns

The number of OS ticks.

static inline uint32_t **os_time_ticks_to_ms32**(*os_time_t* ticks)

Converts OS ticks to milliseconds.

This function does not check if conversion overflows and should be only used in cases where input is known to be small enough not to overflow.

Parameters

- **ticks** – The OS ticks input.

Returns

The number of milliseconds.

void **os_time_change_listen**(struct *os_time_change_listener* *listener)

Registers a time change listener.

Whenever the time is set, all registered listeners are notified. The provided pointer is added to an internal list, so the listener's lifetime must extend indefinitely (or until the listener is removed).

Note

This function is not thread safe. The following operations must be kept exclusive: o Addition of listener o Removal of listener o Setting time

Parameters

- **listener** – The listener to register.

int **os_time_change_remove**(const struct *os_time_change_listener* *listener)

Unregisters a time change listener.

Note

This function is not thread safe. The following operations must be kept exclusive: o Addition of listener o Removal of listener o Setting time

Parameters

- **listener** – The listener to unregister.

Returns

0 on success; non-zero error code on failure

INT32_MAX

Maximum value of 32-bit signed integer.

OS_TICKS_PER_SEC

OS_TIME_MAX

Maximum value for *os_time_t*.

OS_STIME_MAX

Maximum value for *os_stime_t*.

OS_TIMEOUT_NEVER

Used to wait forever for events and mutexes.

os_timeradd(tvp, uvp, vvp)

Add first two timeval arguments and place results in third timeval argument.

os_timersub(tvp, uvp, vvp)

Subtract first two timeval arguments and place results in third timeval argument.

struct os_timeval

#include <os_time.h> Structure representing time since Jan 1 1970 with microsecond granularity.

Public Members**int64_t tv_sec**

Seconds.

int32_t tv_usec

Microseconds within the second.

struct os_timezone

#include <os_time.h> Structure representing a timezone offset.

Public Members**int16_t tz_minuteswest**

Minutes west of GMT.

int16_t tz_dsttime

Daylight savings time correction (if any)

struct os_time_change_info

#include <os_time.h> Represents a time change.

Passed to time change listeners when the current time-of-day is set.

Public Members**const struct os_timeval *tci_prev_tv**

UTC time prior to change.

const struct os_timezone *tci_prev_tz

Time zone prior to change.

```
const struct os_timeval *tci_cur_tv
    UTC time after change.

const struct os_timezone *tci_cur_tz
    Time zone after change.

bool tci_newly_synced
    True if the time was not set prior to change.
```

```
struct os_time_change_listener
#include <os_time.h> Time change listener.

Notified when the time-of-day is set.
```

Public Functions

```
STAILQ_ENTRY (os_time_change_listener) tcl_next
    Next listener in the list.
```

Public Members

```
os_time_change_fn *tcl_fn
    Callback invoked when the time-of-day is set.

void *tcl_arg
    Argument to be passed to the callback function.
```

6.1.13 Sanity

The **Sanity** task is a software watchdog task, which runs periodically to check system state, and ensure that everything is still operating properly.

In a typical system design, there are multiple stages of watchdog:

- Internal Watchdog
- External Watchdog
- Sanity Watchdog

The *Internal Watchdog* is typically an MCU watchdog, which is tickled in the core of the OS. The internal watchdog is tickled frequently, and is meant to be an indicator the OS is running.

The *External Watchdog* is a watchdog that's typically run slower. The purpose of an external watchdog is to provide the system with a hard reset when it has lost its mind.

The *Sanity Watchdog* is the least frequently run watchdog, and is meant as an application watchdog.

This document is about the operation of the Mynewt Sanity Watchdog.

Description

Sanity Task

Mynewt OS uses the OS Idle task to check sanity. The SANITY_INTERVAL syscfg setting specifies the interval in seconds to perform the sanity checks.

By default, every operating system task provides the frequency it will check in with the sanity task, with the `sanity_itvl` parameter in the `c:func:os_task_init()` function:

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

`c:var:sanity_itvl` is the time in OS time ticks that the task being created must register in with the sanity task.

Checking in with Sanity Task

The task must then register in with the sanity task every `sanity_itvl` seconds. In order to do that, the task should call the `os_sanity_task_checkin()` function, which will reset the sanity check associated with this task. Here is an example of a task that uses a callout to checkin with the sanity task every 50 seconds:

```
#define TASK1_SANITY_CHECKIN_ITVL (50 * OS_TICKS_PER_SEC)
struct os_eventq task1_evq;

static void
task1(void *arg)
{
    struct os_task *t;
    struct os_event *ev;
    struct os_callout c;

    /* Get current OS task */
    t = os_sched_get_current_task();

    /* Initialize the event queue. */
    os_eventq_init(&task1_evq);

    /* Initialize the callout */
    os_callout_init(&c, &task1_evq, NULL);

    /* reset the callout to checkin with the sanity task
     * in 50 seconds to kick off timing.
     */
    os_callout_reset(&c, TASK1_SANITY_CHECKIN_ITVL);

    while (1) {
        ev = os_eventq_get(&task1_evq);

        /* The sanity timer has reset */
        if (ev->ev_arg == &c) {
            os_sanity_task_checkin(t);
        } else {
    }
```

(continues on next page)

(continued from previous page)

```

        /* not expecting any other events */
        assert(0);
    }

    /* Should never reach */
    assert(0);
}

```

Registering a Custom Sanity Check

If a particular task wants to further hook into the sanity framework to perform other checks during the sanity task's operation, it can do so by registering a `struct os_sanity_check` using the `os_sanity_check_register()` function.

```

static int
mymodule_perform_sanity_check(struct os_sanity_check *sc, void *arg)
{
    /* Perform your checking here. In this case, we check if there
     * are available buffers in mymodule, and return 0 (all good)
     * if true, and -1 (error) if not.
    */
    if (mymodule_has_buffers()) {
        return (0);
    } else {
        return (-1);
    }
}

static int
mymodule_register_sanity_check(void)
{
    struct os_sanity_check sc;

    os_sanity_check_init(&sc);
    /* Only assert() if mymodule_perform_sanity_check() fails 50
     * times. SANITY_TASK_INTERVAL is defined by the user, and
     * is the frequency at which the sanity_task runs in seconds.
    */
    OS_SANITY_CHECK_SETFUNC(&sc, mymodule_perform_sanity_check, NULL,
                           50 * SANITY_TASK_INTERVAL);

    rc = os_sanity_check_register(&sc);
    if (rc != 0) {
        goto err;
    }

    return (0);
err:
    return (rc);
}

```

In the above example, every time the custom sanity check `mymodule_perform_sanity_check` returns successfully (0), the sanity check is reset. In the `c:macro:OS_SANITY_CHECK_SETFUNC` macro, the sanity checkin interval is specified as `50 * SANITY_TASK_INTERVAL` (which is the interval at which the sanity task runs.) This means that the `mymodule_perform_sanity_check()` function needs to fail 50 times consecutively before the sanity task will crash the system.

TIP: When checking things like memory buffers, which can be temporarily be exhausted, it's a good idea to have the sanity check fail multiple consecutive times before crashing the system. This will avoid crashing for temporary failures.

API

`typedef int (*os_sanity_check_func_t)(struct os_sanity_check *osc, void *arg)`

Sanity check callback function.

Param osc

Pointer to the sanity check structure.

Param arg

Argument passed to the callback.

Return

0 on success; non-zero error code on failure

`int os_sanity_task_checkin(struct os_task *t)`

Provide a “task checkin” for the sanity task.

Parameters

- `t` – The task to check in

Returns

0 on success; non-zero error code on failure

`int os_sanity_check_init(struct os_sanity_check *sc)`

Initialize a sanity check.

Parameters

- `sc` – The sanity check to initialize

Returns

0 on success; non-zero error code on failure

`int os_sanity_check_register(struct os_sanity_check *sc)`

Register a sanity check.

Parameters

- `sc` – The sanity check to register

Returns

0 on success; non-zero error code on failure

`int os_sanity_check_reset(struct os_sanity_check *sc)`

Reset the os sanity check, so that it doesn't trip up the sanity timer.

Parameters

- `sc` – The sanity check to reset

Returns

0 on success; non-zero error code on failure

OS_SANITY_CHECK_SETFUNC(__sc, __f, __arg, __itvl)

Set the function, argument, and checkin interval for a sanity check.

struct os_sanity_check

#include <os_sanity.h> Structure representing a sanity check.

Public Functions

SLIST_ENTRY (os_sanity_check) sc_next

Next sanity check in the list.

Public Members

os_time_t **sc_checkin_last**

Time this check last ran successfully.

os_time_t **sc_checkin_itvl**

Interval this task should check in at.

os_sanity_check_func_t **sc_func**

Sanity check to run.

void *sc_arg

Argument to pass to sanity check.

6.2 System Modules

This section covers the core system functionality of the Apache Mynewt Operating System outside of the Kernel itself. This includes how you count statistics, log data and manage configuration of a device in the Apache Mynewt platform.

6.2.1 Config

Config subsystem is meant for persisting per-device configuration and runtime state for packages.

Configuration items are stored as key-value pairs, where both the key and the value are expected to be strings. Keys are divided into component elements, where packages register their subtree using the first element. E.g key `id/serial` consists of 2 components, `id` and `serial`. Package `sys/id` registered it's subtree of configuration elements to be under `id`.

There are convenience routines for converting value back and forth from string.

Handlers

Config handlers for subtree implement a set of handler functions. These are registered using a call to `conf_register()`.

- **ch_get**
This gets called when somebody asks for a config element value by it's name using `conf_get_value()`.
- **ch_set**
This is called when value is being set using `conf_set_value()`, and also when configuration is loaded from persisted storage with `conf_load()`.
- **ch_commit**
This gets called after configuration has been loaded in full. Sometimes you don't want individual configuration value to take effect right away, for example if there are multiple settings which are interdependent.
- **ch_export**
This gets called to dump all current configuration. This happens when `conf_save()` tries to save the settings, or when CLI is dumping current system configuration to console.

Persistence

Backend storage for the config can be either FCB, a file in filesystem, or both.

You can declare multiple sources for configuration; settings from all of these are restored when `conf_load()` is called.

There can be only one target for writing configuration; this is where data is stored when you call `conf_save()`, or `conf_save_one()`.

FCB read target is registered using `conf_fcb_src()`, and write target using `conf_fcb_dst()`. `conf_fcb_src()` as side-effect initializes the FCB area, so it must be called when calling `conf_fcb_dst()`. File read target is registered using `conf_file_src()`, and write target `conf_file_dst()`.

Convenience initialization of one config area can be enabled by setting either syscfg variable `CONFIG_FCB` or `CONFIG_NFFS`. These use other syscfg variables to figure which flash_map entry of BSP defines flash area, or which file to use. Check the `syscfg.yml` in `sys/config` package for more detailed description.

CLI

This can be enabled when shell package is enabled by setting syscfg variable `CONFIG_CLI` to 1.

Here you can set config variables, inspect their values and print both saved configuration as well as running configuration.

- **config dump**
Dump the current running configuration
- **config dump saved**
Dump the saved configuration. The values are printed in the ordered they're restored.
- **config <key>**
Print the value of config variable identified by <key>
- **config <key> <value>**
Set the value of config variable <key> to be <value>

Example: Device Configuration

This is a simple example, config handler only implements `ch_set` and `ch_export`. `ch_set` is called when value is restored from storage (or when set initially), and `ch_export` is used to write the value to storage (or dump to console).

```
static int8 foo_val;

struct conf_handler my_conf = {
    .ch_name = "foo",
    .ch_set = foo_conf_set,
    .ch_export = foo_conf_export
}

static int
foo_conf_set(int argc, char **argv, char *val)
{
    if (argc == 1) {
        if (!strcmp(argv[0], "bar")) {
            return CONF_VALUE_SET(val, CONF_INT8, foo_val);
        }
    }
    return OS_ENOENT;
}

static int
foo_conf_export(void (*func)(char *name, char *val), enum conf_export_tgt tgt)
{
    char buf[4];

    conf_str_from_value(CONF_INT8, &foo_val, buf, sizeof(buf));
    func("foo/bar", buf)
    return 0;
}
```

Example: Persist Runtime State

This is a simple example showing how to persist runtime state. Here there is only `ch_set` defined, which is used when restoring value from persisted storage.

There's a `os_callout` function which increments `foo_val`, and then persists the latest number. When system restarts, and calls `conf_load()`, `foo_val` will continue counting up from where it was before restart.

```
static int8 foo_val;

struct conf_handler my_conf = {
    .ch_name = "foo",
    .ch_set = foo_conf_set
}

static int
foo_conf_set(int argc, char **argv, char *val)
{
    if (argc == 1) {
```

(continues on next page)

(continued from previous page)

```

if (!strcmp(argv[0], "bar")) {
    return CONF_VALUE_SET(val, CONF_INT8, foo_val);
}
return OS_ENOENT;
}

static void
foo_callout(struct os_event *ev)
{
    struct os_callout *c = (struct os_callout *)ev;
    char buf[4];

    foo_val++;
    conf_str_from_value(CONF_INT8, &foo_val, buf, sizeof(buf));
    conf_save_one("foo/bar", bar);

    callout_reset(c, OS_TICKS_PER_SEC * 120);
}

```

API

enum **conf_type**

Type of configuration value.

Values:

enum **conf_export_tgt**

Parameter to commit handler describing where data is going to.

Values:

enumerator **CONF_EXPORT_PERSIST**

Value is to be persisted.

enumerator **CONF_EXPORT_SHOW**

Value is to be display.

typedef enum *conf_export_tgt* **conf_export_tgt_t**

typedef char *(***conf_get_handler_t**)(int argc, char **argv, char *val, int val_len_max)

Handler for getting configuration items, this handler is called per-configuration section.

Configuration sections are delimited by '/', for example:

- section/name/value

Would be passed as:

- argc = 3
- argv[0] = section
- argv[1] = name
- argv[2] = value

The handler returns the value into val, null terminated, up to val_len_max.

Param argc

The number of sections in the configuration variable

Param argv

The array of configuration sections

Param val

A pointer to the buffer to return the configuration value into.

Param val_len_max

The maximum length of the val buffer to copy into.

Return

A pointer to val or NULL if error.

```
typedef char *(*conf_get_handler_ext_t)(int argc, char **argv, char *val, int val_len_max, void *arg)
```

```
typedef int (*conf_set_handler_t)(int argc, char **argv, char *val)
```

Set the configuration variable pointed to by argc and argv.

See description of ch_get_handler_t for format of these variables. This sets the configuration variable to the shadow value, but does not apply the configuration change. In order to apply the change, call the ch_commit() handler.

Param argc

The number of sections in the configuration variable.

Param argv

The array of configuration sections

Param val

The value to configure that variable to

Return

0 on success, non-zero error code on failure.

```
typedef int (*conf_set_handler_ext_t)(int argc, char **argv, char *val, void *arg)
```

```
typedef int (*conf_commit_handler_t)(void)
```

Commit shadow configuration state to the active configuration.

Return

0 on success, non-zero error code on failure.

```
typedef int (*conf_commit_handler_ext_t)(void *arg)
```

```
typedef void (*conf_export_func_t)(char *name, char *val)
```

Called per-configuration variable being exported.

Param name

The name of the variable to export

Param val

The value of the variable to export

```
typedef int (*conf_export_handler_t)(conf_export_func_t export_func, conf_export_tgt_t tgt)
```

Export all of the configuration variables, calling the export_func per variable being exported.

Param export_func

The export function to call.

Param tgt

The target of the export, either for persistence or display.

Return

0 on success, non-zero error code on failure.

```
typedef int (*conf_export_handler_ext_t)(conf_export_func_t export_func, conf_export_tgt_t tgt, void *arg)
```

```
typedef void (*conf_store_load_cb)(char *name, char *val, void *cb_arg)
```

void **conf_init**(void)

void **conf_store_init**(void)

int **conf_register**(struct *conf_handler* *cf)

Register a handler for configurations items.

Parameters

- **cf** – Structure containing registration info.

Returns

0 on success, non-zero on failure.

int **conf_load**(void)

Load configuration from registered persistence sources.

Handlers for configuration subtrees registered earlier will be called for encountered values.

Returns

0 on success, non-zero on failure.

int **conf_load_one**(char *name)

Load configuration from a specific registered persistence source.

Handlers will be called for configuration subtree for encountered values.

Parameters

- **name** – of the configuration subtree.

Returns

0 on success, non-zero on failure.

```
int conf_ensure_loaded(void)  
    Loads the configuration if it hasn't been loaded since reboot.
```

Returns

0 on success, non-zero on failure.

```
void conf_export(conf_export_func_t export_func, enum conf_export_tgt tgt)  
    Export configuration via user defined function.
```

Parameters

- **export_func** – function to receive configuration entry
- **tgt** -- export target declaration

```
int conf_set_from_storage(void)  
    Config setting comes as a result of conf_load().
```

Returns

1 if yes, 0 if not.

```
int conf_save(void)  
    Save currently running configuration.
```

All configuration which is different from currently persisted values will be saved.

Returns

0 on success, non-zero on failure.

```
int conf_save_tree(char *name)  
    Save currently running configuration for configuration subtree.
```

Parameters

- **name** – Name of the configuration subtree.

Returns

0 on success, non-zero on failure.

```
int conf_save_one(const char *name, char *var)  
    Write a single configuration value to persisted storage (if it has changed value).
```

Parameters

- **name** – Name/key of the configuration item.
- **var** – Value of the configuration item.

Returns

0 on success, non-zero on failure.

```
int conf_set_value(char *name, char *val_str)  
    Set configuration item identified by name to be value val_str.
```

This finds the configuration handler for this subtree and calls it's set handler.

Parameters

- **name** – Name/key of the configuration item.
- **val_str** – Value of the configuration item.

Returns

0 on success, non-zero on failure.

```
char *conf_get_value(char *name, char *buf, int buf_len)
```

Get value of configuration item identified by `name`.

This calls the configuration handler `ch_get` for the subtree.

Configuration handler can copy the string to `buf`, the maximum number of bytes it will copy is limited by `buf_len`.

Return value will be pointer to beginning of the value. Note that this might, or might not be the same as `buf`.

Parameters

- `name` – Name/key of the configuration item.
- `val_str` – Value of the configuration item.

Returns

pointer to value on success, NULL on failure.

```
int conf_get_stored_value(char *name, char *buf, int buf_len)
```

Get stored value of configuration item identified by `name`.

This traverses the configuration area(s), and copies the value of the latest value.

Value is copied to `buf`, the maximum number of bytes it will copy is limited by `buf_len`.

Parameters

- `name` – Name/key of the configuration item.
- `val_str` – Value of the configuration item.

Returns

0 on success, non-zero on failure.

```
int conf_commit(char *name)
```

Call commit for all configuration handler.

This should apply all configuration which has been set, but not applied yet.

Parameters

- `name` – Name of the configuration subtree, or NULL to commit everything.

Returns

0 on success, non-zero on failure.

```
int conf_value_from_str(char *val_str, enum conf_type type, void *vp, int maxlen)
```

Convenience routine for converting value passed as a string to native data type.

Parameters

- `val_str` – Value of the configuration item as string.
- `type` – Type of the value to convert to.
- `vp` – Pointer to variable to fill with the decoded value.
- `vp` – Size of that variable.

Returns

0 on success, non-zero on failure.

`int conf_bytes_from_str(char *val_str, void *vp, int *len)`

Convenience routine for converting byte array passed as a base64 encoded string.

Parameters

- **val_str** – Value of the configuration item as string.
- **vp** – Pointer to variable to fill with the decoded value.
- **len** – Size of that variable. On return the number of bytes in the array.

Returns

0 on success, non-zero on failure.

`char *conf_str_from_value(enum conf_type type, void *vp, char *buf, int buf_len)`

Convenience routine for converting native data type to a string.

Parameters

- **type** – Type of the value to convert from.
- **vp** – Pointer to variable to convert.
- **buf** – Buffer where string value will be stored.
- **buf_len** – Size of the buffer.

Returns

0 on success, non-zero on failure.

`char *conf_str_from_bytes(void *vp, int vp_len, char *buf, int buf_len)`

Convenience routine for converting byte array into a base64 encoded string.

Parameters

- **vp** – Pointer to variable to convert.
- **vp_len** – Number of bytes to convert.
- **buf** – Buffer where string value will be stored.
- **buf_len** – Size of the buffer.

Returns

0 on success, non-zero on failure.

`void conf_lock(void)`

Locks the config package.

If another task tries to read or write a setting while the config package is locked, the operation will block until the package is unlocked. The locking task is permitted to read and write settings as usual. A call to `conf_lock()` should always be followed by `conf_unlock()`.

Typical usage of the config API does not require manual locking. This function is only needed for unusual use cases such as temporarily changing the destination medium for an isolated series of writes.

`void conf_unlock(void)`

Unlocks the config package.

This function reverts the effects of the `conf_lock()` function. Typical usage of the config API does not require manual locking.

`void conf_src_register(struct conf_store *cs)`

`void conf_dst_register(struct conf_store *cs)`

CONF_STR_FROM_BYTES_LEN(len)

Return the length of a configuration string from buffer length.

CONF_VALUE_SET(str, type, val)

Convert a string into a value of type.

struct **conf_handler**

#include <config.h> Configuration handler, used to register a config item/subtree.

Public Members

char ***ch_name**

The name of the configuration item/subtree.

bool **ch_ext**

Whether to use the extended callbacks.

false: standard true: extended

union conf_handler

Get configuration value.

union conf_handler

Set configuration value.

union conf_handler

Commit configuration value.

union conf_handler

Export configuration value.

void ***ch_arg**

Custom argument that gets passed to the extended callbacks.

struct **conf_store_itf**

#include <config_store.h>

struct **conf_store**

#include <config_store.h>

6.2.2 Logging

Mynewt log package supports logging of information within a Mynewt application. It allows packages to define their own log streams with separate names. It also allows an application to control the output destination of logs.

- *Description*
- *Syscfg Settings*
 - *Log*
 - *Log Handler*
 - *Configuring Logging for Packages that an Application Uses*
 - *Implementing a Package that Uses Logging*
 - *Log API and Log Levels*

Description

In the Mynewt OS, the log package comes in two versions:

- The `sys/log/full` package implements the complete log functionality and API.
- The `sys/log/stub` package implements stubs for the API.

Both packages export the `log` API, and any package that uses the `log` API must list `log` as a requirement in its `pkg.yml` file as follows:

```
pkg.req_apis:  
  - log
```

The application's `pkg.yml` file specifies the version of the log package to use. A project that requires the full logging capability must list the `sys/log/full` package as a dependency in its `pkg.yml` file:

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

You can use the `sys/log/stub` package if you want to build your application without logging to reduce code size.

Syscfg Settings

The `LOG_LEVEL` syscfg setting allows you to specify the level of logs to enable in your application. Only logs for levels higher or equal to the value of `LOG_LEVEL` are enabled. The amount of logs you include affects your application code size. `LOG_LEVEL: 0` specifies `LOG_LEVEL_DEBUG` and includes all logs. You set `LOG_LEVEL: 255` to disable all logging. The `#defines` for the log levels are specified in the `sys/log/full/include/log/log.h` file. For example the following setting corresponds to `LOG_LEVEL_ERROR`:

```
syscfg.vals:  
  LOG_LEVEL: 3
```

The `LOG_LEVEL` setting applies to all modules registered with the log package.

Log

Each log stream requires a `log` structure to define its logging properties.

Log Handler

To use logs, a log handler that handles the I/O from the log is required. The log package comes with three pre-built log handlers:

- console – streams log events directly to the console port. Does not support walking and reading.
- cbmem – writes/reads log events to a circular buffer. Supports walking and reading for access by newtmgr and shell commands.
- fcb – writes/reads log events to a *flash circular buffer*. Supports walking and reading for access by newtmgr and shell commands.

In addition, it is possible to create custom log handlers for other methods. Examples may include

- Flash file system
- Flat flash buffer
- Streamed over some other interface

To use logging, you typically do not need to create your own log handler. You can use one of the pre-built ones.

A package or an application must define a variable of type `struct log` and register a log handler for it with the log package. It must call the `log_register()` function to specify the log handler to use:

```
log_register(char *name, struct log *log, const struct log_handler *lh, void *arg, uint8_
↪t level)
```

The parameters are:

- name - Name of the log stream.
- log - Log instance to register,
- lh - Pointer to the log handler. You can specify one of the pre-built ones:
 - `&log_console_handler` for console
 - `&log_cbm_handler` for circular buffer
 - `&log_fcb_handler` for flash circular buffer
- arg - Opaque argument that the specified log handler uses. The value of this argument depends on the log handler you specify:
 - NULL for the `log_console_handler`.
 - Pointer to an initialized `cbmem` structure (see `util/cbmem` package) for the `log_cbm_handler`.
 - Pointer to an initialized `fcb_log` structure (see `fs/fcb` package) for the `log_fcb_handler`.

Typically, a package that uses logging defines a global variable, such as `my_package_log`, of type `struct log`. The package can call the `log_register()` function with default values, but usually an application will override the logging properties and where to log to. There are two ways a package can allow an application to override the values:

- Define system configuration settings that an application can set and the package can then call the `log_register()` function with the configuration values.

- Make the `my_package_log` variable external and let the application call the `log_register()` function to specify a log handler for its specific purpose.

Configuring Logging for Packages that an Application Uses

Here is an example of how an application can set the log handlers for the logs of the packages that the application includes.

In this example, the `package1` package defines the variable `package1_log` of type `struct log` and externs the variable. Similarly, the `package2` package defines the variable `package2_log` and externs the variable. The application sets logs for `package1` to use console and sets logs for `package2` to use a circular buffer.

```
#include <package1/package1.h>
#include <package2/package2.h>
#include <util/cbmem.h>

#include <log/log.h>

#define MAX_CBMEM_BUF 300
static uint8_t cbmem_buf[MAX_CBMEM_BUF];
static struct cbmem cbmem;

void app_log_init(void)
{
    log_register("package1_log", &package1_log, &log_console_handler, NULL, LOG_
    ↪SYSLEVEL);

    cbmem_init(&cbmem, cbmem_buf, MAX_CBMEM_BUF);
    log_register("package2_log", &package2_log, &log_cbmem_handler, &cbmem, LOG_
    ↪SYSLEVEL);

}
```

Implementing a Package that Uses Logging

This example shows how a package logs to console. The package registers default logging properties to use the console, but allows an application to override the values. It defines the `my_package_log` variable and makes it external so an application can override log handler.

Make the `my_package_log` variable external:

```
/* my_package.h */

/* pick a unique name here */
extern struct log my_package_log;
```

Define the `my_package_log` variable and register the console log handler:

```
/* my_package.c */

struct log my_package_log;

{

    ...

    /* register my log with a name to the system */
    log_register("log", &my_package_log, &log_console_handler, NULL, LOG_LEVEL_DEBUG);

    LOG_DEBUG(&my_package_log, LOG_MODULE_DEFAULT, "bla");
    LOG_DEBUG(&my_package_log, LOG_MODULE_DEFAULT, "bab");
}
```

Log API and Log Levels

Defines

LOG_HDR_SIZE

LOG_IMG_HASHLEN

LOG_FLAGS_IMG_HASH

LOG_BASE_ENTRY_HDR_SIZE

LOG_MODULE_STR(module)

LOG_DEBUG(__l, __mod, ...)

LOG_INFO(__l, __mod, __msg, ...)

LOG_WARN(__l, __mod, __msg, ...)

LOG_ERROR(__l, __mod, __msg, ...)

LOG_CRITICAL(__l, __mod, __msg, ...)

LOG_STATS_INC(log, name)

LOG_STATS_INCN(log, name, cnt)

Typedefs

```
typedef int (*log_walk_func_t)(struct log*, struct log_offset *log_offset, const void *dptr, uint16_t len)

typedef int (*log_walk_body_func_t)(struct log *log, struct log_offset *log_offset, const struct log_entry_hdr *hdr, const void *dptr, uint16_t len)

typedef int (*lh_read_func_t)(struct log*, const void *dptr, void *buf, uint16_t offset, uint16_t len)

typedef int (*lh_read_mbuf_func_t)(struct log*, const void *dptr, struct os_mbuf *om, uint16_t offset, uint16_t len)

typedef int (*lh_append_func_t)(struct log*, void *buf, int len)

typedef int (*lh_append_body_func_t)(struct log *log, const struct log_entry_hdr *hdr, const void *body, int body_len)

typedef int (*lh_append_mbuf_func_t)(struct log*, struct os_mbuf *om)

typedef int (*lh_append_mbuf_body_func_t)(struct log *log, const struct log_entry_hdr *hdr, struct os_mbuf *om)

typedef int (*lh_walk_func_t)(struct log*, log_walk_func_t walk_func, struct log_offset *log_offset)

typedef int (*lh_flush_func_t)(struct log*)

typedef int (*lh_storage_info_func_t)(struct log*, struct log_storage_info*)

typedef int (*lh_set_watermark_func_t)(struct log*, uint32_t)

typedef int (*lh_registered_func_t)(struct log*)
```

Functions

```
void log_init(void)

struct log *log_list_get_next(struct log*)

uint8_t log_module_register(uint8_t id, const char *name)

const char *log_module_get_name(uint8_t id)

int log_register(const char *name, struct log *log, const struct log_handler*, void *arg, uint8_t level)
```

```
void log_set_append_cb(struct log *log, log_append_cb *cb)
```

Configures the given log with the specified append callback.

A log's append callback is executed each time an entry is appended to the log.

Parameters

- **log** – The log to configure.
- **cb** – The callback to associate with the log.

```
struct log *log_find(const char *name)
```

Searches the list of registered logs for one with the specified name.

Parameters

- **name** – The name of the log to search for.

Returns

The sought after log if found, NULL otherwise.

```
int log_append_typed(struct log *log, uint8_t module, uint8_t level, uint8_t etype, void *data, uint16_t len)
```

Writes the raw contents of a flat buffer to the specified log.

NOTE: The flat buffer must have an initial padding of length LOG_HDR_SIZE. This padding is *not* reflected in the specified length. So, to log the string “abc”, you should pass the following arguments to this function:

<code>data: <padding>abc (total of `LOG_HDR_SIZE`+3 bytes.)</code>
<code>len: 3</code>

Parameters

- **log** – The log to write to.
- **module** – The log module of the entry to write.
- **level** – The severity of the log entry to write.
- **etype** – The type of data being written; one of the LOGETYPE_[...] constants.
- **data** – The flat buffer to write.
- **len** – The number of bytes in the *message body*.

Returns

0 on success; nonzero on failure.

```
int log_append_mbuf_typed_no_free(struct log *log, uint8_t module, uint8_t level, uint8_t etype, struct os_mbuf **om_ptr)
```

Logs the contents of the provided mbuf, only freeing the mbuf on failure.

Logs the contents of the provided mbuf, only freeing the mbuf on failure. On success, the mbuf remains allocated, but its structure may have been modified by pullup operations. The updated mbuf address is passed back to the caller via a write to the supplied mbuf pointer-to-pointer.

NOTE: The mbuf must have an initial padding of length LOG_HDR_SIZE. So, to log the string “abc”, you should pass an mbuf with the following characteristics:

<code>om_data: <padding>abc</code>
<code>om_len: `LOG_HDR_SIZE` + 3</code>

Parameters

- **log** – The log to write to.
- **module** – The module ID of the entry to write.
- **level** – The severity of the entry to write; one of the LOG_LEVEL_[...] constants.
- **etype** – The type of data to write; one of the LOGETYPE_[...] constants.
- **om_ptr** – Indirectly points to the mbuf to write. This function updates the mbuf address if it changes.

Returns

0 on success; nonzero on failure.

```
int log_append_mbuf_typed(struct log *log, uint8_t module, uint8_t level, uint8_t etype, struct os_mbuf *om)
```

Logs the contents of the provided mbuf.

Logs the contents of the provided mbuf. This function always frees the mbuf regardless of the outcome.

NOTE: The mbuf must have an initial padding of length LOG_HDR_SIZE. So, to log the string “abc”, you should pass an mbuf with the following characteristics:

```
om_data: <padding>abc  
om_len: `LOG_HDR_SIZE` + 3
```

Parameters

- **log** – The log to write to.
- **module** – The module ID of the entry to write.
- **level** – The severity of the entry to write; one of the LOG_LEVEL_[...] constants.
- **etype** – The type of data to write; one of the LOGETYPE_[...] constants.
- **om** – The mbuf to write.

Returns

0 on success; nonzero on failure.

```
int log_append_body(struct log *log, uint8_t module, uint8_t level, uint8_t etype, const void *body, uint16_t body_len)
```

Writes the contents of a flat buffer to the specified log.

Parameters

- **log** – The log to write to.
- **module** – The log module of the entry to write.
- **level** – The severity of the log entry to write.
- **etype** – The type of data being written; one of the LOGETYPE_[...] constants.
- **data** – The flat buffer to write.
- **len** – The number of bytes in the message body.

Returns

0 on success; nonzero on failure.

```
int log_append_mbuf_body_no_free(struct log *log, uint8_t module, uint8_t level, uint8_t etype, struct os_mbuf
                                *om)
```

Logs the contents of the provided mbuf, only freeing the mbuf on failure.

Logs the contents of the provided mbuf, only freeing the mbuf on failure. On success, the mbuf remains allocated, but its structure may have been modified by pullup operations. The updated mbuf address is passed back to the caller via a write to the supplied mbuf pointer-to-pointer.

Parameters

- **log** – The log to write to.
- **module** – The module ID of the entry to write.
- **level** – The severity of the entry to write; one of the LOG_LEVEL_[...] constants.
- **etype** – The type of data to write; one of the LOGETYPE_[...] constants.
- **om_ptr** – Indirectly points to the mbuf to write. This function updates the mbuf address if it changes.

Returns

0 on success; nonzero on failure.

```
int log_append_mbuf_body(struct log *log, uint8_t module, uint8_t level, uint8_t etype, struct os_mbuf *om)
```

Logs the contents of the provided mbuf.

Logs the contents of the provided mbuf. This function always frees the mbuf regardless of the outcome.

Parameters

- **log** – The log to write to.
- **module** – The module ID of the entry to write.
- **level** – The severity of the entry to write; one of the LOG_LEVEL_[...] constants.
- **etype** – The type of data to write; one of the LOGETYPE_[...] constants.
- **om** – The mbuf to write.

Returns

0 on success; nonzero on failure.

```
struct log *log_console_get(void)
```

```
void log_console_init(void)
```

```
static inline int log_append(struct log *log, uint8_t module, uint8_t level, void *data, uint16_t len)
```

Writes the raw contents of a flat buffer to the specified log.

NOTE: The flat buffer must have an initial padding of length LOG_HDR_SIZE. This padding is *not* reflected in the specified length. So, to log the string “abc”, you should pass the following arguments to this function:

data: <padding>abc (total of `LOG_HDR_SIZE`+3 bytes.)
len: 3

Parameters

- **log** – The log to write to.
- **module** – The log module of the entry to write.
- **level** – The severity of the log entry to write.

- **data** – The flat buffer to write.
- **len** – The number of byte in the *message body*.

Returns

0 on success; nonzero on failure.

```
static inline int log_append_mbuf_no_free(struct log *log, uint8_t module, uint8_t level, struct os_mbuf **om)
```

Logs the contents of the provided mbuf, only freeing the mbuf on failure.

Logs the contents of the provided mbuf, only freeing the mbuf on failure. On success, the mbuf remains allocated, but its structure may have been modified by pullup operations. The updated mbuf address is passed back to the caller via a write to the supplied mbuf pointer-to-pointer.

NOTE: The mbuf must have an initial padding of length LOG_HDR_SIZE. So, to log the string “abc”, you should pass an mbuf with the following characteristics:

```
om_data: <padding>abc  
om_len: `LOG_HDR_SIZE` + 3
```

Parameters

- **log** – The log to write to.
- **module** – The module ID of the entry to write.
- **level** – The severity of the entry to write; one of the LOG_LEVEL_[. . .] constants.
- **om_ptr** – Indirectly points to the mbuf to write. This function updates the mbuf address if it changes.

Returns

0 on success; nonzero on failure.

```
static inline int log_append_mbuf(struct log *log, uint8_t module, uint8_t level, struct os_mbuf *om)
```

Logs the contents of the provided mbuf.

Logs the contents of the provided mbuf. This function always frees the mbuf regardless of the outcome.

NOTE: The mbuf must have an initial padding of length LOG_HDR_SIZE. So, to log the string “abc”, you should pass an mbuf with the following characteristics:

```
om_data: <padding>abc  
om_len: `LOG_HDR_SIZE` + 3
```

Parameters

- **log** – The log to write to.
- **module** – The module ID of the entry to write.
- **level** – The severity of the entry to write; one of the LOG_LEVEL_[. . .] constants.
- **om** – The mbuf to write.

Returns

0 on success; nonzero on failure.

```
void log_printf(struct log *log, uint8_t module, uint8_t level, const char *msg, ...)
```

`int log_read(struct log *log, const void *dptr, void *buf, uint16_t off, uint16_t len)`

`int log_read_hdr(struct log *log, const void *dptr, struct log_entry_hdr *hdr)`

Reads a single log entry header.

Parameters

- **log** – The log to read from.
- **dptr** – Medium-specific data describing the area to read from; typically obtained by a call to `log_walk`.
- **hdr** – The destination header to read into.

Returns

0 on success; nonzero on failure.

`uint16_t log_hdr_len(const struct log_entry_hdr *hdr)`

Reads the header length.

Parameters

- **hdr** – Ptr to the header

Returns

Length of the header

`int log_read_body(struct log *log, const void *dptr, void *buf, uint16_t off, uint16_t len)`

Reads data from the body of a log entry into a flat buffer.

Parameters

- **log** – The log to read from.
- **dptr** – Medium-specific data describing the area to read from; typically obtained by a call to `log_walk`.
- **buf** – The destination buffer to read into.
- **off** – The offset within the log entry at which to start the read.
- **len** – The number of bytes to read.

Returns

The number of bytes actually read on success; -1 on failure.

`int log_read_mbuf(struct log *log, const void *dptr, struct os_mbuf *om, uint16_t off, uint16_t len)`

`int log_read_mbuf_body(struct log *log, const void *dptr, struct os_mbuf *om, uint16_t off, uint16_t len)`

Reads data from the body of a log entry into an mbuf.

Parameters

- **log** – The log to read from.
- **dptr** – Medium-specific data describing the area to read from; typically obtained by a call to `log_walk`.
- **om** – The destination mbuf to read into.
- **off** – The offset within the log entry at which to start the read.
- **len** – The number of bytes to read.

Returns

The number of bytes actually read on success; -1 on failure.

```
int log_walk(struct log *log, log_walk_func_t walk_func, struct log_offset *log_offset)
int log_walk_body(struct log *log, log_walk_body_func_t walk_body_func, struct log_offset *log_offset)
```

Applies a callback to each message in the specified log.

Similar to `log_walk`, except it passes the message header and body separately to the callback.

Parameters

- **log** – The log to iterate.
- **walk_body_func** – The function to apply to each log entry.
- **log_offset** – Specifies the range of entries to process. Entries not matching these criteria are skipped during the walk.

Returns

0 if the walk completed successfully; nonzero on error or if the walk was aborted.

```
int log_flush(struct log *log)
```

```
int log_walk_body_section(struct log *log, log_walk_body_func_t walk_body_func, struct log_offset
                           *log_offset)
```

Walking a section of FCB.

Parameters

- **log** – The log to iterate.
- **walk_body_func** – The function to apply to each log entry.
- **log_offset** – Specifies the range of entries to process. Entries not matching these criteria are skipped during the walk.

Returns

0 if the walk completed successfully; nonzero on error or if the walk was aborted.

```
uint8_t log_level_get(uint8_t module)
```

Retrieves the globally configured minimum log level for the specified module ID.

Writes with a level less than the module's minimum level are discarded.

Parameters

- **module** – The module whose level should be retrieved.

Returns

The configured minimum level, or 0 (LOG_LEVEL_DEBUG) if unconfigured.

```
int log_level_set(uint8_t module, uint8_t level)
```

Sets the globally configured minimum log level for the specified module ID.

Writes with a level less than the module's minimum level are discarded.

Parameters

- **module** – The module to configure.
- **level** – The minimum level to assign to the module (0-15, inclusive).

```
void log_set_level(struct log *log, uint8_t level)
```

Set log level for a logger.

Parameters

- **log** – The log to set level to.

- **level** – New log level

`uint8_t log_get_level(const struct log *log)`

Get log level for a logger.

Parameters

- **log** – The log to set level to.

Returns

current value of log level.

`void log_set_max_entry_len(struct log *log, uint16_t max_entry_len)`

Set maximum length of an entry in the log.

If set to 0, no check will be made for maximum write length. Note that this is maximum log body length; the log entry header is not included in the check.

Parameters

- **log** – Log to set max entry length
- **level** – New max entry length

`uint32_t log_get_last_index(struct log *log)`

Return last entry index in log.

Parameters

- **log** – Log to check last entry index from.

Returns

last entry index

`int log_storage_info(struct log *log, struct log_storage_info *info)`

Return information about log storage.

This return information about size and usage of storage on top of which log instance is created.

Parameters

- **log** – The log to query.
- **info** – The destination to write information to.

Returns

0 on success, error code otherwise

`void log_set_rotate_notify_cb(struct log *log, log_notify_rotate_cb *cb)`

Assign a callback function to be notified when the log is about to be rotated.

Parameters

- **log** – The log
- **cb** – The callback function to be executed.

`int log_set_watermark(struct log *log, uint32_t index)`

Set watermark on log.

This sets watermark on log item with given index. This information is used to calculate size of entries which were logged after watermark item, i.e. unread items. The watermark is stored persistently for each log.

Parameters

- **log** – The log to set watermark on.

- **index** – The index of a watermarked item.

Returns

0 on success, error code otherwise.

```
int log_fill_current_img_hash(struct log_entry_hdr *hdr)
```

Fill log current image hash.

Parameters

- **hdr** – Ptr to the header

Returns

0 on success, non-zero on failure

Variables

```
const struct log_handler log_console_handler
```

```
const struct log_handler log_cbmem_handler
```

```
const struct log_handler log_fcb_handler
```

struct log_offset

#include <log.h> Used for walks and reads; indicates part of log to access.

Public Members

```
int64_t lo_ts
```

```
uint32_t lo_index
```

```
uint32_t lo_data_len
```

```
void *lo_arg
```

```
bool lo_walk_backward
```

struct log_storage_info

#include <log.h> Log storage information.

Public Members

```
uint32_t size  
  
uint32_t used  
  
uint32_t used_unread  
  
struct log_handler  
#include <log.h>
```

Public Members

```
int log_type  
  
lh_read_func_t log_read  
  
lh_read_mbuf_func_t log_read_mbuf  
  
lh_append_func_t log_append  
  
lh_append_body_func_t log_append_body  
  
lh_append_mbuf_func_t log_append_mbuf  
  
lh_append_mbuf_body_func_t log_append_mbuf_body  
  
lh_walk_func_t log_walk  
  
lh_walk_func_t log_walk_sector  
  
lh_flush_func_t log_flush  
  
lh_storage_info_func_t log_storage_info  
  
lh_set_watermark_func_t log_set_watermark  
  
lh_registered_func_t log_registered  
  
struct log  
#include <log.h>
```

Public Functions

```
STAILQ_ENTRY (log) l_next
```

```
STATS_SECT_DECL (logs) l_stats
```

Public Members

```
const char *l_name
```

```
const struct log_handler *l_log
```

```
void *l_arg
```

```
log_append_cb *l_append_cb
```

```
log_notify_rotate_cb *l_rotate_notify_cb
```

```
uint8_t l_level
```

```
uint16_t l_max_entry_len
```

6.2.3 Statistics Module

The statistics module allows application, libraries, or drivers to record statistics that can be shown via the Newtmgr tool and console.

This allows easy integration of statistics for troubleshooting, maintenance, and usage monitoring.

By creating and registering your statistics, they are automatically included in the Newtmgr shell and console APIs.

- *Implementation Details*
 - *Enabling Statistic Names*
 - *Adding Stats to your code.*
 - *Include the stats header file*
 - *Define a stats section*
 - *Declaring a variable to hold the stats*
 - *Define the stats section name table*
 - *Implement stats in your code.*
 - *Initialize the statistics*
 - *Register the statistic section*

- *Retrieving stats through console or Newtmgr*
- *A note on multiple stats sections*
- *API*

Implementation Details

A statistic is an unsigned integer that can be set by the code. When building stats, the implementer chooses the size of the statistic depending on the frequency of the statistic and the resolution required before the counter wraps.

Typically the stats are incremented upon code events; however, they are not limited to that purpose.

Stats are organized into sections. Each section of stats has its own name and can be queried separately through the API. Each section of stats also has its own statistic size, allowing the user to separate large (64-bit) statistics from small (16 bit statistics). NOTE: It is not currently possible to group different size stats into the same section. Please ensure all stats in a section have the same size.

Stats sections are currently stored in a single global stats group.

Statistics are stored in a simple structure which contains a small stats header followed by a list of stats. The stats header contains:

```
struct stats_hdr {
    char *s_name;
    uint8_t s_size;
    uint8_t s_cnt;
    uint16_t s_pad1;
#ifndef MYNEWT_VAL(STATS_NAMES)
    const struct stats_name_map *s_map;
    int s_map_cnt;
#endif
    STAILQ_ENTRY(stats_hdr) s_next;
};
```

The fields define within the `#if MYNEWT_VAL(STATS_NAME)` directive are only included when the `STATS_NAMES` syscfg setting is set to 1 and enables use statistic names.

Enabling Statistic Names

By default, statistics are queried by number. You can use the `STATS_NAMES` syscfg setting to enable statistic names and view the results by name. Enabling statistic names provides better descriptions in the reported statistics, but takes code space to store the strings within the image.

To enable statistic names, set the `STATS_NAMES` value to 1 in the application `syscfg.yml` file or use the `newt target set` command to set the syscfg setting value. Here are examples for each method:

Method 1 - Set the value in the application `syscfg.yml` files:

```
# Package: apps/myapp

syscfg.vals:
    STATS_NAMES: 1
```

Method 2 - Set the target `syscfg` variable:

```
newt target set myapp syscfg=STATS_NAMES=1
```

Note: This `newt target set` command only sets the `syscfg` variable for the `STATS_NAMES` setting as an example. For your target, you should set the `syscfg` variable with the other settings that you want to override.

Adding Stats to your code.

Creating new stats table requires the following steps.

- Include the stats header file
- Define a stats section
- Declare an instance of the section
- Define the stat sections names table
- Implement stat in your code
- Initialize the stats
- Register the stats

Include the stats header file

Add the stats library to your `pkg.yml` file for your package or app by adding this line to your package dependencies.

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

Add this include directive to code files using the stats library.

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

Define a stats section

You must use the `stats.h` macros to define your stats table. A stats section definition looks like this.

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

In this case we chose to make the stats 32-bits each. `stats.h` supports three different stats sizes through the following macros:

- `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

When this compiles/pre-processes, it produces a structure definition like this

```
struct stats_my_stat_section {
    struct stats_hdr s_hdr;
    uint32_t sattempt_stat;
    uint32_t serror_stat;
};
```

You can see that the defined structure has a small stats structure header and the two stats we have defined.

Depending on whether these stats are used in multiple modules, you may need to include this definition in a header file.

Declaring a variable to hold the stats

Declare the global variable to hold your statistics. Since it is possible to have multiple copies of the same section (for example a stat section for each of 5 identical peripherals), the variable name of the stats section must be unique.

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

Again, if your stats section is used in multiple C files you will need to include the above definition in one of the C files and ‘extern’ this declaration in your header file.

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

Define the stats section name table

Whether or not you have STATS_NAMES enabled, you must define a stats name table. If STATS_NAMES is not enabled, this will not take any code space or image size.

```
/* 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)
```

When compiled by the preprocessor, it creates a structure that looks like this.

```
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" },
};
```

This table will allow the UI components to find a nice string name for the stat.

Implement stats in your code.

You can use the STATS_INC or STATS_INCN macros to increment your statistics within your C-code. For example, your code may do this:

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

Initialize the statistics

You must initialize the stats so they can be operated on by the stats library. As per our example above, it would look like the following.

This tells the system how large each statistic is and the number of statistics in the section. It also initialize the name information for the statistics if enabled as shown above.

```
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);
```

Register the statistic section

If you want the system to know about your stats, you must register them.

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

There is also a method that does initialization and registration at the same time, called `stats_init_and_reg`.

Retrieving stats through console or Newtmgr

If you enable console in your project you can see stats through the serial port defined.

This is the stats as shown from the example above with names enabled.

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

This is the stats as shown from the example without names enabled.

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

A note on multiple stats sections

If you are implementing a device with multiple instances, you may want multiple stats sections with the exact same format.

For example, suppose I write a driver for an external distance sensor. My driver supports up to 5 sensors and I want to record the stats of each device separately.

This works identically to the example above, except you would need to register each one separately with a unique name. The stats system will not let two sections be entered with the same name.

API

Defines

`STATS_HDR_F_PERSIST`

The stat group is periodically written to sys/config.

`STATS_SECT_DECL(__name)`

`STATS_SECT_START(__name)`

`STATS_SECT_END`

`STATS_SECT_VAR(__var)`

`STATS_HDR(__sectname)`

`STATS_SIZE_16`

`STATS_SIZE_32`

`STATS_SIZE_64`

`STATS_SECT_ENTRY(__var)`

`STATS_SECT_ENTRY16(__var)`

`STATS_SECT_ENTRY32(__var)`

`STATS_SECT_ENTRY64(__var)`

`STATS_RESET(__sectvarname)`

Resets all stats in the provided group to 0.

NOTE: This must only be used with non-persistent stat groups.

`STATS_SIZE_INIT_PARMS(__sectvarname, __size)`

`STATS_GET(__sectvarname, __var)`

`STATS_SET_RAW(__sectvarname, __var, __val)`

`STATS_SET(__sectvarname, __var, __val)`

STATS_INCN_RAW(__sectvarname, __var, __n)

Adjusts a stat's in-RAM value by the specified delta.

For non-persistent stats, this is more efficient than [STATS_INCN\(\)](#). This must only be used with non-persistent stats; for persistent stats the behavior is undefined.

Parameters

- **__sectvarname** – The name of the stat group containing the stat to modify.
- **__var** – The name of the individual stat to modify.
- **__n** – The amount to add to the specified stat.

STATS_INC_RAW(__sectvarname, __var)

Increments a stat's in-RAM value.

For non-persistent stats, this is more efficient than [STATS_INCN\(\)](#). This must only be used with non-persistent stats; for persistent stats the behavior is undefined.

Parameters

- **__sectvarname** – The name of the stat group containing the stat to modify.
- **__var** – The name of the individual stat to modify.

STATS_INCN(__sectvarname, __var, __n)

Adjusts a stat's value by the specified delta.

If the specified stat group is persistent, this also schedules the group to be flushed to disk.

Parameters

- **__sectvarname** – The name of the stat group containing the stat to modify.
- **__var** – The name of the individual stat to modify.
- **__n** – The amount to add to the specified stat.

STATS_INC(__sectvarname, __var)

Increments a stat's value.

If the specified stat group is persistent, this also schedules the group to be flushed to disk.

Parameters

- **__sectvarname** – The name of the stat group containing the stat to modify.
- **__var** – The name of the individual stat to modify.

STATS_CLEAR(__sectvarname, __var)

STATS_NAME_MAP_NAME(__sectname)

STATS_NAME_START(__sectname)

STATS_NAME(__sectname, __entry)

STATS_NAME_END(__sectname)

STATS_NAME_INIT_PARMS(__name)

STATS_PERSISTED_SECT_START(__name)

Starts the definition of a persisted stat group.

- o Follow with invocations of the STATS_SECT_ENTRY[...] macros to define individual stats.
- o Use STATS_SECT_END to complete the group definition.

STATS_PERSISTED_HDR(__sectname)**STATS_PERSIST_SCHED(hdrp_)**

(private) Starts the provided stat group's persistence timer if it is a persistent group.

This should be used whenever a statistic's value changes. This is a no-op for non-persistent stat groups.

Typedefs

```
typedef int (*stats_walk_func_t)(struct stats_hdr*, void*, char*, uint16_t)
```

```
typedef int (*stats_group_walk_func_t)(struct stats_hdr*, void*)
```

Functions**STAILQ_HEAD (stats_registry_list, stats_hdr)**

```
int stats_init(struct stats_hdr *shdr, uint8_t size, uint8_t cnt, const struct stats_name_map *map, uint8_t map_cnt)
```

```
int stats_register(const char *name, struct stats_hdr *shdr)
```

```
int stats_init_and_reg(struct stats_hdr *shdr, uint8_t size, uint8_t cnt, const struct stats_name_map *map, uint8_t map_cnt, const char *name)
```

```
void stats_reset(struct stats_hdr *shdr)
```

```
int stats_walk(struct stats_hdr*, stats_walk_func_t, void*)
```

```
int stats_group_walk(stats_group_walk_func_t, void*)
```

```
struct stats_hdr *stats_group_find(const char *name)
```

```
int stats_mgmt_register_group(void)
```

```
int stats_shell_register(void)
```

```
void stats_persist_sched(struct stats_hdr *hdr)
```

(private) Starts the provided stat group's persistence timer.

This should be used whenever a statistic's value changes. This is a no-op for non-persistent stat groups.

```
int stats_persist_flush(void)
```

Flushes to disk all persisted stat groups with pending writes.

Returns

0 on success; nonzero on failure.

```
int stats_persist_init(struct stats_hdr *hdr, uint8_t size, uint8_t cnt, const struct stats_name_map *map,
                      uint8_t map_cnt, os_time_t persist_delay)
```

Initializes a persistent stat group.

This function must be called before any other stats API functions are applied to the specified stat group. This is typically done during system startup.

Example usage: `STATS_PERSISTED_SECT_START(my_stats) STATS_SECT_ENTRY(stat1)`
`STATS_SECT_END(my_stats)`

`STATS_NAME_START(my_stats) STATS_SECT_ENTRY(my_stats, stat1) STATS_NAME_END(my_stats)`

`rc = stats_persist_init(STATS_PERSISTED_HDR(my_stats), STATS_SIZE_INIT_PARMS(my_stats,
STATS_SIZE_32), STATS_NAME_INIT_PARMS(my_stats), 1 * OS_TICKS_PER_SEC); // One second.`

Parameters

- **hdr** – The header of the stat group to initialize. Use the `STATS_PERSISTED_HDR()` macro to generate this argument.
- **size** – The size, in bytes, of each statistic. Use the `STATS_SIZE_INIT_PARMS()` macro to generate this and the **cnt** arguments.
- **cnt** – The number of statistics in the group. Use the `STATS_SIZE_INIT_PARMS()` macro to generate this and the **size** arguments.
- **map** – Maps each stat to a human-readable name. Use the `STATS_NAME_INIT_PARMS()` macro to generate this and the **map_cnt** arguments.
- **map_cnt** – The number of names in **map**. Use the `STATS_NAME_INIT_PARMS()` macro to generate this and the **map** arguments.
- **persist_delay** – The delay, in OS ticks, before the stat group is flushed to disk after modification.

Returns

0 on success; nonzero on failure.

Variables

```
struct stats_registry_list g_stats_registry
```

```
uint16_t snm_off
```

```
char *snm_name
```

```
struct stats_name_map
```

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

Public Members

```
uint16_t snm_off
```

```
char *snm_name
```

```
struct stats_hdr
```

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

Public Functions

```
STAILQ_ENTRY (stats_hdr) s_next
```

Public Members

```
const char *s_name
```

```
uint8_t s_size
```

```
uint8_t s_cnt
```

```
uint16_t s_flags
```

```
const struct stats_name_map *s_map
```

```
int s_map_cnt
```

```
struct stats_persisted_hdr
```

```
#include <stats.h> Header describing a persistent stat group.
```

A pointer to a regular *stats_hdr* can be safely cast to a pointer to *stats_persisted_hdr* (and vice-versa) if the STATS_HDR_F_PERSIST flag is set.

Public Members

```
struct stats_hdr sp_hdr
```

```
struct os_callout sp_persist_timer
```

```
os_time_t sp_persist_delay
```

6.2.4 Console

The console is an operating system window where users interact with the OS subsystems or a console application. A user typically inputs text from a keyboard and reads the OS output text on a computer monitor. The text is sent as a sequence of characters between the user and the OS.

- *Description*
 - *Using the Full Console Package*
 - *Using the Stub Console Package*
 - *Using the Minimal Console Package*
 - *Output to the Console*
 - *Input from the Console*
 - *Mynewt 1.0 Console API*
 - *Mynewt 1.1 Console API*
- *API*

You can configure the console to communicate via a UART or the SEGGER Real Time Terminal (RTT) . The CONSOLE_UART syscfg setting enables the communication via a UART and is enabled by default. The CONSOLE_RTT setting enables the communication via the RTT and is disabled by default. When the CONSOLE_UART setting is enabled, the following settings apply:

- CONSOLE_UART_DEV: Specifies the UART device to use. The default is `uart0`.
- CONSOLE_UART_BAUD: Specifies the UART baud rate. The default is 115200.
- CONSOLE_UART_FLOW_CONTROL: Specifies the UART flow control. The default is `UART_FLOW_CONTROL_NONE`.
- CONSOLE_UART_TX_BUF_SIZE: Specifies the transmit buffer size and must be a power of 2. The default is 32.

The CONSOLE_TICKS setting enables the console to print the current OS ticks in each output line.

Notes:

- SEGGER RTT support is not available in the Mynewt 1.0 console package.
- The console package is initialized during system initialization (sysinit) so you do not need to initialize the console. However, if you use the Mynewt 1.0 console API to read from the console, you will need to call the `console_init()` function to enable backward compatibility support.

Description

In the Mynewt OS, the console library comes in three versions:

- The `sys/console/full` package implements the complete console functionality and API.
- The `sys/console/stub` package implements stubs for the API.
- The `sys/console/minimal` package implements minimal console functionality of reading from and writing to console. It implements the `console_read()` and `console_write()` functions and stubs for all the other console functions.

All the packages export the `console` API, and any package that uses the console API must list `console` as a requirement its `pkg.yml` file:

```

pkg.name: sys/shell
pkg.deps:
  - "@apache-mynewt-core/kernel/os"
  - "@apache-mynewt-core/encoding/base64"
  - "@apache-mynewt-core/time/datetime"
  - "@apache-mynewt-core/util/crc"
pkg.req_apis:
  - console

```

The project `pkg.yml` file also specifies the version of the `console` package to use.

Using the Full Console Package

A project that requires the full console capability must list the `sys/console/full` package as a dependency in its `pkg.yml` file.

An example is the `slinky` application. It requires the full console capability and has the following `pkg.yml` file:

```

pkg.name: apps/slinky
pkg.deps:
  - "@apache-mynewt-core/test/flash_test"
  - "@apache-mynewt-core/mgmt/imgmgr"
  - "@apache-mynewt-core/mgmt/newtmgr"
  - "@apache-mynewt-core/mgmt/newtmgr/transport/nmgr_shell"
  - "@apache-mynewt-core/kernel/os"
  - "@mcuboot/boot/bootutil"
  - "@apache-mynewt-core/sys/shell"
  - "@apache-mynewt-core/sys/console/full"
  ...
  - "@apache-mynewt-core/sys/id"

```

Using the Stub Console Package

A project that uses console stub API must list the `sys/console/stub` package as a dependency in its `pkg.yml` file.

Examples of when a project would use the console stubs might be:

- A project may not have a physical console (e.g. a UART port to connect a terminal to) but may have a dependency on a package that has console capability.
- A bootloader project where we want to keep the size of the image small. It includes the `kernel/os` package that can print out messages on a console (e.g. if there is a hard fault). However, we do not want to use any console I/O capability in this particular bootloader project to keep the size small.

The project would use the console stub API and has the following `pkg.yml` file:

Another example would be the bootloader project where we want to keep the size of the image small. It includes the `libs/os` pkg that can print out messages on a console (e.g. if there is a hard fault) and the `libs/util` pkg that uses full console (but only if SHELL is present to provide a CLI). However, we do not want to use any console I/O capability in this particular bootloader project to keep the size small. We simply use the console stub instead, and the `pkg.yml` file for the project boot pkg looks like the following:

```
pkg.name: apps/boot
pkg.deps:
  - "@mcuboot/boot/bootutil"
  - "@apache-mynewt-core/kernel/os"
  - "@apache-mynewt-core/sys/console/stub"
```

Using the Minimal Console Package

There might be projects that need to read and write data on a serial connection but do not need the full console capability. An example might be a project that supports serial image upgrade but does not need full newtmgr capability. The project would use the console minimal API and has the following `pkg.yml` file:

```
pkg.name: apps/boot
pkg.type: app
pkg.description: Boot loader application.
pkg.author: "Apache Mynewt <dev@mynewt.apache.org>"
pkg.homepage: "http://mynewt.apache.org/"
pkg.keywords:
  - loader

pkg.deps:
  - "@mcuboot/boot/bootutil"
  - "@apache-mynewt-core/kernel/os"
  - "@apache-mynewt-core/sys/console/stub"

pkg.deps.BOOT_SERIAL.OVERWRITE:
  - "@apache-mynewt-core/sys/console/minimal"
  - "@apache-mynewt-core/boot/boot_serial"
```

Output to the Console

You use the `console_write()` function to write raw output and the `console_printf()` function to write a C-style formatted string to the console.

Input from the Console

The following syscfg settings control input from the console:

- `CONSOLE_INPUT`: Enables input from the console. The setting is enabled by default.
- `CONSOLE_ECHO`: Enables echoing of the received data back to the console. Echoing is enabled by default. Terminal programs expect this, and is a way for the user to know that the console is connected and responsive. You can also use the `console_echo()` function to set echo on or off programmatically.
- `CONSOLE_MAX_INPUT_LEN`: Specifies the maximum input line length.

The Mynewt 1.1 console package adds a new API for reading input data from the console. The package supports backward compatibility for the Mynewt 1.0 console API. The steps you use to receive data from the console for each API version are provided below.

Mynewt 1.0 Console API

To use the Mynewt 1.0 console API for reading input from the console, you perform the follow steps:

1. Call the `console_init()` function and pass either a pointer to a callback function or NULL for the argument. The console calls this callback function, if specified, when it receives a full line of data.
2. Call the `console_read()` function to read the input data.

Note: The CONSOLE_COMPAT syscfg setting must be set to 1 to enable backward compatibility support. The setting is enabled by default.

Mynewt 1.1 Console API

Mynewt 1.1 console API adds the `console_set_queues()` function. An application or the package, such as the shell, calls this function to specify two event queues that the console uses to manage input data buffering and to send notification when a full line of data is received. The two event queues are used as follows:

- **avail_queue:** Each event in this queue indicates that a buffer is available for the console to use for buffering input data.

The caller must initialize the avail_queue and initialize and add an *Event Queues* to the avail_queue before calling the `console_set_queues()` function. The fields for the event should be set as follows:

- `ev_cb`: Pointer to the callback function to call when a full line of data is received.
- `ev_arg`: Pointer to a `console_input` structure. This structure contains a data buffer to store the current input.

The console removes an event from this queue and uses the `console_input` buffer from this event to buffer the received characters until it receives a new line, ‘\n’, character. When the console receives a full line of data, it adds this event to the **lines_queue**.

- **lines_queue:** Each event in this queue indicates a full line of data is received and ready for processing. The console adds an event to this queue when it receives a full line of data. This event is the same event that the console removes from the avail_queue.

The task that manages the lines_queue removes an event from the queue and calls the event callback function to process the input line. The event callback function must add the event back to the avail_queue when it completes processing the current input data, and allows the console to use the `console_input` buffer set for this event to buffer input data.

We recommend that you use the OS default queue for the lines_queue so that the callback is processed in the context of the OS main task. If you do not use the OS default event queue, you must initialize an event queue and create a task to process events from the queue.

Note: If the callback function needs to read another line of input from the console while processing the current line, it may use the `console_read()` function to read the next line of input from the console. The console will need another `console_input` buffer to store the next input line, so two events, initialized with the pointers to the callback and the `console_input` buffer, must be added to the avail_queue.

Minimal Console Example

Here is a code excerpt that shows how to use the `console_set_queues()` function. The example adds one event to the `avail_queue` and uses the OS default event queue for the `lines_queue`.

```
static void myapp_process_input(struct os_event *ev);

static struct os_eventq avail_queue;

static struct console_input myapp_console_buf;

static struct os_event myapp_console_event = {
    .ev_cb = myapp_process_input,
    .ev_arg = &myapp_console_buf
};

/* Event callback to process a line of input from console. */
static void
myapp_process_input(struct os_event *ev)
{
    char *line;
    struct console_input *input;

    input = ev->ev_arg;
    assert (input != NULL);

    line = input->line;
    /* Do some work with line */

    ...
    /* Done processing line. Add the event back to the avail_queue */
    os_eventq_put(&avail_queue, ev);
    return;
}

static void
myapp_init(void)
{
    os_eventq_init(&avail_queue);
    os_eventq_put(&avail_queue, &myapp_console_event);

    console_set_queues(&avail_queue, os_eventq_dflt_get());
}
```

Full Console Example

For the full console, setting the queue is done via `console_line_queue_set()`. This example uses the OS default event queue for calling the line received callback.

```
static void myapp_process_input(struct os_event *ev);

static struct console_input myapp_console_buf;

static struct os_event myapp_console_event = {
    .ev_cb = myapp_process_input,
    .ev_arg = &myapp_console_buf
};

/* Event callback to process a line of input from console. */
static void
myapp_process_input(struct os_event *ev)
{
    char *line;
    struct console_input *input;

    input = ev->ev_arg;
    assert (input != NULL);

    line = input->line;
    /* Do some work with line */

    ...
    /* Done processing line. Add the event back to the avail_queue */
    console_line_event_put(ev);
    return;
}

static void
myapp_init(void)
{
    console_line_event_put(&myapp_console_event);
    console_line_queue_set(os_eventq_dflt_get());
}
```

API

Typedefs

`typedef void (*console_rx_cb)(void)`

`typedef int (*console_append_char_cb)(char *line, uint8_t byte)`

`typedef void (*completion_cb)(char *str, console_append_char_cb cb)`

Functions

```
void console_deinit(void)
    De initializes the UART console.

void console_reinit(void)
    Re Initializes the UART console.

int console_init(console_rx_cb rx_cb)

int console_is_init(void)

void console_write(const char *str, int cnt)

int console_read(char *str, int cnt, int *newline)

void console_blocking_mode(void)

void console_non_blocking_mode(void)

void console_echo(int on)

int console_vprintf(const char *fmt, va_list ap)

inline int console_printf (const char *fmt,...) __attribute__((format(sprintf
int void console_set_completion_cb(completion_cb cb)

int console_handle_char(uint8_t byte)

void console_line_queue_set(struct os_eventq *evq)

void console_line_event_put(struct os_event *ev)

static inline void console_silence(bool silent)
    Silences console output, input is still active.
```

Parameters

- **silent** – Let console know if it needs to be silent, true for silence, false otherwise

```
static inline void console_silence_non_nlip(bool silent)
```

Silences non nlip console output, input is still active.

Parameters

- **silent** – Let non nlip console know if it needs to be silent, true for silence, false otherwise

```
static inline void console_ignore_non_nlip(bool ignore)
```

Ignores console input that is non nlip, output is still active.

Parameters

- **ignore** – Lets console know if non nlip input should be disabled, true for ignore input, false otherwise

```
int console_out(int character)
```

```
void console_rx_restart(void)
```

```
int console_lock(int timeout)
int console_unlock(void)
```

void console_prompt_set(const char *prompt, const char *line)

Set prompt and current input line.

This shows prompt with current input editor, cursor is placed at the end of line.

Parameters

- **prompt** – non-editable part of user input line
- **line** – editable part

Variables

bool g_console_silence

Global indicating whether console is silent or not.

bool g_console_silence_non_nlip

Global indicating whether non nlip console is silent or not.

bool g_console_ignore_non_nlip

Global indicating whether console input is disabled or not.

int console_is_midline

struct console_input

#include <console.h> The *console_input* data structure represents a console input buffer.

Each event added to the console avail_queue must have the ev_arg field point to a *console_input* structure.

Public Members

char **line**[MYNEWT_VAL(CONSOLE_MAX_INPUT_LEN)]

Data buffer that the console uses to save received characters until a new line is received.

6.2.5 Shell

The shell runs above the console and provides two functionalities:

- Processes console input. See the *Enabling the Console and Shell tutorial* for example of the shell.
- Implements the *newtmgr* line protocol over serial transport.

The shell uses the OS default event queue for shell events and runs in the context of the main task. An application can, optionally, specify a dedicated event queue for the shell to use.

- *Description*
 - Processing Console Input Commands
 - Processing Newtmgr Line Protocol Over Serial Transport
- *Data Structures*
- *API*

The sys/shell package implements the shell. To use the shell you must:

- Include the sys/shell package.
- Set the SHELL_TASK syscfg setting value to 1 to enable the shell.

Note: The functions for the shell API are only compiled and linked with the application when the SHELL_TASK setting is enabled. When you develop a package that supports shell commands, we recommend that your package define:

1. A syscfg setting that enables shell command processing for your package, with a restriction that when this setting is enabled, the SHELL_TASK setting must also be enabled.
2. A conditional dependency on the sys/shell package when the setting defined in 1 above is enabled.

Here are example definitions from the syscfg.yml and pkg.yml files for the sys/log/full package. It defines the LOG_CLI setting to enable the log command in the shell:

```
# sys/log/full syscfg.yml
LOG_CLI:
    description: 'Expose "log" command in shell.'
    value: 0
    restrictions:
        - SHELL_TASK

# sys/log/full pkg.yml
pkg.deps.LOG_CLI:
    - "@apache-mynewt-core/sys/shell"
```

Description

Processing Console Input Commands

The shell's first job is to direct incoming commands to other subsystems. It parses the incoming character string into tokens and uses the first token to determine the subsystem command handler to call to process the command. When the shell calls the command handler, it passes the other tokens as arguments to the handler.

Registering Command Handlers

A package that implements a shell command must register a command handler to process the command.

New in release 1.1: The shell supports the concept of modules and allows a package to group shell commands under a name space. To run a command in the shell, you enter the module name and the command name. You can set a default module, using the `select` command, so that you only need to enter the command name to run a command from the default module. You can switch the module you designate as the default module.

There are two methods to register command handlers in Mynewt 1.1:

- Method 1 (New in release 1.1): Define and register a set of commands for a module. This method allows grouping shell commands into namespaces. A package calls the `shell_register()` function to define a module and register the command handlers for the module.

Note: The `SHELL_MAX_MODULES` syscfg setting specifies the maximum number of modules that can be registered. You can increase this value if your application and the packages it includes register more than the default value.

- Method 2: Register a command handler without defining a module. A package calls the `shell_cmd_register()` function defined in Mynewt 1.0 to register a command handler. When a shell command is registered using this method, the command is automatically added to the `compat` module. The `compat` module supports backward compatibility for all the shell commands that are registered using the `shell_cmd_register()` function.

Notes:

- The `SHELL_COMPAT` syscfg setting must be set to 1 to enable backward compatibility support and the `shell_cmd_register()` function. Since Mynewt packages use method 2 to register shell commands and Mynewt plans to continue this support in future releases, you must keep the default setting value of 1.
- The `SHELL_MAX_COMPAT_COMMANDS` syscfg setting specifies the maximum number of command handlers that can be registered using this method. You can increase this value if your application and the packages it includes register more than the default value.

Enabling Help Information for Shell Commands

The shell supports command help. A package that supports command help initializes the `struct shell_cmd` data structure with help text for the command before it registers the command with the shell. The `SHELL_CMD_HELP` syscfg setting enables or disables help support for all shell commands. The feature is enabled by default.

Note: A package that implements help for a shell command should only initialize the help data structures within the `#if MYNEWT_VAL(SHELL_CMD_HELP)` preprocessor directive.

Enabling the OS and Prompt Shell Modules

The shell implements the `os` and `prompt` modules. These modules support the shell commands to view OS resources.

The `os` module implements commands to list task and mempool usage information and to view and change the time of day. The `SHELL_OS_MODULE` syscfg setting enables or disables the module. The module is enabled by default.

The `prompt` module implements the `ticks` command that controls whether to print the current os ticks in the prompt. The `SHELL_PROMPT_MODULE` syscfg setting enables or disables this module. The module is disabled by default.

Enabling Command Name Completion

The shell supports command name completion. The SHELL_COMPLETION syscfg setting enables or disables the feature. The feature is enabled by default.

Processing Newtmgr Line Protocol Over Serial Transport

The shell's second job is to handle packet framing, encoding, and decoding of newtmgr protocol messages that are sent over the console. The Newtmgr serial transport package (`mgmt/newtmgr/transport/newtmgr_shell`) calls the `shell_nlip_input_register()` function to register a handler that the shell calls when it receives newtmgr request messages.

The SHELL_NEWTMGR syscfg setting specifies whether newtmgr is enabled over shell. The setting is enabled by default.

Data Structures

The `struct shell_cmd` data structure represents a shell command and is used to register a command.

```
struct shell_cmd {
    const char *sc_cmd;
    shell_cmd_func_t sc_cmd_func;
    const struct shell_cmd_help *help;
};
```

Element	Description
<code>sc_cmd</code>	Character string of the command name.
<code>sc_cmd_f</code>	Pointer to the command handler that processes the command.
<code>unc_t</code>	
<code>help</code>	Pointer to the <code>shell_cmd_he lp</code> structure. If the pointer is NULL, help information is not provided.

The `sc_cmd_func_t` is the command handler function type.

```
typedef int (*shell_cmd_func_t)(int argc, char *argv[]);
```

The `argc` parameter specifies the number of command line arguments and the `argv` parameter is an array of character pointers to the command arguments. The SHELL_CMD_ARGC_MAX syscfg setting specifies the maximum number of command line arguments that any shell command can have. This value must be increased if a shell command requires more than SHELL_CMD_ARGC_MAX number of command line arguments.

The `struct shell_module` data structure represents a shell module. It is used to register a shell module and the shell commands for the module.

```
struct shell_module {
    const char *name;
    const struct shell_cmd *commands;
};
```

Element	Description
<code>name</code>	Character string of the module name.
<code>commands</code>	Array of <code>shell_cmd</code> structures that specify the commands for the module. The <code>sc_cmd</code> , <code>sc_cmd_func</code> , and <code>help</code> fields in the last entry must be set to NULL to indicate the last entry in the array.

Note: A command handler registered via the `shell_cmd_register()` function is automatically added to the `compat` module.

The `struct shell_param` and `struct shell_cmd_help` data structures hold help texts for a shell command.

```
struct shell_param {
    const char *param_name;
    const char *help;
};
```

Element	Description
<code>param_name</code>	Character string of the command parameter name.
<code>help</code>	Character string of the help text for the parameter.

```
struct shell_cmd_help {
    const char *summary;
    const char *usage;
    const struct shell_param *params;
};
```

Element	Description
<code>summary</code>	Character string of a short description of the command.
<code>usage</code>	Character string of a usage description for the command.
<code>parameters</code>	Array of <code>shell_param</code> structures that describe each parameter for the command. The last <code>struct shell_param</code> in the array must have the <code>param_name</code> and <code>help</code> fields set to NULL to indicate the last entry in the array.

API

Defines

`SHELL_NMGR_OP_EXEC`

Command IDs in the “shell” newtmgr group.

`SHELL_HELP_(help_)`

`SHELL_CMD(cmd_, func_, help_)`

constructs a legacy shell command.

SHELL_CMD_EXT(cmd_, func_, help_)
constructs an extended shell command.

Typedefs

typedef int (***shell_cmd_func_t**)(int argc, char *argv[])
Callback called when command is entered.

Param argc

Number of parameters passed.

Param argv

Array of option strings. First option is always command name.

Return

0 in case of success or negative value in case of error.

typedef int (***shell_cmd_ext_func_t**)(const struct *shell_cmd* *cmd, int argc, char *argv[], struct streamer *streamer)
Callback for “extended” shell commands.

Param cmd

The shell command being executed.

Param argc

Number of arguments passed.

Param argv

Array of option strings. First option is always command name.

Param streamer

The streamer to write shell output to.

Return

0 on success; SYS_E[...] on failure.

typedef const char *(***shell_prompt_function_t**)(void)

Callback to get the current prompt.

Return

Current prompt string.

typedef int (***shell_nlip_input_func_t**)(struct *os_mbuf**, void *arg)

Functions

int **shell_register**(const char *shell_name, const struct *shell_cmd* *shell_commands)
Register a *shell_module* object.

Parameters

- **shell_name** – Module name to be entered in shell console.
- **shell_commands** – Array of commands to register. The array should be terminated with an empty element.

```
void shell_register_app_cmd_handler(shell_cmd_func_t handler)
```

Optionally register an app default cmd handler.

Parameters

- **handler** – To be called if no cmd found in cmds registered with shell_init.

```
void shell_register_prompt_handler(shell_prompt_function_t handler)
```

Optionally register a custom prompt callback.

Parameters

- **handler** – To be called to get the current prompt.

```
void shell_register_default_module(const char *name)
```

Optionally register a default module, to avoid typing it in shell console.

Parameters

- **name** – Module name.

```
void shell_evq_set(struct os_eventq *evq)
```

Optionally set event queue to process shell command events.

Parameters

- **evq** – Event queue to be used in shell

```
int shell_exec(int argc, char **argv, struct streamer *streamer)
```

Processes a set of arguments and executes their corresponding shell command.

Parameters

- **argc** – The argument count (including command name).
- **argv** – The argument list ([0] is command name).
- **streamer** – The streamer to send output to.

Returns

0 on success; SYS_E[...] on failure.

```
int shell_nlip_input_register(shell_nlip_input_func_t nf, void *arg)
```

```
int shell_nlip_output(struct os_mbuf *m)
```

```
int shell_cmd_register(const struct shell_cmd *sc)
```

```
struct shell_param
```

#include <shell.h>

Public Members

```
const char *param_name
```

```
const char *help
```

```
struct shell_cmd_help
```

#include <shell.h>

Public Members

```
const char *summary  
  
const char *usage  
  
const struct shell_param *params  
  
struct shell_cmd  
#include <shell.h>
```

Public Members

```
uint8_t sc_ext  
  
union shell_cmd  
  
const char *sc_cmd  
  
const struct shell_cmd_help *help  
  
struct shell_module  
#include <shell.h>
```

Public Members

```
const char *name  
  
const struct shell_cmd *commands
```

6.3 Hardware Abstraction Layer

6.3.1 Description

The Hardware Abstraction Layer (HAL) in Mynewt is a low-level, base peripheral abstraction. HAL provides a core set of services that is implemented for each MCU supported by Mynewt. Device drivers are typically the software libraries that initialize the hardware and manage access to the hardware by higher layers of software. In the Mynewt OS, the layers can be depicted in the following manner.



(continued from previous page)



- The Board Support Package (BSP) abstracts board specific configurations e.g. CPU frequency, input voltage, LED pins, on-chip flash map etc.
- The Hardware Abstraction Layer (HAL) abstracts architecture-specific functionality. It initializes and enables components within a master processor. It is designed to be portable across all the various MCUs supported in Mynewt (e.g. Nordic's nRF51, Nordic's nRF52, NXP's MK64F12 etc.). It includes code that initializes and manages access to components of the board such as board buses (I2C, PCI, PCMCIA, etc.), off-chip memory (controllers, level 2+ cache, Flash, etc.), and off-chip I/O (Ethernet, RS-232, display, mouse, etc.)
- The driver sits atop the BSP and HAL. It abstracts the common modes of operation for each peripheral device connected via the standard interfaces to the processor. There may be multiple driver implementations of differing complexities for a particular peripheral device. The drivers are the ones that register with the kernel's power management APIs, and manage turning on and off peripherals and external chipsets, etc.

6.3.2 General design principles

- The HAL API should be simple. It should be as easy to implement for hardware as possible. A simple HAL API makes it easy to bring up new MCUs quickly.
- The HAL API should be portable across all the various MCUs supported in Mynewt (e.g. Nordic's nRF51, Nordic's nRF52, NXP's MK64F12 etc.).

6.3.3 Example

A Mynewt contributor might write a light-switch driver that provides the functionality of an intelligent light switch. This might involve using a timer, a General Purpose Output (GPO) to set the light to the on or off state, and flash memory to log the times the lights were turned on or off. The contributor would like this package to work with as many different hardware platforms as possible, but can't possibly test across the complete set of hardware supported by Mynewt.

Solution: The contributor uses the HAL APIs to control the peripherals. The Mynewt team ensures that the underlying HAL devices all work equivalently through the HAL APIs. The contributors library is independent of the specifics of the hardware.

6.3.4 Dependency

To include the HAL within your project, simply add it to your package dependencies as follows:

```
pkg.deps:
  .
  .
  hw/hal
```

6.3.5 Platform Support

Not all platforms (MCU and BSP) support all HAL devices. Consult your MCU or BSP documentation to find out if you have hardware support for the peripherals you are interested in using. Once you verify support, then consult the MCU implementation and see if the specific HAL interface (xxxx) you are using is in the `mcu/<mcu-name>/src/hal_xxxx.c` implementation. Finally, you can build your project and ensure that there are no unresolved `hal_xxxx` externals.

6.3.6 Timer

The hardware independent timer structure and API to configure, initialize, and run timers.

Description

The HAL timer structure is shown below. The user can declare as many of these structures as required. They are enqueued on a particular HW timer queue when the user calls the `hal_timer_start` or `hal_timer_start_at` API. The user must have called `hal_timer_set_cb` before starting a timer.

NOTE: the user should not have to modify/examine the contents of this structure; the hal timer API should be used.

```
struct hal_timer
{
    void          *bsp_timer; /* Internal platform specific pointer */
    hal_timer_cb   cb_func;   /* Callback function */
    void          *cb_arg;   /* Callback argument */
    uint32_t       expiry;   /* Tick at which timer should expire */
    TAILQ_ENTRY(hal_timer) link; /* Queue linked list structure */
};
```

API

`typedef void (*hal_timer_cb)(void *arg)`

`int hal_timer_init(int timer_num, void *cfg)`

Initialize a HW timer.

Parameters

- `timer_num` – The number of the HW timer to initialize
- `cfg` – Hardware specific timer configuration. This is passed from BSP directly to the MCU specific driver.

`int hal_timer_deinit(int timer_num)`

Un-initialize a HW timer.

Parameters

- `timer_num` – The number of the HW timer to un-initialize

`int hal_timer_config(int timer_num, uint32_t freq_hz)`

Config a HW timer at the given frequency and start it.

If the exact frequency is not obtainable the closest obtainable frequency is set.

Parameters

- **timer_num** – The number of the HW timer to configure
- **freq_hz** – The frequency in Hz to configure the timer at

Returns

0 on success, non-zero error code on failure

```
uint32_t hal_timer_get_resolution(int timer_num)
```

Returns the resolution of the HW timer.

NOTE: the frequency may not be obtainable so the caller can use this to determine the resolution. Returns resolution in nanoseconds. A return value of 0 indicates an invalid timer was used.

Parameters

- **timer_num** – The number of the HW timer to get resolution for

Returns

The resolution of the timer

```
uint32_t hal_timer_read(int timer_num)
```

Returns the HW timer current tick value.

Parameters

- **timer_num** – The HW timer to read the tick value from

Returns

The current tick value

```
int hal_timer_delay(int timer_num, uint32_t ticks)
```

Perform a blocking delay for a number of ticks.

Parameters

- **timer_num** – The timer number to use for the blocking delay
- **ticks** – The number of ticks to delay for

Returns

0 on success, non-zero error code on failure

```
int hal_timer_set_cb(int timer_num, struct hal_timer *tmr, hal_timer_cb cb_func, void *arg)
```

Set the timer structure prior to use.

Should not be called if the timer is running. Must be called at least once prior to using timer.

Parameters

- **timer_num** – The number of the HW timer to configure the callback on
- **tmr** – The timer structure to use for this timer
- **cb_func** – The timer callback to call when the timer fires
- **arg** – An opaque argument to provide the timer callback

Returns

0 on success, non-zero error code on failure.

```
int hal_timer_start(struct hal_timer *tmr, uint32_t ticks)
```

Start a timer that will expire in ‘ticks’ ticks.

Ticks cannot be 0

Parameters

- **tmr** – The timer to start
- **ticks** – The number of ticks to expire the timer in

Returns

0 on success, non-zero error code on failure.

int hal_timer_start_at(struct hal_timer *tmr, uint32_t tick)

Start a timer that will expire when the timer reaches ‘tick’.

If tick has already passed the timer callback will be called “immediately” (at interrupt context).

Parameters

- **tmr** – The timer to start
- **tick** – The absolute tick value to fire the timer at

Returns

0 on success, non-zero error code on failure.

int hal_timer_stop(struct hal_timer *tmr)

Stop a currently running timer; associated callback will NOT be called.

Parameters

- **tmr** – The timer to stop

struct hal_timer

#include <hal_timer.h> The HAL timer structure.

The user can declare as many of these structures as desired. They are enqueued on a particular HW timer queue when the user calls the :c:func:[hal_timer_start\(\)](#) or :c:func:[hal_timer_start_at\(\)](#) API. The user must have called :c:func:[hal_timer_set_cb\(\)](#) before starting a timer.

NOTE: the user should not have to modify/examine the contents of this structure; the hal timer API should be used.

Public Members

void *bsp_timer

Internal platform specific pointer.

hal_timer_cb cb_func

Callback function.

void *cb_arg

Callback argument.

uint32_t expiry

Tick at which timer should expire.

6.3.7 GPIO

This is the hardware independent GPIO (General Purpose Input Output) Interface for Mynewt.

Description

Contains the basic operations to set and read General Purpose Digital I/O Pins within a Mynewt system.

Individual GPIOs are referenced in the APIs as pins. However, in this interface the pins are virtual GPIO pins. The MCU header file maps these virtual pins to the physical GPIO ports and pins.

Typically, the BSP code may define named I/O pins in terms of these virtual pins to describe the devices attached to the physical pins.

Here's a brief example so you can get the gist of the translation.

Suppose my product uses the stm32F4xx processor. There already exists support for this processor within Mynewt. The processor has N ports (A,B,C..) of 16 GPIO pins per port. The MCU hal_gpio driver maps these to a set of virtual pins 0-N where port A maps to 0-15, Port B maps to 16-31, Port C maps to 32-47 and so on. The exact number of physical port (and virtual port pins) depends on the specific variant of the stm32F4xx.

So if I want to turn on port B pin 3, that would be virtual pin $1*16 + 3 = 19$. This translation is defined in the MCU implementation of [hal_gpio.c](#) for the stm32. Each MCU will typically have a different translation method depending on its GPIO architecture.

Now, when writing a BSP, it's common to give names to the relevant port pins that you are using. Thus, the BSP may define a mapping between a function and a virtual port pin in the bsp.h header file for the BSP. For example,

```
#define SYSTEM_LED          (37)
#define FLASH_SPI_CHIP_SELECT (3)
```

would map the system indicator LED to virtual pin 37 which on the stm32F4xx would be Port C pin 5 and the chip select line for the external SPI flash to virtual pin 3 which on the stm32F4xx is port A pin 3.

Said another way, in this specific system we get

```
SYSTEM_LED --> hal_gpio virtual pin 37 --> port C pin 5 on the stm32F4xx
```

API

enum **hal_gpio_mode_e**

The “mode” of the gpio.

The gpio is either an input, output, or it is “not connected” (the pin specified is not functioning as a gpio)

Values:

enumerator **HAL_GPIO_MODE_NC**

Not connected.

enumerator **HAL_GPIO_MODE_IN**

Input.

enumerator **HAL_GPIO_MODE_OUT**

Output.

enum **hal_gpio_pull**

Values:

enumerator **HAL_GPIO_PULL_NONE**

Pull-up/down not enabled.

enumerator **HAL_GPIO_PULL_UP**

Pull-up enabled.

enumerator **HAL_GPIO_PULL_DOWN**

Pull-down enabled.

enum **hal_gpio_irq_trigger**

Values:

enumerator **HAL_GPIO_TRIG_NONE**

enumerator **HAL_GPIO_TRIG_RISING**

IRQ occurs on rising edge.

enumerator **HAL_GPIO_TRIG_FALLING**

IRQ occurs on falling edge.

enumerator **HAL_GPIO_TRIG_BOTH**

IRQ occurs on either edge.

enumerator **HAL_GPIO_TRIG_LOW**

IRQ occurs when line is low.

enumerator **HAL_GPIO_TRIG_HIGH**

IRQ occurs when line is high.

typedef enum *hal_gpio_mode_e* **hal_gpio_mode_t**

typedef enum *hal_gpio_pull* **hal_gpio_pull_t**

typedef enum *hal_gpio_irq_trigger* **hal_gpio_irq_trig_t**

typedef void (***hal_gpio_irq_handler_t**)(void *arg)

```
int hal_gpio_init_in(int pin, hal_gpio_pull_t pull)
```

Initializes the specified pin as an input.

Parameters

- **pin** – Pin number to set as input
- **pull** – pull type

Returns

int 0: no error; -1 otherwise.

```
int hal_gpio_init_out(int pin, int val)
```

Initialize the specified pin as an output, setting the pin to the specified value.

Parameters

- **pin** – Pin number to set as output
- **val** – Value to set pin

Returns

int 0: no error; -1 otherwise.

```
int hal_gpio_deinit(int pin)
```

Deinitialize the specified pin to revert the previous initialization.

Parameters

- **pin** – Pin number to unset

Returns

int 0: no error; -1 otherwise.

```
void hal_gpio_write(int pin, int val)
```

Write a value (either high or low) to the specified pin.

Parameters

- **pin** – Pin to set
- **val** – Value to set pin (0:low 1:high)

```
int hal_gpio_read(int pin)
```

Reads the specified pin.

Parameters

- **pin** – Pin number to read

Returns

int 0: low, 1: high

```
int hal_gpio_toggle(int pin)
```

Toggles the specified pin.

Parameters

- **pin** – Pin number to toggle

Returns

current gpio state int 0: low, 1: high

```
int hal_gpio_irq_init(int pin, hal_gpio_irq_handler_t handler, void *arg, hal_gpio_irq_trig_t trig,  
                      hal_gpio_pull_t pull)
```

Initialize a given pin to trigger a GPIO IRQ callback.

Parameters

- **pin** – The pin to trigger GPIO interrupt on
- **handler** – The handler function to call
- **arg** – The argument to provide to the IRQ handler
- **trig** – The trigger mode (e.g. rising, falling)
- **pull** – The mode of the pin (e.g. pullup, pulldown)

Returns

0 on success, non-zero error code on failure.

```
void hal_gpio_irq_release(int pin)
```

Release a pin from being configured to trigger IRQ on state change.

Parameters

- **pin** – The pin to release

```
void hal_gpio_irq_enable(int pin)
```

Enable IRQs on the passed pin.

Parameters

- **pin** – The pin to enable IRQs on

```
void hal_gpio_irq_disable(int pin)
```

Disable IRQs on the passed pin.

Parameters

- **pin** – The pin to disable IRQs on

6.3.8 UART

The hardware independent UART interface for Mynewt.

Description

Contains the basic operations to send and receive data over a UART (Universal Asynchronous Receiver Transmitter). It also includes the API to apply settings such as speed, parity etc. to the UART. The UART port should be closed before any reconfiguring.

Examples

This example shows a user writing a character to the uart in blocking mode where the UART has to block until character has been sent.

```
/* write to the console with blocking */
{
    char *str = "Hello World!";
    char *ptr = str;

    while(*ptr) {
        hal_uart_blocking_tx(MY_UART, *ptr++);
    }
    hal_uart_blocking_tx(MY_UART, '\n');
}
```

API

enum **hal_uart_parity**

Values:

enumerator **HAL_UART_PARITY_NONE**

No Parity.

enumerator **HAL_UART_PARITY_ODD**

Odd parity.

enumerator **HAL_UART_PARITY_EVEN**

Even parity.

enum **hal_uart_flow_ctl**

Values:

enumerator **HAL_UART_FLOW_CTL_NONE**

No Flow Control.

enumerator **HAL_UART_FLOW_CTL_RTS_CTS**

RTS/CTS.

typedef int (***hal_uart_tx_char**)(void *arg)

Function prototype for UART driver to ask for more data to send.

Returns -1 if no more data is available for TX. Driver must call this with interrupts disabled.

typedef void (***hal_uart_tx_done**)(void *arg)

Function prototype for UART driver to report that transmission is complete.

This should be called when transmission of last byte is finished. Driver must call this with interrupts disabled.

```
typedef int (*hal_uart_rx_char)(void *arg, uint8_t byte)
```

Function prototype for UART driver to report incoming byte of data.

Returns -1 if data was dropped. Driver must call this with interrupts disabled.

```
int hal_uart_init_cbs(int uart, hal_uart_tx_char tx_func, hal_uart_tx_done tx_done, hal_uart_rx_char rx_func,  
void *arg)
```

Initializes given uart.

Mapping of logical UART number to physical UART/GPIO pins is in BSP.

```
int hal_uart_init(int uart, void *cfg)
```

Initialize the HAL uart.

Parameters

- **uart** – The uart number to configure
- **cfg** – Hardware specific uart configuration. This is passed from BSP directly to the MCU specific driver.

Returns

0 on success, non-zero error code on failure

```
int hal_uart_config(int uart, int32_t speed, uint8_t databits, uint8_t stopbits, enum hal_uart_parity parity, enum  
hal_uart_flow_ctl flow_ctl)
```

Applies given configuration to UART.

Parameters

- **uart** – The UART number to configure
- **speed** – The baudrate in bps to configure
- **databits** – The number of databits to send per byte
- **stopbits** – The number of stop bits to send
- **parity** – The UART parity
- **flow_ctl** – Flow control settings on the UART

Returns

0 on success, non-zero error code on failure

```
int hal_uart_close(int uart)
```

Close UART port.

Can call *hal_uart_config()* with different settings after calling this.

Parameters

- **uart** – The UART number to close

```
void hal_uart_start_tx(int uart)
```

More data queued for transmission.

UART driver will start asking for that data.

Parameters

- **uart** – The UART number to start TX on

```
void hal_uart_start_rx(int uart)
```

Upper layers have consumed some data, and are now ready to receive more.

This is meaningful after uart_rx_char callback has returned -1 telling that no more data can be accepted.

Parameters

- **uart** – The UART number to begin RX on

```
void hal_uart_blocking_tx(int uart, uint8_t byte)
```

This is type of write where UART has to block until character has been sent.

Used when printing diag output from system crash. Must be called with interrupts disabled.

Parameters

- **uart** – The UART number to TX on
- **byte** – The byte to TX on the UART

6.3.9 SPI

SPI (Serial Peripheral Interface) is a synchronous 4-wire serial interface commonly used to connect components in embedded systems.

For a detailed description of SPI, see [Wikipedia](#).

Description

The Mynewt HAL interface supports the SPI master functionality with both blocking and non-blocking interface. SPI slave functionality is supported in non-blocking mode.

Theory Of Operation

SPI is called a 4-wire interface because of the 4 signals, MISO, MOSI, CLK, and SS. The SS signal (slave select) is an active low signal that activates a SPI slave device. This is how a master “addresses” a particular slave device. Often this signal is also referred to as “chip select” as it selects particular slave device for communications.

The Mynewt SPI HAL has blocking and non-blocking transfers. Blocking means that the API call to transfer a byte will wait until the byte completes transmissions before the function returns. Blocking interface can be used for only the master slave SPI type. Non-blocking means he function returns control to the execution environment immediately after the API call and a callback function is executed at the completion of the transmission. Non-blocking interface can be used for both master and slave SPI types.

The hal_spi_config method in the API above allows the SPI to be configured with appropriate settings for master or slave. It Must be called after the spi is initialized (i.e. after hal_spi_init is called) and when the spi is disabled (i.e. user must call hal_spi_disable if the spi has been enabled through hal_spi_enable prior to calling this function). It can also be used to reconfigure an initialized SPI (assuming it is disabled as described previously).

```
int hal_spi_config(int spi_num, struct hal_spi_settings *psettings);
```

The SPI settings consist of the following:

```
struct hal_spi_settings {
    uint8_t        data_mode;
    uint8_t        data_order;
```

(continues on next page)

(continued from previous page)

```
    uint8_t          word_size;
    uint32_t         baudrate;           /* baudrate in kHz */
};


```

The Mynewt SPI HAL does not include built-in SS (Slave Select) signaling. It's up to the hal_spi user to control their own SS pins. Typically applications will do this with GPIO.

API

typedef void (*hal_spi_txrx_cb)(void *arg, int len)

int **hal_spi_init**(int spi_num, void *cfg, uint8_t spi_type)

Initialize the SPI, given by spi_num.

Parameters

- **spi_num** – The number of the SPI to initialize
- **cfg** – HW/MCU specific configuration, passed to the underlying implementation, providing extra configuration.
- **spi_type** – SPI type (master or slave)

Returns

int 0 on success, non-zero error code on failure.

int **hal_spi_init_hw**(uint8_t spi_num, uint8_t spi_type, const struct *hal_spi_hw_settings* *cfg)

Initialize SPI controller.

This initializes SPI controller hardware before 1st use. Shall be called only once.

Parameters

- **spi_num** – Number of SPI controller
- **cfg** – Configuration

Returns

0 on success, non-zero error code on failure

int **hal_spi_config**(int spi_num, struct *hal_spi_settings* *psettings)

Configure the spi.

Must be called after the spi is initialized (after hal_spi_init is called) and when the spi is disabled (user must call hal_spi_disable if the spi has been enabled through hal_spi_enable prior to calling this function). Can also be used to reconfigure an initialized SPI (assuming it is disabled as described previously).

Parameters

- **spi_num** – The number of the SPI to configure.
- **psettings** – The settings to configure this SPI with

Returns

int 0 on success, non-zero error code on failure.

`int hal_spi_set_txrx_cb(int spi_num, hal_spi_txrx_cb txrx_cb, void *arg)`

Sets the txrx callback (executed at interrupt context) when the buffer is transferred by the master or the slave using the non-blocking API.

Cannot be called when the spi is enabled. This callback will also be called when chip select is de-asserted on the slave.

NOTE: This callback is only used for the non-blocking interface and must be called prior to using the non-blocking API.

Parameters

- **spi_num** – SPI interface on which to set callback
- **txrx** – Callback function
- **arg** – Argument to be passed to callback function

Returns

`int 0` on success, non-zero error code on failure.

`int hal_spi_enable(int spi_num)`

Enables the SPI.

This does not start a transmit or receive operation; it is used for power mgmt. Cannot be called when a SPI transfer is in progress.

Parameters

- **spi_num**

Returns

`int 0` on success, non-zero error code on failure.

`int hal_spi_disable(int spi_num)`

Disables the SPI.

Used for power mgmt. It will halt any current SPI transfers in progress.

Parameters

- **spi_num**

Returns

`int 0` on success, non-zero error code on failure.

`uint16_t hal_spi_tx_val(int spi_num, uint16_t val)`

Blocking call to send a value on the SPI.

Returns the value received from the SPI slave.

MASTER: Sends the value and returns the received value from the slave. SLAVE: Invalid API. Returns 0xFFFF

Parameters

- **spi_num** – Spi interface to use
- **val** – Value to send

Returns

`uint16_t` Value received on SPI interface from slave. Returns 0xFFFF if called when the SPI is configured to be a slave

```
int hal_spi_txrx(int spi_num, void *txbuf, void *rdbuf, int cnt)
```

Blocking interface to send a buffer and store the received values from the slave.

The transmit and receive buffers are either arrays of 8-bit (`uint8_t`) values or 16-bit values depending on whether the spi is configured for 8 bit data or more than 8 bits per value. The ‘cnt’ parameter is the number of 8-bit or 16-bit values. Thus, if ‘cnt’ is 10, txbuf/rdbuf would point to an array of size 10 (in bytes) if the SPI is using 8-bit data; otherwise txbuf/rdbuf would point to an array of size 20 bytes (ten, `uint16_t` values).

NOTE: these buffers are in the native endian-ness of the platform.

```
MASTER: master sends all the values in the buffer and stores the  
stores the values in the receive buffer if rdbuf is not NULL.  
The txbuf parameter cannot be NULL.
```

```
SLAVE: cannot be called for a slave; returns -1
```

Parameters

- `spi_num` – SPI interface to use
- `txbuf` – Pointer to buffer where values to transmit are stored.
- `rdbuf` – Pointer to buffer to store values received from peer.
- `cnt` – Number of 8-bit or 16-bit values to be transferred.

Returns

int 0 on success, non-zero error code on failure.

```
int hal_spi_txrx_noblock(int spi_num, void *txbuf, void *rdbuf, int cnt)
```

Non-blocking interface to send a buffer and store received values.

Can be used for both master and slave SPI types. The user must configure the callback (using `hal_spi_set_txrx_cb`); the txrx callback is executed at interrupt context when the buffer is sent.

The transmit and receive buffers are either arrays of 8-bit (`uint8_t`) values or 16-bit values depending on whether the spi is configured for 8 bit data or more than 8 bits per value. The ‘cnt’ parameter is the number of 8-bit or 16-bit values. Thus, if ‘cnt’ is 10, txbuf/rdbuf would point to an array of size 10 (in bytes) if the SPI is using 8-bit data; otherwise txbuf/rdbuf would point to an array of size 20 bytes (ten, `uint16_t` values).

NOTE: these buffers are in the native endian-ness of the platform.

```
MASTER: master sends all the values in the buffer and stores the  
stores the values in the receive buffer if rdbuf is not NULL.  
The txbuf parameter cannot be NULL
```

```
SLAVE: Slave "preloads" the data to be sent to the master (values  
stored in txbuf) and places received data from master in rdbuf  
(if not NULL). The txrx callback occurs when len values are  
transferred or master de-asserts chip select. If txbuf is NULL,  
the slave transfers its default byte. Both rdbuf and txbuf cannot  
be NULL.
```

Parameters

- `spi_num` – SPI interface to use
- `txbuf` – Pointer to buffer where values to transmit are stored.
- `rdbuf` – Pointer to buffer to store values received from peer.
- `cnt` – Number of 8-bit or 16-bit values to be transferred.

Returns

int 0 on success, non-zero error code on failure.

int **hal_spi_slave_set_def_tx_val**(int spi_num, uint16_t val)

Sets the default value transferred by the slave.

Not valid for master

Parameters

- **spi_num** – SPI interface to use

Returns

int 0 on success, non-zero error code on failure.

int **hal_spi_abort**(int spi_num)

This aborts the current transfer but keeps the spi enabled.

NOTE: does not return an error if no transfer was in progress.

Parameters

- **spi_num** – SPI interface on which transfer should be aborted.

Returns

int 0 on success, non-zero error code on failure.

int **hal_spi_data_mode_breakout**(uint8_t data_mode, int *out_cpol, int *out_cpha)

Extracts CPOL and CPHA values from a data-mode constant.

Utility function, defined once for every MCU.

Parameters

- **data_mode** – The HAL_SPI_MODE value to convert.
- **out_cpol** – The CPOL gets written here on success.
- **out_cpha** – The CPHA gets written here on success.

Returns

0 on success; nonzero on invalid input.

HAL_SPI_TYPE_MASTER

HAL_SPI_TYPE_SLAVE

HAL_SPI_MODE0

SPI mode 0.

HAL_SPI_MODE1

SPI mode 1.

HAL_SPI_MODE2

SPI mode 2.

HAL_SPI_MODE3

SPI mode 3.

HAL_SPI_MSB_FIRST

SPI data order, most significant bit first.

HAL_SPI_LSB_FIRST

SPI data order, least significant bit first.

HAL_SPI_WORD_SIZE_8BIT

SPI word size 8 bit.

HAL_SPI_WORD_SIZE_9BIT

SPI word size 9 bit.

struct **hal_spi_hw_settings**

#include <hal_spi.h> SPI controller hardware settings.

struct **hal_spi_settings**

#include <hal_spi.h> since one spi device can control multiple devices, some configuration can be changed on the fly from the hal

Public Members

uint8_t **data_mode**

Data mode of SPI driver, defined by HAL_SPI_MODEn.

uint8_t **data_order**

Data order, either HAL_SPI_MSB_FIRST or HAL_SPI_LSB_FIRST.

uint8_t **word_size**

The word size of the SPI transaction, either 8-bit or 9-bit.

uint32_t **baudrate**

Baudrate in kHz.

6.3.10 I2C

The hardware independent interface to I2C Devices.

Description

An Inter-Integrated Circuit (I²C] I-squared-C) bus is a multi-master, multi-save serial interface used to connect components on a circuit board and often peripherals devices located off the circuit board.

I2C is often thought of as a 2-wire protocol because it uses two wires (SDA, SCL) to send data between devices.

For a detailed description of I²C, see the [wikipedia page](#)

HAL_I2C Theory Of Operation

An I²C transaction typically involves acquiring the bus, sending and/or receiving data and release the bus. The bus acquisition portion is important because the bus is typically multi-master so other devices may be trying to read/write the same peripheral.

HAL_I2C implements a master interface to the I²C bus. Typical usage of the interface would involve the following steps.

Initialize an i2c device with: hal_i2c_init()

When you wish to perform an i2c transaction, you call one or both of: hal_i2c_master_write(); hal_i2c_master_read();

These functions will issue a START condition, followed by the device's 7-bit I2C address, and then send or receive the payload based on the data provided. This will cause a repeated start on the bus, which is valid in I2C specification, and the decision to use repeated starts was made to simplify the I2C HAL. To set the STOP condition at an appropriate moment, you set the `last_op` field to a 1 in either function.

For example, in an I2C memory access you might write a register address and then read data back via: hal_i2c_write(); — write to a specific register on the device hal_i2c_read(); — read back data, setting 'last_op' to '1'

An addition API was added called hal_i2c_probe. This command combines hal_i2c_begin(), hal_i2c_read, and hal_i2c_end() to try to read 0-bytes from a specific bus address. its intended to provide an easy way to probe the bus for a specific device. NOTE: if the device is write-only, it will not appear with this command.

A slave API is pending for further release.

HAL_I2C Data

Data to read/write is passed to the hal_i2c APIs via the

```
struct hal_i2c_master_data {
    uint8_t address; /* destination address */
    uint16_t len;     /* number of bytes to transmit or receive */
    uint8_t *buffer;  /* data buffer for transmit or receive */
};
```

`buffer` is a pointer to the data to send. `len` is the number of bytes to send over the bus. `address` is a 7-bit bus address of the device.

When I²C builds its address, it uses the 7-bit address plus a 1-bit R/W (read/write) indicator to identify the device and direction of the transaction. Thus when using this API, you should use a 7-bit address in the data structure and ensure that address is a value between 0-127.

As an example, consider an I²C device address that looks like this:

B7	B6	B5	B4	B3	B2	B1	B0
1	0	0	0	1	1	0	R/W
MSB						LSB	

In the HAL_I2C API you would communicate with this device with address **0b1000110**, which is hex 0x46 or decimal 70. The I²C drive would add the R/W bit and transmit it as hex 0x8C (binary 10001100) or 0x8D (binary 10001101) depending whether it was a read or write command.

API

`int hal_i2c_init(uint8_t i2c_num, void *cfg)`

Initialize a new i2c device with the I2C number.

Parameters

- **i2c_num** – The number of the I2C device being initialized
- **cfg** – The hardware specific configuration structure to configure the I2C with. This includes things like pin configuration.

Returns

0 on success, and non-zero error code on failure

`int hal_i2c_init_hw(uint8_t i2c_num, const struct hal_i2c_hw_settings *cfg)`

Initialize I2C controller.

This initializes I2C controller hardware before 1st use. Shall be called only once.

Parameters

- **i2c_num** – Number of I2C controller
- **cfg** – Configuration

Returns

0 on success, non-zero error code on failure

`int hal_i2c_enable(uint8_t i2c_num)`

Enable I2C controller.

This enables I2C controller before usage.

Parameters

- **i2c_num** – Number of I2C controller

Returns

0 on success, non-zero error code on failure

`int hal_i2c_disable(uint8_t i2c_num)`

Disable I2C controller.

This disabled I2C controller if no longer needed. Hardware configuration be preserved after controller is disabled.

Parameters

- **i2c_num** – Number of I2C controller

Returns

0 on success, non-zero error code on failure

```
int hal_i2c_config(uint8_t i2c_num, const struct hal_i2c_settings *cfg)
```

Configure I2C controller.

This configures I2C controller for operation. Can be called multiple times.

Parameters

- **i2c_num** – Number of I2C controller
- **cfg** – Configuration

```
int hal_i2c_master_write(uint8_t i2c_num, struct hal_i2c_master_data *pdata, uint32_t timeout, uint8_t last_op)
```

Sends a start condition and writes <len> bytes of data on the i2c bus.

This API does NOT issue a stop condition unless **last_op** is set to 1. You must stop the bus after successful or unsuccessful write attempts. This API is blocking until an error or NaK occurs. Timeout is platform dependent.

Parameters

- **i2c_num** – The number of the I2C device being written to
- **pdata** – The data to write to the I2C bus
- **timeout** – How long to wait for transaction to complete in ticks
- **last_op** – Master should send a STOP at the end to signify end of transaction.

Returns

0 on success, and non-zero error code on failure

```
int hal_i2c_master_read(uint8_t i2c_num, struct hal_i2c_master_data *pdata, uint32_t timeout, uint8_t last_op)
```

Sends a start condition and reads <len> bytes of data on the i2c bus.

This API does NOT issue a stop condition unless **last_op** is set to 1. You must stop the bus after successful or unsuccessful write attempts. This API is blocking until an error or NaK occurs. Timeout is platform dependent.

Parameters

- **i2c_num** – The number of the I2C device being written to
- **pdata** – The location to place read data
- **timeout** – How long to wait for transaction to complete in ticks
- **last_op** – Master should send a STOP at the end to signify end of transaction.

Returns

0 on success, and non-zero error code on failure

```
int hal_i2c_master_probe(uint8_t i2c_num, uint8_t address, uint32_t timeout)
```

Probes the i2c bus for a device with this address.

THIS API issues a start condition, probes the address using a read command and issues a stop condition.

Parameters

- **i2c_num** – The number of the I2C to probe
- **address** – The address to probe for
- **timeout** – How long to wait for transaction to complete in ticks

Returns

0 on success, non-zero error code on failure

HAL_I2C_ERR_UNKNOWN

This is the API for an i2c bus.

Currently, this is a master API allowing the mynewt device to function as an I2C master.

A slave API is pending for future release

Typical usage of this API is as follows:

Initialize an i2c device with: `:c:func:hal_i2c_init()`

When you wish to perform an i2c transaction, you call one or both of: `:c:func:hal_i2c_master_write()`; `:c:func:hal_i2c_master_read()`;

These functions will issue a START condition, followed by the device's 7-bit I2C address, and then send or receive the payload based on the data provided. This will cause a repeated start on the bus, which is valid in I2C specification, and the decision to use repeated starts was made to simplify the I2C HAL. To set the STOP condition at an appropriate moment, you set the `last_op` field to a 1 in either function.

For example, in an I2C memory access you might write a register address and then read data back via: `:c:func:hal_i2c_write();` write to a specific register on the device `:c:func:hal_i2c_read();` `—` read back data, setting '`last_op`' to '1' Unknown error.

HAL_I2C_ERR_INVAL

Invalid argument.

HAL_I2C_ERR_TIMEOUT

MCU failed to report result of I2C operation.

HAL_I2C_ERR_ADDR_NACK

Slave responded to address with NACK.

HAL_I2C_ERR_DATA_NACK

Slave responded to data byte with NACK.

struct hal_i2c_hw_settings

`#include <hal_i2c.h>` I2C controller hardware settings.

struct hal_i2c_settings

`#include <hal_i2c.h>` I2C configuration.

Public Members

uint32_t frequency

Frequency in kHz.

struct hal_i2c_master_data

`#include <hal_i2c.h>` When sending a packet, use this structure to pass the arguments.

Public Members

uint8_t address

Destination address An I2C address has 7 bits.

In the protocol these 7 bits are combined with a 1 bit R/W bit to specify read or write operation in an 8-bit address field sent to the remote device. This API accepts the 7-bit address as its argument in the 7 LSBs of the address field above. For example if I2C was writing a 0x81 in its protocol, you would pass only the top 7-bits to this function as 0x40

uint16_t len

Number of buffer bytes to transmit or receive.

uint8_t *buffer

Buffer space to hold the transmit or receive.

6.3.11 Flash

The hardware independent interface to flash memory that is used by applications.

hal_flash_int

The API that flash drivers have to implement.

Description

The BSP for the hardware will implement the structs defined in this API.

Definition

hal_flash_int.h

Examples

The Nordic nRF52 bsp implements the hal_flash_int API as seen in [hal_bsp.c](#)

```
const struct hal_flash *
hal_bsp_flash_dev(uint8_t id)
{
    /*
     * Internal flash mapped to id 0.
     */
    if (id != 0) {
        return NULL;
    }
    return &nrf52k_flash_dev;
}
```

Description

The API offers basic initialization, read, write, erase, sector erase, and other operations.

API

```
int hal_flash_ioctl(uint8_t flash_id, uint32_t cmd, void *args)  
int hal_flash_sector_info(uint8_t flash_id, int sector_index, uint32_t *start_address, uint32_t *size)  
    Return information about flash sector.
```

Parameters

- **flash_id** – The ID of the flash device to read from.
- **sector_index** – The sector number to get information about.
- **start_address** – A buffer to fill with start address of the sector.
- **size** – A buffer for sector size.

Returns

0 on success; SYS_EINVAL on bad argument error; SYS_EIO on flash driver error.

```
int hal_flash_read(uint8_t flash_id, uint32_t address, void *dst, uint32_t num_bytes)
```

Reads a block of data from flash.

Parameters

- **flash_id** – The ID of the flash device to read from.
- **address** – The address to read from.
- **dst** – A buffer to fill with data read from flash.
- **num_bytes** – The number of bytes to read.

Returns

0 on success; SYS_EINVAL on bad argument error; SYS_EIO on flash driver error.

```
int hal_flash_write(uint8_t flash_id, uint32_t address, const void *src, uint32_t num_bytes)
```

Writes a block of data to flash.

Parameters

- **flash_id** – The ID of the flash device to write to.
- **address** – The address to write to.
- **src** – A buffer containing the data to be written.
- **num_bytes** – The number of bytes to write.

Returns

0 on success; SYS_EINVAL on bad argument error; SYS_EACCES if flash region is write protected; SYS_EIO on flash driver error.

```
int hal_flash_erase_sector(uint8_t flash_id, uint32_t sector_address)
```

Erases a single flash sector.

Parameters

- **flash_id** – The ID of the flash device to erase.
- **sector_address** – An address within the sector to erase.

Returns

0 on success; SYS_EINVAL on bad argument error; SYS_EACCES if flash region is write protected; SYS_EIO on flash driver error.

```
int hal_flash_erase(uint8_t flash_id, uint32_t address, uint32_t num_bytes)
```

Erases a contiguous sequence of flash sectors.

If the specified range does not correspond to a whole number of sectors, any partially-specified sectors are fully erased. For example, if a device has 1024-byte sectors, then these arguments: o address: 300 o num_bytes: 1000 cause the first two sectors to be erased in their entirety.

Parameters

- **flash_id** – The ID of the flash device to erase.
- **address** – An address within the sector to begin the erase at.
- **num_bytes** – The length, in bytes, of the region to erase.

Returns

0 on success; SYS_EINVAL on bad argument error; SYS_EACCES if flash region is write protected; SYS_EIO on flash driver error.

```
int hal_flash_isempty(uint8_t flash_id, uint32_t address, void *dst, uint32_t num_bytes)
```

Determines if the specified region of flash is completely unwritten.

Parameters

- **id** – The ID of the flash hardware to inspect.
- **address** – The starting address of the check.
- **dst** – A buffer to hold the contents of the flash region. This must be at least **num_bytes** large.
- **num_bytes** – The number of bytes to check.

Returns

0 if any written bytes were detected; 1 if the region is completely unwritten; SYS_EINVAL on bad argument error; SYS_EIO on flash driver error.

```
int hal_flash_isempty_no_buf(uint8_t id, uint32_t address, uint32_t num_bytes)
```

Determines if the specified region of flash is completely unwritten.

This function is like [hal_flash_isempty\(\)](#), except the caller does not need to provide a buffer. Instead, a buffer of size MYNEWT_VAL(HAL_FLASH_VERIFY_BUF_SZ) is allocated on the stack.

Parameters

- **id** – The ID of the flash hardware to inspect.
- **address** – The starting address of the check.
- **num_bytes** – The number of bytes of flash to check.

Returns

0 if any written bytes were detected; 1 if the region is completely unwritten; SYS_EINVAL on bad argument error; SYS_EIO on flash driver error.

```
uint8_t hal_flash_align(uint8_t flash_id)
```

Determines the minimum write alignment of a flash device.

Parameters

- **id** – The ID of the flash hardware to check.

Returns

The flash device's minimum write alignment.

`uint8_t hal_flash_erased_val(uint8_t flash_id)`

Determines the value of an erased byte for a particular flash device.

Parameters

- **id** – The ID of the flash hardware to check.

Returns

The value of an erased byte.

`int hal_flash_init(void)`

Initializes all flash devices in the system.

Returns

0 on success; SYS_EIO on flash driver error.

`int hal_flash_write_protect(uint8_t id, uint8_t protect)`

Set or clears write protection.

This function allows to disable write to the device if for some reason (i.e. low power state) writes could result in data corruption.

Parameters

- **id** – The ID of the flash
- **protect** – 1 - disable writes 0 - enable writes

Returns

SYS_EINVAL - if flash id is not valid SYS_OK - on success

6.3.12 System

A hardware independent interface for starting and resetting the system.

Description

The API allows the user to detect whether a debugger is connected, issue a soft reset, and enumerate the reset causes. The functions are implemented in the MCU specific directories e.g. `hal_reset_cause.c`, `hal_system.c`, and `hal_system_start.c` in `/hw/mcu/nordic/nrf52xxx/src/` directory for Nordic nRF52 series of chips.

API

enum `hal_reset_reason`

Reboot reason.

Values:

enumerator `HAL_RESET_POR`

Power on Reset.

enumerator **HAL_RESET_PIN**

Caused by Reset Pin.

enumerator **HAL_RESET_WATCHDOG**

Caused by Watchdog.

enumerator **HAL_RESET_SOFT**

Soft reset, either system reset or crash.

enumerator **HAL_RESET_BROWNOUT**

Low supply voltage.

enumerator **HAL_RESET_REQUESTED**

Restart due to user request.

enumerator **HAL_RESET_SYS_OFF_INT**

System Off, wakeup on external interrupt.

enumerator **HAL_RESET_DFU**

Restart due to DFU.

enumerator **HAL_RESET_OTHER**

Restart reason other.

void hal_system_reset (void) __attribute__((noreturn))

System reset.

void hal_system_start (void *img_start) __attribute__((noreturn))

Called by bootloader to start loaded program.

void hal_system_restart (void *img_start) __attribute__((noreturn))

Called by split app loader to start the app program.

int hal_debugger_connected(void)

Returns non-zero if there is a HW debugger attached.

enum *hal_reset_reason* hal_reset_cause(void)

Return the reboot reason.

Returns

A reboot reason

const char *hal_reset_cause_str(void)

Return the reboot reason as a string.

Returns

String describing previous reset reason

void hal_system_clock_start(void)

Starts clocks needed by system.

```
void hal_system_reset_cb(void)
    Reset callback to be called before an reset happens inside hal_system_reset()
```

6.3.13 Watchdog

The hardware independent interface to enable internal hardware watchdogs.

Description

The `hal_watchdog_init` interface can be used to set a recurring watchdog timer to fire no sooner than in ‘`expire_secs`’ seconds.

```
int hal_watchdog_init(uint32_t expire_msecs);
```

Watchdog needs to be then started with a call to `hal_watchdog_enable()`. Watchdog should be tickled periodically with a frequency smaller than ‘`expire_secs`’ using `hal_watchdog_tickle()`.

API

```
int hal_watchdog_init(uint32_t expire_msecs)
```

Set a recurring watchdog timer to fire no sooner than in ‘`expire_secs`’ seconds.

Watchdog should be tickled periodically with a frequency smaller than ‘`expire_secs`’. Watchdog needs to be then started with a call to :c:func:`hal_watchdog_enable()`.

Parameters

- **expire_msecs** – Watchdog timer expiration time in msec

Returns

< 0 on failure; on success return the actual expiration time as positive value

```
void hal_watchdog_enable(void)
```

Starts the watchdog.

```
void hal_watchdog_tickle(void)
```

Tickles the watchdog.

This needs to be done periodically, before the value configured in :c:func:`hal_watchdog_init()` expires.

6.3.14 BSP

This is the hardware independent BSP (Board Support Package) Interface for Mynewt.

Description

Contains the basic operations to initialize, specify memory to include in coredump, configure interrupt priority etc.

API

`void hal_bsp_init(void)`

Initializes BSP; registers flash_map with the system.

`void hal_bsp_deinit(void)`

De-initializes BSP.

Intended to be called from bootloader before it call application reset handler. It should leave resources (timers/DMA/peripherals) in a state that nothing unexpected is active before application code is ready to handle it.

`const struct hal_flash *hal_bsp_flash_dev(uint8_t flash_id)`

`void *_sbrk(int incr)`

`const struct hal_bsp_mem_dump *hal_bsp_core_dump(int *area_cnt)`

Report which memory areas should be included inside a coredump.

`int hal_bsp_hw_id_len(void)`

Retrieves the length, in bytes, of the hardware ID.

Returns

The length of the hardware ID.

`int hal_bsp_hw_id(uint8_t *id, int max_len)`

Get unique HW identifier/serial number for platform.

Returns the number of bytes filled in.

Parameters

- **id** – Pointer to the ID to fill out
- **max_len** – Maximum length to fill into the ID

Returns

0 on success, non-zero error code on failure

`int hal_bsp_power_state(int state)`

Move the system into the specified power state.

Parameters

- **state** – The power state to move the system into, this is one of
 - HAL_BSP_POWER_ON: Full system on
 - HAL_BSP_POWER_WFI: Processor off, wait for interrupt.
 - HAL_BSP_POWER_SLEEP: Put the system to sleep
 - HAL_BSP_POWER_DEEP_SLEEP: Put the system into deep sleep.
 - HAL_BSP_POWER_OFF: Turn off the system.
 - HAL_BSP_POWER_PERUSER: From this value on, allow user defined power states.

Returns

0 on success, non-zero if system cannot move into this power state.

`uint32_t hal_bsp_get_nvic_priority(int irq_num, uint32_t pri)`

Returns priority of given interrupt number.

HAL_BSP_MAX_ID_LEN

HAL_BSP_POWER_ON

Full System On.

HAL_BSP_POWER_WFI

Wait for Interrupt: CPU off.

HAL_BSP_POWER_SLEEP

System sleep mode, processor off, some peripherals off too.

HAL_BSP_POWER_DEEP_SLEEP

Deep sleep: possible loss of RAM retention, system wakes up in undefined state.

HAL_BSP_POWER_OFF

System powering off.

HAL_BSP_POWER_PERUSER

Per-user power state, base number for user to define custom power states.

`struct hal_bsp_mem_dump`

`#include <hal_bsp.h>`

6.3.15 OS Tick

The hardware independent interface to set up interrupt timers or halt CPU in terms of OS ticks.

Description

Set up the periodic timer to interrupt at a frequency of ‘os_ticks_per_sec’ using the following function call where ‘prio’ is the cpu-specific priority of the periodic timer interrupt.

`void os_tick_init(uint32_t os_ticks_per_sec, int prio);`

You can halt CPU for up to n ticks:

`void os_tick_idle(os_time_t n);`

The function implementations are in the mcu-specific directories such as hw/mcu/nordic/nrf51xxx/src/hal_os_tick.c.

API

`void os_tick_init(uint32_t os_ticks_per_sec, int prio)`

Set up the periodic timer to interrupt at a frequency of ‘os_ticks_per_sec’.

‘prio’ is the cpu-specific priority of the periodic timer interrupt.

Parameters

- **os_ticks_per_sec** – Frequency of the OS tick timer
- **prio** – Priority of the OS tick timer

`void os_tick_idle(os_time_t n)`

Halt CPU for up to ‘n’ ticks.

Parameters

- **n** – The number of ticks to halt the CPU for

6.3.16 Creating New HAL Interfaces

HAL API

A HAL always includes header file with function declarations for the HAL functionality in `/hw/hal/include/hal`. The first argument of all functions in the interface typically include the virtual device_id of the device you are controlling.

For example, in `'hal_gpio.h <https://github.com/apache/incubator-mynewt-core/blob/master/hw/hal/include/hal/hal_gpio.h>'` the device enumeration is the first argument to most methods and called `pin`.

```
void hal_gpio_write(int pin, int val);
```

The device_id (in this case called `pin`) is not a physical device (actual hardware pin), but a virtual pin which is defined by the implementation of the HAL (and documented in the implementation of the HAL).

6.3.17 Using HAL in Your Libraries

This page describes the recommended way to implement libraries that utilize HAL functionality.

An example of the GPIO HAL being used by a driver for a UART bitbanger that programs the start bit, data bits, and stop bit can be seen in `hw/drivers/uart/uart_bitbang/src/uart_bitbang.c`

An example of the flash HAL being used by a file system can be seen in `fs/nffs/src/nffs_flash.c`.

6.4 Bootloader

The “bootloader” is the code that loads the Mynewt OS image into memory and conducts some checks before allowing the OS to be run. It manages images for the embedded system and upgrades of those images using protocols over various interfaces (e.g. serial, BLE, etc.). Typically, systems with bootloaders have at least two program images coexisting on the same microcontroller, and hence must include branch code that performs a check to see if an attempt to update software is already underway and manage the progress of the process.

The bootloader in the Apache Mynewt project verifies the cryptographic signature of the firmware image before running it. It maintains a detailed status log for each stage of the boot process.

The “secure bootloader” should be placed in protected memory on a given microcontroller.

The Apache Mynewt bootloader is the foundation of MCUboot, a secure bootloader for 32-bit MCUs that has been ported to other Operating Systems as well.

- [*Limitations*](#)
- [*Image Format*](#)
- [*Flash Map*](#)
- [*Image Slots*](#)
- [*Boot States*](#)
- [*Boot Vector*](#)
- [*High-level Operation*](#)
- [*Image Swapping*](#)
- [*Swap Status*](#)
- [*Reset Recovery*](#)
- [*Integrity Check*](#)
- [*Image Signing and Verification*](#)

The Mynewt bootloader comprises two packages:

- The bootutil library (boot/bootutil)
- The boot application (apps/boot)

The Mynewt code is thus structured so that the generic bootutil library performs most of the functions of a boot loader. The final step of actually jumping to the main image is kept out of the bootutil library. This last step should instead be implemented in an architecture-specific project. Boot loader functionality is separated in this manner for the following two reasons:

1. By keeping architecture-dependent code separate, the bootutil library can be reused among several boot loaders.
2. By excluding the last boot step from the library, the bootloader can be unit tested since a library can be unit tested but an applicant can't.

6.4.1 Limitations

The boot loader currently only supports images with the following characteristics:

- Built to run from flash.
- Build to run from a fixed location (i.e., position-independent).

6.4.2 Image Format

The following definitions describe the image header format.

```
#define IMAGE_MAGIC          0x96f3b83c
#define IMAGE_MAGIC_NONE      0xffffffffffff

struct image_version {
    uint8_t iv_major;
    uint8_t iv_minor;
    uint16_t iv_revision;
    uint32_t iv_build_num;
};

/** Image header. All fields are in little endian byte order. */
struct image_header {
    uint32_t ih_magic;
    uint16_t ih_tlv_size; /* Trailing TLVs */
    uint8_t ih_key_id;
    uint8_t _pad1;
    uint16_t ih_hdr_size;
    uint16_t _pad2;
    uint32_t ih_img_size; /* Does not include header. */
    uint32_t ih_flags;
    struct image_version ih_ver;
    uint32_t _pad3;
};
```

The `ih_hdr_size` field indicates the length of the header, and therefore the offset of the image itself. This field provides for backwards compatibility in case of changes to the format of the image header.

The following are the image header flags available.

```
#define IMAGE_F_PIC           0x00000001
#define IMAGE_F_SHA256          0x00000002 /* Image contains hash TLV */
#define IMAGE_F_PKCS15_RSA2048_SHA256 0x00000004 /* PKCS15 w/RSA and SHA */
#define IMAGE_F_ECDSA224_SHA256   0x00000008 /* ECDSA256 over SHA256 */
#define IMAGE_F_NON_BOOTABLE     0x00000010
#define IMAGE_HEADER_SIZE        32
```

Optional type-length-value records (TLVs) containing image metadata are placed after the end of the image. For example, security data gets added as a footer at the end of the image.

```
/** Image trailer TLV format. All fields in little endian. */
struct image_tlv {
    uint8_t it_type; /* IMAGE_TLV_[...]. */
    uint8_t _pad;
    uint16_t it_len; /* Data length (not including TLV header). */
};

/*
 * Image trailer TLV types.
 */
#define IMAGE_TLV_SHA256          1 /* SHA256 of image hdr and body */
```

(continues on next page)

(continued from previous page)

<code>#define IMAGE_TLV_RSA2048</code>	2 /* RSA2048 of hash output */
<code>#define IMAGE_TLV_ECDSA224</code>	3 /* ECDSA of hash output */

6.4.3 Flash Map

A Mynewt device's flash is partitioned according to its *flash map*. At a high level, the flash map maps numeric IDs to *flash areas*. A flash area is a region of disk with the following properties:

1. An area can be fully erased without affecting any other areas.
2. A write to one area does not restrict writes to other areas.

The boot loader uses the following flash areas:

<code>#define FLASH_AREA_BOOTLOADER</code>	0
<code>#define FLASH_AREA_IMAGE_0</code>	1
<code>#define FLASH_AREA_IMAGE_1</code>	2
<code>#define FLASH_AREA_IMAGE_SCRATCH</code>	3

6.4.4 Image Slots

A portion of the flash memory is partitioned into two image slots: a primary slot and a secondary slot. The boot loader will only run an image from the primary slot, so images must be built such that they can run from that fixed location in flash. If the boot loader needs to run the image resident in the secondary slot, it must swap the two images in flash prior to booting.

In addition to the two image slots, the boot loader requires a scratch area to allow for reliable image swapping.

6.4.5 Boot States

Logically, you can think of a pair of flags associated with each image slot: pending and confirmed. On startup, the boot loader determines the state of the device by inspecting each pair of flags. These flags have the following meanings:

- pending: image gets tested on next reboot; absent subsequent confirm command, revert to original image on second reboot.
- confirmed: always use image unless excluded by a test image.

In English, when the user wants to run the secondary image, they set the pending flag for the second slot and reboot the device. On startup, the boot loader will swap the two images in flash, clear the secondary slot's pending flag, and run the newly-copied image in slot 0. This is a temporary state; if the device reboots again, the boot loader swaps the images back to their original slots and boots into the original image. If the user doesn't want to revert to the original state, they can make the current state permanent by setting the confirmed flag in slot 0.

Switching to an alternate image is a two-step process (set + confirm) to prevent a device from becoming “bricked” by bad firmware. If the device crashes immediately upon booting the second image, the boot loader reverts to the working image, rather than repeatedly rebooting into the bad image.

The following set of tables illustrate the three possible states that the device can be in:

	slot-0	slot-1	
pending			

(continues on next page)

(continued from previous page)

confirmed X		
	- - - - -	- - - - -
Image 0 confirmed;		
No change on reboot		
	- - - - -	- - - - -
	slot-0 slot-1	
	- - - - -	- - - - -
pending	X	
confirmed X		
	- - - - -	- - - - -
Image 0 confirmed;		
Test image 1 on next reboot		
	- - - - -	- - - - -
	slot-0 slot-1	
	- - - - -	- - - - -
pending		
confirmed X		
	- - - - -	- - - - -
Testing image 0;		
Revert to image 1 on next reboot		
	- - - - -	- - - - -

6.4.6 Boot Vector

At startup, the boot loader determines which of the above three boot states a device is in by inspecting the boot vector. The boot vector consists of two records (called “image trailers”), one written at the end of each image slot. An image trailer has the following structure:

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
+ +			
~	MAGIC (16 octets)		~
+ +			
~			~
~	Swap status (128 * min-write-size * 3)		~
~			~
+ +			
Copy done 0xff padding (up to min-write-sz - 1)			~
+ +			
Image OK 0xff padding (up to min-write-sz - 1)			~
+ +			

These records are at the end of each image slot. The offset immediately following such a record represents the start of the next flash area.

Note: `min-write-size` is a property of the flash hardware. If the hardware allows individual bytes to be written at arbitrary addresses, then `min-write-size` is 1. If the hardware only allows writes at even addresses, then `min-write-size` is 2, and so on.

The fields are defined as follows:

1. MAGIC: The following 16 bytes, written in host-byte-order:

```
const uint32_t boot_img_magic[4] = {
    0xf395c277,
    0x7fefd260,
    0x0f505235,
    0x8079b62c,
};
```

2. Swap status: A series of single-byte records. Each record corresponds to a flash sector in an image slot. A swap status byte indicate the location of the corresponding sector data. During an image swap, image data is moved one sector at a time. The swap status is necessary for resuming a swap operation if the device rebooted before a swap operation completed.
3. Copy done: A single byte indicating whether the image in this slot is complete (0x01=done, 0xff=not done).
4. Image OK: A single byte indicating whether the image in this slot has been confirmed as good by the user (0x01=confirmed; 0xff=not confirmed).

The boot vector records are structured around the limitations imposed by flash hardware. As a consequence, they do not have a very intuitive design, and it is difficult to get a sense of the state of the device just by looking at the boot vector. It is better to map all the possible vector states to the swap types (None, Test, Revert) via a set of tables. These tables are reproduced below. In these tables, the “pending” and “confirmed” flags are shown for illustrative purposes; they are not actually present in the boot vector.

State I

	slot-0	slot-1	
magic	Unset	Unset	
image-ok	Any	N/A	
<hr/>			
pending			
confirmed	X		
<hr/>			
swap: none			
<hr/>			

State II

	slot-0	slot-1	
magic	Any	Good	
image-ok	Any	N/A	
<hr/>			
pending		X	
confirmed	X		
<hr/>			
swap: test			
<hr/>			

State III

	slot-0	slot-1	
magic	Good	Unset	

(continues on next page)

(continued from previous page)

image-ok	0xff	N/A	
pending			
confirmed		X	
swap: revert (test image running)			

State IV

	slot-0	slot-1	
magic	Good	Unset	
image-ok	0x01	N/A	
pending			
confirmed	X		
swap: none (confirmed test image)			

6.4.7 High-level Operation

With the terms defined, we can now explore the boot loader's operation. First, a high-level overview of the boot process is presented. Then, the following sections describe each step of the process in more detail.

Procedure:

- A. Inspect swap status region; is an interrupted swap is being resumed?
 Yes: Complete the partial swap operation; skip to step C.
 No: Proceed to step B.
- B. Inspect boot vector; is a swap requested?
 Yes.
 - 1. Is the requested image valid (integrity and security check)?
 Yes.
 - a. Perform swap operation.
 - b. Persist completion of swap procedure to boot vector.
 - c. Proceed to step C.
 - No.
 - a. Erase invalid image.
 - b. Persist failure of swap procedure to boot vector.
 - c. Proceed to step C.
 No: Proceed to step C.
- C. Boot into image in slot 0.

6.4.8 Image Swapping

The boot loader swaps the contents of the two image slots for two reasons:

- User has issued an “image test” operation; the image in slot-1 should be run once (state II).
- Test image rebooted without being confirmed; the boot loader should revert to the original image currently in slot-1 (state III).

If the boot vector indicates that the image in the secondary slot should be run, the boot loader needs to copy it to the primary slot. The image currently in the primary slot also needs to be retained in flash so that it can be used later. Furthermore, both images need to be recoverable if the boot loader resets in the middle of the swap operation. The two images are swapped according to the following procedure:

1. Determine how many flash sectors each image slot consists of. This number must be the same for both slots.
2. Iterate the list of sector indices in descending order (i.e., starting with the greatest index); current element = "index".
 - a. Erase scratch area.
 - b. Copy slot1[index] to scratch area.
 - c. Write updated swap status (i).
 - d. Erase slot1[index]
 - e. Copy slot0[index] to slot1[index]
 - If these are the last sectors (i.e., first swap being performed), copy the full sector *except* the image trailer.
 - Else, copy entire sector contents.
 - f. Write updated swap status (ii).
 - g. Erase slot0[index].
 - h. Copy scratch area slot0[index].
 - i. Write updated swap status (iii).
3. Persist completion of swap procedure to slot 0 image trailer.

The additional caveats in step 2f are necessary so that the slot 1 image trailer can be written by the user at a later time. With the image trailer unwritten, the user can test the image in slot 1 (i.e., transition to state II).

The particulars of step 3 vary depending on whether an image is being tested or reverted:

```
* test:  
    o Write slot0.copy_done = 1  
      (should now be in state III)  
  
* revert:  
    o Write slot0.magic = BOOT_MAGIC  
    o Write slot0.copy_done = 1  
    o Write slot0.image_ok = 1  
      (should now be in state IV)
```

6.4.9 Swap Status

The swap status region allows the boot loader to recover in case it restarts in the middle of an image swap operation. The swap status region consists of a series of single-byte records. These records are written independently, and therefore must be padded according to the minimum write size imposed by the flash hardware. In the below figure, a `min-write-size` of 1 is assumed for simplicity. The structure of the swap status region is illustrated below. In this figure, a `min-write-size` of 1 is assumed for simplicity.

```

0          1          2          3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+
|sec127,state 0 |sec127,state 1 |sec127,state 2 |sec126,state 0 |
+-----+-----+-----+-----+
|sec126,state 1 |sec126,state 2 |sec125,state 0 |sec125,state 1 |
+-----+-----+-----+-----+
|sec125,state 2 |
+-----+
~           ~           ~           ~
~           [Records for indices 124 through 1
~           ~           ~           ~
~           +-----+
|sec000,state 0 |sec000,state 1 |sec000,state 2 |
+-----+

```

And now, in English...

Each image slot is partitioned into a sequence of flash sectors. If we were to enumerate the sectors in a single slot, starting at 0, we would have a list of sector indices. Since there are two image slots, each sector index would correspond to a pair of sectors. For example, sector index 0 corresponds to the first sector in slot 0 and the first sector in slot 1. Furthermore, we impose a limit of 128 indices. If an image slot consists of more than 128 sectors, the flash layout is not compatible with this boot loader. Finally, reverse the list of indices such that the list starts with index 127 and ends with 0. The swap status region is a representation of this reversed list.

During a swap operation, each sector index transitions through four separate states:

- 0. slot 0: image 0, slot 1: image 1, scratch: N/A
- 1. slot 0: image 0, slot 1: N/A, scratch: image 1 (1->s, erase 1)
- 2. slot 0: N/A, slot 1: image 0, scratch: image 1 (0->1, erase 0)
- 3. slot 0: image 1, slot 1: image 0, scratch: N/A (s->0)

Each time a sector index transitions to a new state, the boot loader writes a record to the swap status region. Logically, the boot loader only needs one record per sector index to keep track of the current swap state. However, due to limitations imposed by flash hardware, a record cannot be overwritten when an index's state changes. To solve this problem, the boot loader uses three records per sector index rather than just one.

Each sector-state pair is represented as a set of three records. The record values map to the above four states as follows

	rec0	rec1	rec2
state 0	0xff	0xff	0xff
state 1	0x01	0xff	0xff
state 2	0x01	0x02	0xff
state 3	0x01	0x02	0x03

The swap status region can accommodate 128 sector indices. Hence, the size of the region, in bytes, is `128 * min-write-size * 3`. The number 128 is chosen somewhat arbitrarily and will likely be made configurable. The

only requirement for the index count is that it is great enough to account for a maximum-sized image (i.e., at least as great as the total sector count in an image slot). If a device’s image slots use less than 128 sectors, the first record that gets written will be somewhere in the middle of the region. For example, if a slot uses 64 sectors, the first sector index that gets swapped is 63, which corresponds to the exact halfway point within the region.

6.4.10 Reset Recovery

If the boot loader resets in the middle of a swap operation, the two images may be discontiguous in flash. Bootutil recovers from this condition by using the boot vector to determine how the image parts are distributed in flash.

The first step is determine where the relevant swap status region is located. Because this region is embedded within the image slots, its location in flash changes during a swap operation. The below set of tables map boot vector contents to swap status location. In these tables, the “source” field indicates where the swap status region is located.

	slot-0	scratch	
magic	Good	Any	
copy-done	0x01	N/A	
source:	none		
	slot-0	scratch	
magic	Good	Any	
copy-done	0xff	N/A	
source:	slot 0		
	slot-0	scratch	
magic	Any	Good	
copy-done	Any	N/A	
source:	scratch		
	slot-0	scratch	
magic	Unset	Any	
copy-done	0xff	N/A	
source:	varies		
This represents one of two cases:			
o No swaps ever (no status to read, so no harm in checking).			
o Mid-revert; status in slot 0.			

If the swap status region indicates that the images are not contiguous, bootutil completes the swap operation that was in progress when the system was reset. In other words, it applies the procedure defined in the previous section, moving image 1 into slot 0 and image 0 into slot 1. If the boot status file indicates that an image part is present in the scratch

area, this part is copied into the correct location by starting at step e or step h in the area-swap procedure, depending on whether the part belongs to image 0 or image 1.

After the swap operation has been completed, the boot loader proceeds as though it had just been started.

6.4.11 Integrity Check

An image is checked for integrity immediately before it gets copied into the primary slot. If the boot loader doesn't perform an image swap, then it doesn't perform an integrity check.

During the integrity check, the boot loader verifies the following aspects of an image:

- 32-bit magic number must be correct (0x96f3b83c).
- Image must contain a SHA256 TLV.
- Calculated SHA256 must match SHA256 TLV contents.
- Image *may* contain a signature TLV. If it does, its contents must be verifiable using a key embedded in the boot loader.

6.4.12 Image Signing and Verification

As indicated above, the final step of the integrity check is signature verification. The boot loader can have one or more public keys embedded in it at build time. During signature verification, the boot loader verifies that an image was signed with a private key that corresponds to one of its public keys. The image signature TLV indicates the index of the key that is has been signed with. The boot loader uses this index to identify the corresponding public key.

For information on embedding public keys in the boot loader, as well as producing signed images, see [here](#).

6.5 Split Images

6.5.1 Description

The split image mechanism divides a target into two separate images: one capable of image upgrade; the other containing application code. By isolating upgrade functionality to a separate image, the application can support over-the-air upgrade without dedicating flash space to network stack and management code.

6.5.2 Concept

Mynewt supports three image setups:

Setup	Description
Single	One large image; upgrade not supported.
Unified	Two standalone images.
Split	Kernel in slot 0; application in slot 1.

Each setup has its tradeoffs. The Single setup gives you the most flash space, but doesn't allow you to upgrade after manufacturing. The Unified setup allows for a complete failover in case a bad image gets uploaded, but requires a lot of redundancy in each image, limiting the amount of flash available to the application. The Split setup sits somewhere between these two options.

Before exploring the split setup in more detail, it might be helpful to get a basic understanding of the Mynewt boot sequence. The boot process is summarized below.

Boot Sequence - Single

In the Single setup, there is no boot loader. Instead, the image is placed at address 0. The hardware boots directly into the image code. Upgrade is not possible because there is no boot loader to move an alternate image into place.

Boot Sequence - Unified

In the Unified setup, the boot loader is placed at address 0. At startup, the boot loader arranges for the correct image to be in image slot 0, which may entail swapping the contents of the two image slots. Finally, the boot loader jumps to the image in slot 0.

Boot Sequence - Split

The Split setup differs from the other setups mainly in that a target is not fully contained in a single image. Rather, the target is partitioned among two separate images: the *loader*, and the *application*. Functionality is divided among these two images as follows:

1. Loader:

- Mynewt OS.
- Network stack for connectivity during upgrade e.g. BLE stack.
- Anything else required for image upgrade.

2. Application:

- Parts of Mynewt not required for image upgrade.
- Application-specific code.

The loader image serves three purposes:

1. *Second-stage boot loader*: it jumps into the application image at start up.
2. *Image upgrade server*: the user can upgrade to a new loader + application combo, even if an application image is not currently running.
3. *Functionality container*: the application image can directly access all the code present in the loader image

From the perspective of the boot loader, a loader image is identical to a plain unified image. What makes a loader image different is a change to its start up sequence: rather than starting the Mynewt OS, it jumps to the application image in slot 1 if one is present.

6.5.3 Tutorial

Building a Split Image

We will be referring to the Nordic PCA10028 (nRF51 DK) for examples in this document. Let's take a look at this board's flash map (defined in `hw/bsp/nrf51dk/bsp.yml`):

Name	Offset	Size (kB)
Boot loader	0x00000000	16
Reboot log	0x00004000	16
Image slot 0	0x00008000	110
Image slot 1	0x00023800	110
Image scratch	0x0003f000	2
Flash file system	0x0003f800	2

The application we will be building is `bleprph`. First, we create a target to tie our BSP and application together.

```
newt target create bleprph-nrf51dk
newt target set bleprph-nrf51dk \
    app=@apache-mynewt-core/apps/bleprph \
    bsp=@apache-mynewt-core/hw/bsp/nordic_pca10028 \
    build_profile=optimized \
    syscfg=BLE_LL_CFG_FEAT_LE_ENCRYPTION=0:BLE_SM_LEGACY=0
```

The two syscfg settings disable bluetooth security and keep the code size down.

We can verify the target using the `target show` command:

```
[~/tmp/myproj2]$ newt target show bleprph-nrf51dk
targets/bleprph-nrf51dk
    app=@apache-mynewt-core/apps/bleprph
    bsp=@apache-mynewt-core/hw/bsp/nordic_pca10028
    build_profile=optimized
    syscfg=BLE_LL_CFG_FEAT_LE_ENCRYPTION=0:BLE_SM_LEGACY=0
```

Next, build the target:

```
[~/tmp/myproj2]$ newt build bleprph-nrf51dk
Building target targets/bleprph-nrf51dk
# [...]
Target successfully built: targets/bleprph-nrf51dk
```

With our target built, we can view a code size breakdown using the `newt size <target>` command. In the interest of brevity, the smaller entries are excluded from the below output:

```
[~/tmp/myproj2]$ newt size bleprph-nrf51dk
Size of Application Image: app
FLASH      RAM
 2446     1533 apps_bleprph.a
 1430      104 boot_bootutil.a
 1232        0 crypto_mbedtls.a
 1107        0 encoding_cborattr.a
 2390        0 encoding_tinycbor.a
 1764        0 fs_fcb.a
 2959      697 hw_drivers_nimble_nrf51.a
 4126      108 hw_mcu_nordic_nrf51xxx.a
 8161     4049 kernel_os.a
 2254       38 libc_baselIBC.a
 2612        0 libgcc.a
 2232       24 mgmt_immgmgr.a
```

(continues on next page)

(continued from previous page)

```

1499    44 mgmt_newtmgm_nmgr_os.a
23918   1930 net_nimble_controller.a
28537   2779 net_nimble_host.a
2207    205 sys_config.a
1074    197 sys_console_full.a
3268    97 sys_log.a
1296    0 time_datetime.a

objsize
text      data      bss      dec      hex filename
105592   1176    13392   120160   1d560 /home/me/tmp/myproj2/bin/targets/bleprph-nrf51dk/
app/apps/bleprph/bleprph.elf

```

The full image text size is about 103kB (where 1kB = 1024 bytes). With an image slot size of 110kB, this leaves only about 7kB of flash for additional application code and data. Not good. This is the situation we would be facing if we were using the Unified setup.

The Split setup can go a long way in solving our problem. Our unified bleprph image consists mostly of components that get used during an image upgrade. By using the Split setup, we turn the unified image into two separate images: the loader and the application. The functionality related to image upgrade can be delegated to the loader image, freeing up a significant amount of flash in the application image slot.

Let's create a new target to use with the Split setup. We designate a target as a split target by setting the `loader` variable. In our example, we are going to use `bleprph` as the loader, and `splitty` as the application. `bleprph` makes sense as a loader because it contains the BLE stack and everything else required for an image upgrade.

```

newt target create split-nrf51dk
newt target set split-nrf51dk
  loader=@apache-mynewt-core/apps/bleprph           \
  app=@apache-mynewt-core/apps/splitty               \
  bsp=@apache-mynewt-core/hw/bsp/nordic_pca10028   \
  build_profile=optimized                           \
  syscfg=BLE_LL_CFG_FEAT_LE_ENCRYPTION=0:BLE_SM_LEGACY=0

```

Verify that the target looks correct:

```
[~/tmp/myproj2]$ newt target show split-nrf51dk
targets/split-nrf51dk
  app=@apache-mynewt-core/apps/splitty
  bsp=@apache-mynewt-core/hw/bsp/nordic_pca10028
  build_profile=optimized
  loader=@apache-mynewt-core/apps/bleprph
  syscfg=BLE_LL_CFG_FEAT_LE_ENCRYPTION=0:BLE_SM_LEGACY=0
```

Now, let's build the new target:

```
[~/tmp/myproj2]$ newt build split-nrf51dk
Building target targets/split-nrf51dk
# [...]
Target successfully built: targets/split-nrf51dk
```

And look at the size breakdown (again, smaller entries are removed):

```
[~/tmp/myproj2]$ newt size split-nrf51dk
Size of Application Image: app
  FLASH      RAM
  3064      251 sys_shell.a

objsize
  text      data      bss      dec      hex filename
  4680      112    17572    22364    575c /home/me/tmp/myproj2/bin/targets/split-nrf51dk/
  ↵app/apps/splitty/splitty.elf

Size of Loader Image: loader
  FLASH      RAM
  2446     1533 apps_bleprph.a
  1430      104 boot_bootutil.a
  1232        0 crypto_mbedtls.a
  1107        0 encoding_cborattr.a
  2390        0 encoding_tinycbor.a
  1764        0 fs_fcb.a
  3168      705 hw_drivers_nimble_nrf51.a
  4318      109 hw_mcu_nordic_nrf51xxx.a
  8285     4049 kernel_os.a
  2274       38 libc_baselIBC.a
  2612        0 libgcc.a
  2232       24 mgmt_imgmgr.a
  1491       44 mgmt_newtmgr_nmgr_os.a
  25169    1946 net_nimble_controller.a
  31397    2827 net_nimble_host.a
  2259       205 sys_config.a
  1318       202 sys_console_full.a
  3424       97 sys_log.a
  1053       60 sys_stats.a
  1296        0 time_datetime.a

objsize
  text      data      bss      dec      hex filename
  112020    1180   13460   126660   1eec4 /home/me/tmp/myproj2/bin/targets/split-nrf51dk/
  ↵loader/apps/bleprph/bleprph.elf
```

The size command shows two sets of output: one for the application, and another for the loader. The addition of the split functionality did make bleprph slightly bigger, but notice how small the application is: 4.5 kB! Where before we only had 7 kB left, now we have 105.5 kB. Furthermore, all the functionality in the loader is available to the application at any time. For example, if your application needs bluetooth functionality, it can use the BLE stack present in the loader instead of containing its own copy.

Finally, let's deploy the split image to our Nordic PCA10028 (nRF51 DK) board. The procedure here is the same as if we were using the Unified setup, i.e., via either the `newt load` or `newt run` command.

```
[~/repos/mynewt/core]$ newt load split-nrf51dk 0
Loading app image into slot 2
Loading loader image into slot 1
```

6.5.4 Image Management

Retrieve Current State (image list)

Image management in the split setup is a bit more complicated than in the unified setup. You can determine a device's image management state with the `newtmgr image list` command. Here is how a device responds to this command after our loader + application combo has been deployed:

```
[~/tmp/myproj2]$ newtmgr -c A600ANJ1 image list
Images:
slot=0
  version: 0.0.0
  bootable: true
  flags: active confirmed
  hash: 948f118966f7989628f8f3be28840fd23a200fc219bb72acdf9096f06c4b39b
slot=1
  version: 0.0.0
  bootable: false
  flags:
  hash: 78e4d263eeb5af5635705b7cae026cc184f14aa6c6c59c6e80616035cd2efc8f
Split status: matching
```

There are several interesting things about this response:

1. *Two images*: This is expected; we deployed both a loader image and an application image.
2. *bootable flag*: Notice slot 0's bootable flag is set, while slot 1's is not. This tells us that slot 0 contains a loader and slot 1 contains an application. If an image is bootable, it can be booted directly from the boot loader. Non-bootable images can only be started from a loader image.
3. *flags*: Slot 0 is **active** and **confirmed**; none of slot 1's flags are set. The **active** flag indicates that the image is currently running; the **confirmed** flag indicates that the image will continue to be used on subsequent reboots. Slot 1's lack of enabled flags indicates that the image is not being used at all.
4. *Split status*: The split status field tells you if the loader and application are compatible. A loader + application combo is compatible only if both images were built at the same time with `newt`. If the loader and application are not compatible, the loader will not boot into the application.

Enabling a Split Application

By default, the application image in slot 1 is disabled. This is indicated in the `image list` response above. When you deploy a loader / application combo to your device, the application image won't actually run. Instead, the loader will act as though an application image is not present and remain in "loader mode". Typically, a device in loader mode simply acts as an image management server, listening for an image upgrade or a request to activate the application image.

Use the following command sequence to enable the split application image:

1. Tell device to "test out" the application image on next boot (`newtmgr image test <application-image-hash>`).
2. Reboot device (`newtmgr reset`).
3. Make above change permanent (`newtmgr image confirm`).

After the above sequence, a `newtmgr image list` command elicits the following response:

```
[~/tmp/myproj2]$ newtmgr -c A600ANJ1 image confirm
Images:
slot=0
  version: 0.0.0
  bootable: true
  flags: active confirmed
  hash: 948f118966f7989628f8f3be28840fd23a200fc219bb72acdf9096f06c4b39b
slot=1
  version: 0.0.0
  bootable: false
  flags: active confirmed
  hash: 78e4d263eeb5af5635705b7cae026cc184f14aa6c6c59c6e80616035cd2efc8f
Split status: matching
```

The `active confirmed` flags value on both slots indicates that both images are permanently running.

Image Upgrade

First, let's review of the image upgrade process for the Unified setup. The user upgrades to a new image in this setup with the following steps:

Image Upgrade - Unified

1. Upload new image to slot 1 (`newtmgr image upload <filename>`).
2. Tell device to “test out” the new image on next boot (`newtmgr image test <image-hash>`).
3. Reboot device (`newtmgr reset`).
4. Make new image permanent (`newtmgr image confirm`).

Image Upgrade - Split

The image upgrade process is a bit more complicated in the Split setup. It is more complicated because two images need to be upgraded (loader and application) rather than just one. The split upgrade process is described below:

1. Disable split functionality; we need to deactivate the application image in slot 1 (`newtmgr image test <current-loader-hash>`).
2. Reboot device (`newtmgr reset`).
3. Make above change permanent (`newtmgr image confirm`).
4. Upload new loader to slot 1 (`newtmgr image upload <filename>`).
5. Tell device to “test out” the new loader on next boot (`newtmgr image test <new-loader-hash>`).
6. Reboot device (`newtmgr reset`).
7. Make above change of loader permanent (`newtmgr image confirm`).
8. Upload new application to slot 1 (`newtmgr image upload <filename>`).
9. Tell device to “test out” the new application on next boot (`newtmgr image test <new-application-hash>`).
10. Reboot device (`newtmgr reset`).
11. Make above change of application permanent (`newtmgr image confirm`).

When performing this process manually, it may be helpful to use `image list` to check the image management state as you go.

6.5.5 Syscfg

Syscfg is Mynewt's system-wide configuration mechanism. In a split setup, there is a single umbrella syscfg configuration that applies to both the loader and the application. Consequently, overriding a value in an application-only package potentially affects the loader (and vice-versa).

6.5.6 Loaders

The following applications have been enabled as loaders. You may choose to build your own loader application, and these can serve as samples.

- @apache-mynewt-core/apps/slinky
- @apache-mynewt-core/apps/bleprph

6.5.7 Split Apps

The following applications have been enabled as split applications. If you choose to build your own split application these can serve as samples. Note that slinky can be either a loader image or an application image.

- @apache-mynewt-core/apps/slinky
- @apache-mynewt-core/apps/splitty

6.5.8 Theory of Operation

A split image is built as follows:

First newt builds the application and loader images separately to ensure they are consistent (no errors) and to generate elf files which can inform newt of the symbols used by each part.

Then newt collects the symbols used by both application and loader in two ways. It collects the set of symbols from the .elf files. It also collects all the possible symbols from the .a files for each application.

Newt builds the set of packages that the two applications share. It ensures that all the symbols used in those packages are matching. NOTE: because of features and #ifdefs, its possible for the two package to have symbols that are not the same. In this case newt generates an error and will not build a split image.

Then newt creates the list of symbols that the two applications share from those packages (using the .elf files).

Newt re-links the loader to ensure all of these symbols are present in the loader application (by forcing the linker to include them in the .elf).

Newt builds a special copy of the loader.elf with only these symbols (and the handful of symbols discussed in the linking section above).

Finally, newt links the application, replacing the common .a libraries with the special loader.elf image during the link.

6.6 Porting Mynewt OS

This chapter describes how to adapt the Mynewt OS to different platforms.

- *Description*
- *Board Support Package (BSP) Dependency*
- *MCU Dependency*
- *MCU HAL*
- *CPU Core Dependency*

6.6.1 Description

The Mynewt OS is a complete multi-tasking environment with scheduler, time control, buffer management, and synchronization objects. it also includes libraries and services like console, command shell, image manager, bootloader, and file systems etc.

The majority of this software is platform independent and requires no intervention to run on your platform, but some of the components require support from the underlying platform.

The platform dependency of these components can fall into several categories:

- **CPU Core Dependencies** – Specific code or configuration to operate the CPU core within your target platform
- **MCU Dependencies** – Specific code or configuration to operate the MCU or SoC within your target platform
- **BSP Dependencies** – Specific code or configuration to accommodate the specific layout and functionality of your target platform

6.6.2 Board Support Package (BSP) Dependency

With all of the functionality provided by the core, MCU, and MCU HAL (Hardware Abstraction Layer), there are still some things that must be specified for your particular system. This is provided in Mynewt to allow you the flexibility to design for the exact functionality, peripherals and features that you require in your product.

In Mynewt, these settings/components are included in a Board Support Package (BSP). The BSP contains the information specific to running Mynewt on a target platform or hardware board. Mynewt supports some common open source hardware as well as the development boards for some common MCUs. These development systems might be enough for you to get your prototype up and running, but when building a product you are likely going to have your own board which is slightly different from those already supported by Mynewt.

For example, you might decide on your system that 16 Kilobytes of flash space in one flash device is reserved for a flash file system. Or on your system you may decide that GPIO pin 5 of the MCU is connected to the system LED. Or you may decide that the OS Tick (the underlying time source for the OS) should run slower than the defaults to conserve battery power. These types of behaviors are specified in the BSP.

The information provided in the BSP (what you need to specify to get a complete executable) can vary depending on the MCU and its underlying core architecture. For example, some MCUs have dedicated pins for UART, SPI etc, so there is no configuration required in the BSP when using these peripherals. However some MCUs have a pin multiplexor that allows the UART to be mapped to several different pins. For these MCUs, the BSP must specify if and where the UART pins should appear to match the hardware layout of your system.

- If your BSP is already supported by Mynewt, there is no additional BSP work involved in porting to your platform. You need only to set the `bsp` attribute in your Mynewt target using the [*newt command tool*](#).
- If your BSP is not yet supported by Mynewt, you can add support following the instructions on [*BSP Porting*](#).

6.6.3 MCU Dependency

Some OS code depends on the MCU or SoC that the system contains. For example, the MCU may specify the potential memory map of the system - where code and data can reside.

- If your MCU is already supported by Mynewt, there is no additional MCU work involved in porting to your platform. You need only to set the `arch` attribute in your Mynewt target using the [*newt command tool*](#).
- If your MCU is not yet supported by Mynewt, you can add support following the instructions in [*Porting Mynewt to a new MCU*](#).

6.6.4 MCU HAL

Mynewt's architecture supports a hardware abstraction layer (HAL) for common on or off-chip MCU peripherals such as GPIO, UARTs, flash memory etc. Even if your MCU is supported for the core OS, you may find that you need to implement the HAL functionality for a new peripheral. For a description of the HAL abstraction and implementation information, see the [*HAL API*](#)

6.6.5 CPU Core Dependency

Some OS code depends on the CPU core that your system is using. For example, a given CPU core has a specific assembly language instruction set, and may require special cross compiler or compiler settings to use the appropriate instruction set.

- If your CPU architecture is already supported by Mynewt, there is no CPU core work involved in porting to your platform. You need only to set the `arch` and `compiler` attributes in your Mynewt target using the [*newt command tool*](#).
- If your CPU architecture is not supported by Mynewt, you can add support following the instructions on [*Porting Mynewt to a new CPU Architecture*](#).

6.6.6 BSP Porting

- *Introduction*
- *Download the BSP package template*
- *Create a set of Mynewt targets*
- *Fill in the `bsp.yml` file*
- *Flash map*
- *Add the MCU dependency to `pkg.yml`*
- *Check the BSP linker scripts*
- *Copy the download and debug scripts*

- *Fill in BSP functions and defines*
- *Add startup code*
- *Satisfy MCU requirements*
- *Test it*
- *Appendix A: BSP files*

Introduction

The Apache Mynewt core repo contains support for several different boards. For each supported board, there is a Board Support Package (BSP) package in the `hw/bsp` directory. If there isn't a BSP package for your hardware, then you will need to make one yourself. This document describes the process of creating a BSP package from scratch.

While creating your BSP package, the following documents will probably come in handy:

- The datasheet for the MCU you have chosen.
- The schematic of your board.
- The information on the CPU core within your MCU if it is not included in your MCU documentation.

This document is applicable to any hardware, but it will often make reference to a specific board as an example. Our example BSP has the following properties:

- **Name:** `hw/bsp/myboard`
- **MCU:** Nordic nRF52

Download the BSP package template

We start by downloading a BSP package template. This template will serve as a good starting point for our new BSP.

Execute the `newt pkg new` command, as below:

```
$ newt pkg new -t bsp hw/bsp/myboard
Download package template for package type bsp.
Package successfully installed into /home/me/myproj/hw/bsp/myboard.
```

Our new package has the following file structure:

```
$ tree hw/bsp/myboard
hw/bsp/myboard
├── README.md
├── boot-myboard.1d
├── bsp.yml
├── include
│   └── myboard
│       └── bsp.h
├── myboard.1d
├── myboard_debug.sh
└── myboard_download.sh
    └── pkg.yml
    └── src
```

(continues on next page)

(continued from previous page)

<pre> └── hal_bsp.c └── sbrk.c syscfg.yml </pre> <p>3 directories, 11 files</p>

We will be adding to this package throughout the remainder of this document. See [Appendix A: BSP files](#) for a full list of files typically found in a BSP package.

Create a set of Mynewt targets

We'll need two *targets* to test our BSP as we go:

1. Boot loader
2. Application

A minimal application is best, since we are just interested in getting the BSP up and running. A good app for our purposes is [blinky](#).

We create our targets with the following set of newt commands:

```

newt target create boot-myboard &&
newt target set boot-myboard app=@mcuboot/boot/mynewt \
                           bsp=hw/bsp/myboard \
                           build_profile=optimized \
                           \
newt target create blinky-myboard &&
newt target set blinky-myboard app=apps/blinky \
                           bsp=hw/bsp/myboard \
                           build_profile=debug

```

Which generates the following output:

```

Target targets/boot-myboard successfully created
Target targets/boot-myboard successfully set target.app to @mcuboot/boot/mynewt
Target targets/boot-myboard successfully set target.bsp to hw/bsp/myboard
Target targets/boot-myboard successfully set target.build_profile to debug
Target targets/blinky-myboard successfully created
Target targets/blinky-myboard successfully set target.app to apps/blinky
Target targets/blinky-myboard successfully set target.bsp to hw/bsp/myboard
Target targets/blinky-myboard successfully set target.build_profile to debug

```

Fill in the `bsp.yml` file

The template `hw/bsp/myboard/bsp.yml` file is missing some values that need to be added. It also assumes certain information that may not be appropriate for your BSP. We need to get this file into a usable state.

Missing fields are indicated by the presence of `XXX` markers. Here are the first several lines of our `bsp.yml` file where all the incomplete fields are located:

```

bsp.arch: # XXX <MCU-architecture>
bsp.compiler: # XXX <compiler-package>

```

(continues on next page)

(continued from previous page)

```
bsp.linkerscript:
  - 'hw/bsp/myboard/myboard.1d'
  # - XXX mcu-linker-script
bsp.linkerscript.BOOT_LOADER.OVERWRITE:
  - 'hw/bsp/myboard/boot-myboard.1d'
  # - XXX mcu-linker-script
```

So we need to specify the following:

- MCU architecture
- Compiler package
- MCU linker script

Our example BSP uses an nRF52 MCU, which implements the `cortex_m4` architecture. We use this information to fill in the incomplete fields:

```
bsp.arch: cortex_m4
bsp.compiler: '@apache-mynewt-core/compiler/arm-none-eabi-m4'
bsp.linkerscript:
  - 'hw/bsp/myboard/myboard.1d'
  - '@apache-mynewt-core/hw/mcu/nordic/nrf52xxx/nrf52.1d'
bsp.linkerscript.BOOT_LOADER.OVERWRITE:
  - 'hw/bsp/myboard/boot-myboard.1d'
  - '@apache-mynewt-core/hw/mcu/nordic/nrf52xxx/nrf52.1d'
```

Naturally, these values must be adjusted accordingly for other MCU types.

Flash map

At the bottom of the `bsp.yml` file is the flash map. The flash map partitions the BSP's flash memory into sections called areas. Flash areas are further categorized into two types: 1) system areas, and 2) user areas. These two area types are defined below.

System areas

- Used by Mynewt core components.
- BSP support is mandatory in most cases.
- Use reserved names.

User areas

- Used by application code and supplementary libraries.
- Identified by user-assigned names.
- Have unique user-assigned numeric identifiers for access by C code.

The flash map in the template `bsp.yml` file is suitable for an MCU with 512kB of internal flash. You may need to adjust the area offsets and sizes if your BSP does not have 512kB of internal flash.

The system flash areas are briefly described below:

Flash area	Description
FLASH_AREA_BOOTLOADER	Contains the Mynewt boot loader.
FLASH_AREA_IMAGE_0	Contains the active Mynewt application image.
FLASH_AREA_IMAGE_1	Contains the secondary image; used for image upgrade.
FLASH_AREA_IMAGE_SCRATCH	Used by the boot loader during image swap.

Add the MCU dependency to `pkg.yml`

A package's dependencies are listed in its `pkg.yml` file. A BSP package always depends on its corresponding MCU package, so let's add that dependency to our BSP now. The `pkg.deps` section of our `hw/bsp/myboard/pkg.yml` file currently looks like this:

```
pkg.deps :
# - XXX <MCU-package>
- '@apache-mynewt-core/kernel/os'
- '@apache-mynewt-core/libc/baselIBC'
```

Continuing with our example nRF52 BSP, we replace the marked line as follows:

```
pkg.deps :
- '@apache-mynewt-core/hw/mcu/nordic/nrf52xxx'
- '@apache-mynewt-core/kernel/os'
- '@apache-mynewt-core/libc/baselIBC'
```

Again, the particulars depend on the MCU that your BSP uses.

Check the BSP linker scripts

Linker scripts are a key component of the BSP package. They specify how code and data are arranged in the MCU's memory. Our BSP package contains two linker scripts:

Filename	Description
<code>myboard.1d</code>	Linker script for Mynewt application images.
<code>boot-myboard.1d</code>	Linker script for the Mynewt boot loader.

First, we will deal with the application linker script. You may have noticed that the `bsp.linkerscript` item in `bsp.yml` actually specifies two linker scripts:

- BSP linker script (`hw/bsp/myboard.1d`)
- MCU linker script (`@apache-mynewt-core/hw/mcu/nordic/nrf52xxx/nrf52.1d`)

Both linker scripts get used in combination when you build a Mynewt image. Typically, all the complexity is isolated to the MCU linker script, while the BSP linker script just contains minimal size and offset information. This makes the job of creating a BSP package much simpler.

Our `myboard.1d` file has the following contents:

```
MEMORY
{
    FLASH (rx) : ORIGIN = 0x00008000, LENGTH = 0x3a000
```

(continues on next page)

(continued from previous page)

```

RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 0x10000
}

/* This linker script is used for images and thus contains an image header */
_imghdr_size = 0x20;

```

Our task is to ensure the offset (ORIGIN) and size (LENGTH) values are correct for the FLASH and RAM regions. Note that the FLASH region does not specify the board's entire internal flash; it only describes the area of the flash dedicated to containing the running Mynewt image. The bounds of the FLASH region should match those of the FLASH_AREA_IMAGE_0 area in the BSP's flash map.

The _imghdr_size is always 0x20, so it can remain unchanged.

The second linker script, boot-myboard.ld, is quite similar to the first. The important difference is the FLASH region: it describes the area of flash which contains the boot loader rather than an image. The bounds of this region should match those of the FLASH_AREA_BOOTLOADER area in the BSP's flash map. For more information about the Mynewt boot loader, see [this page](#).

Copy the download and debug scripts

The newt command line tool uses a set of scripts to load and run Mynewt images. It is the BSP package that provides these scripts.

As with the linker scripts, most of the work done by the download and debug scripts is isolated to the MCU package. The BSP scripts are quite simple, and you can likely get away with just copying them from another BSP. The template myboard_debug.sh script indicates which BSP to copy from:

```

#!/bin/sh

# This script attaches a gdb session to a Mynewt image running on your BSP.

# If your BSP uses JLink, a good example script to copy is:
#     repos/apache-mynewt-core/hw/bsp/nordic_pca10040/nordic_pca10040_debug.sh
#
# If your BSP uses OpenOCD, a good example script to copy is:
#     repos/apache-mynewt-core/hw/bsp/rb-nano2/rb-nano2_debug.sh

```

Our example Nordic nRF52 BSP uses JLink, so we will copy the Nordic PCA10040 (nRF52 DK) BSP's scripts:

```

cp repos/apache-mynewt-core/hw/bsp/nordic_pca10040/nordic_pca10040_debug.sh hw/bsp/
`-myboard/myboard_debug.sh
cp repos/apache-mynewt-core/hw/bsp/nordic_pca10040/nordic_pca10040_download.sh hw/bsp/
`-myboard/myboard_download.sh

```

Fill in BSP functions and defines

There are a few particulars missing from the BSP's C code. These areas are marked with XXX comments to make them easier to spot. The missing pieces are summarized in the table below:

File	Description	Notes
src/ hal_bsp.c	hal_bsp_flash_dev() needs to return a pointer to the MCU's flash object when id == 0.	The flash object is defined in MCU's hal_flash.c file.
include/ bsp/bsp.h	Define LED_BLINK_PIN to the pin number of the BSP's primary LED.	Required by the blinky application.

For our nRF52 BSP, we modify these files as follows:

src/hal_bsp.c:

```
#include "mcu/nrf52_hal.h"

const struct hal_flash *
hal_bsp_flash_dev(uint8_t id)
{
    switch (id) {
        case 0:
            /* MCU internal flash. */
            return &nrf52k_flash_dev;

        default:
            /* External flash. Assume not present in this BSP. */
            return NULL;
    }
}
```

include/bsp/bsp.h:

```
#define RAM_SIZE      0x10000

/* Put additional BSP definitions here. */

#define LED_BLINK_PIN  17
```

Add startup code

Now we need to add the BSP's assembly startup code. Among other things, this is the code that gets executed immediately on power up, before the Mynewt OS is running. This code must perform a few basic tasks:

- Assign labels to memory region boundaries.
- Define some interrupt request handlers.
- Define the Reset_Handler function, which:
 - Zeroes the .bss section.
 - Copies static data from the image to the .data section.
 - Starts the Mynewt OS.

This file is named according to the following pattern: hw/bsp/myboard/src/arch/<ARCH>/gcc_startup_<MCU>.s

The best approach for creating this file is to copy from other BSPs. If there is another BSP that uses the same MCU, you might be able to use most or all of its startup file.

For our example BSP, we'll just copy the Nordic PCA10040 (nRF52 DK) BSP's startup code:

```
$ mkdir -p hw/bsp/myboard/src/arch/cortex_m4
$ cp repos/apache-mynewt-core/hw/bsp/nordic_pca10040/src/arch/cortex_m4/gcc_startup_nrf52.s hw/bsp/myboard/src/arch/cortex_m4/
```

Satisfy MCU requirements

The MCU package probably requires some build-time configuration. Typically, it is the BSP which provides this configuration. Completing this step will likely involve some trial and error as each unmet requirement gets reported as a build error.

Our example nRF52 BSP requires the following changes:

1. Macro indicating MCU type. We add this to our BSP's pkg.yml file:

```
pkg.cflags:
  - '-DNRF52'
```

2. Enable exactly one low-frequency timer setting in our BSP's syscfg.yml file. This is required by the nRF51 and nRF52 MCU packages:

```
# Settings this BSP overrides.
syscfg.vals:
  XTAL_32768: 1
```

Test it

Now it's finally time to test the BSP package. Build and load your boot and blinky targets as follows:

```
$ newt build boot-myboard
$ newt load boot-myboard
$ newt run blinky-myboard 0
```

If everything is correct, the blinky app should successfully build, and you should be presented with a gdb prompt. Type c <enter> (continue) to see your board's LED blink.

Appendix A: BSP files

The table below lists the files required by all BSP packages. The naming scheme assumes a BSP called "myboard".

File Path Name	Description
pkg.yml	Defines a Mynewt package for the BSP.
bsp.yml	Defines BSP-specific settings.
include/bsp/bsp.h	Contains additional BSP-specific settings.
src/hal_bsp.c	Contains code to initialize the BSP's peripherals.
src/sbrk.c	Low level heap management required by <code>malloc()</code> .
src/arch/<ARCH>/gcc_startup_myboard.s	Startup assembly code to bring up Mynewt
myboard.ld	A linker script providing the memory map for a Mynewt application.
boot-myboard.ld	A linker script providing the memory map for the Mynewt bootloader.
myboard_download.sh	A bash script to download code onto your platform.
myboard_debug.sh	A bash script to initiate a gdb session with your platform.

A BSP can also contain the following optional files:

File Path Name	Description
split-myboard.ld	A linker script providing the memory map for the “application” half of a <i>split image</i>
no-boot-myboard.ld	A linker script providing the memory map for your bootloader
src/arch/<ARCH>/gcc_startup_myboard_split.s	Startup assembly code to bring up the “application” half of a <i>split image</i> .
myboard_download.cmd	An MSDOS batch file to download code onto your platform; required for Windows support.
myboard_debug.cmd	An MSDOS batch file to intiate a gdb session with your platform; required for Windows support.

6.6.7 Porting Mynewt to a new MCU

Porting Mynewt to a new MCU is not a difficult task if the core CPU architectures is already supported.

The depth of work depends on the amount of HAL (Hardware Abstraction Layer) support you need and provide in your port.

To get started:

- Create a `hw/mcu/mymcu` directory where `mymcu` is the MCU you are porting to. Replace the name `mymcu` with a description of the MCU you are using.
- Create a `hw/mcu/mymcu/variant` directory where the variant is the specific variant of the part you are usuing. Many MCU parts have variants with different capabilities (RAM, FLASH etc) or different pinouts. Replace `variant` with a description of the variant of the part you are using.
- Create a `hw/mcu/mymcu/variant/pkg.yml` file. Copy from another mcu and fill out the relevant information
- Create `hw/mcu/mymcu/variant/include`, `hw/mcu/mymcu/variant/include/mcu`, and `hw/mcu/mymcu/variant/src` directories to contain the code for your mcu.

At this point there are two main tasks to complete.

- Implement any OS-specific code required by the OS
- Implement the HAL functionality that you are looking for

Please contact the Mynewt development list for help and advice porting to new MCU.

6.6.8 Porting Mynewt to a new CPU Architecture

A new CPU architecture typically requires the following:

- A new compiler
- New architecture-specific code for the OS
- Helper libraries to help others porting to the same architecture

These are discussed below:

Create A New Compiler

NOTE: Newt does not automatically install the compilers require to build all platforms. Its up to the user using their local machines package manager to install the compilers. The step described here just registers the compiler with newt.

Create a new directory (named after the compiler you are adding). Copy the `pkg.yml` file from another compiler.

Edit the `pkg.yml` file and change the configuration attributes to match your compiler. Most are self-explanatory paths to different compiler and linker tools. There are a few configuration attributes worth noting.

Configuration Attributes	Description
<code>pkg.keywords</code>	Specific keywords to help others search for this using newt
<code>compiler.flags.default</code>	default compiler flags for this architecture
<code>compiler.flags.optimized</code>	additional flags when the newt tool builds an optimized image
<code>compiler.flags.debug</code>	additional flags when the newt tool builds a debug image

Implement Architecture-specific OS code

There are several architecture-specific code functions that are required when implementing a new architecture. You can find examples in the `sim` architecture within Mynewt.

When porting to a new CPU architecture, use the existing architectures as samples when writing your implementation.

Please contact the Mynewt development list for help and advice porting to new MCU.

6.7 Baselibc

Baselibc is a very simple libc for embedded systems geared primarily for 32-bit microcontrollers in the 10-100kB memory range. The library of basic system calls and facilities compiles to less than 5kB total on Cortex-M3, and much less if some functions aren't used.

The code is based on klibc and tinyprintf modules, and licensed under the BSD license.

Baselibc comes from <https://github.com/PetteriAimonen/Baselibc.git>

6.7.1 Description

Mynewt OS can utilize libc which comes with compiler (e.g. newlib bundled with some binary distributions of arm-none-eabi-gcc). However, you may choose to replace the libc with baselibc for a reduced image size. Baselibc optimizes for size rather than performance, which is usually a more important goal in embedded environments.

6.7.2 How to switch to baselibc

In order to switch from using libc to using baselibc you have to add the baselibc pkg as a dependency in the project pkg. Specifying this dependency ensures that the linker first looks for the functions in baselibc before falling back to libc while creating the executable. For example, project boot uses baselibc. Its project description file `boot.yml` looks like the following:

```
project.name: boot
project.identities: bootloader
project.pkgs:
    - libs/os
    - libs/bootutil
    - libs/nffs
    - libs/console/stub
    - libs/util
    - libs/baselibc
```

6.7.3 List of Functions

Documentation for libc functions is available from multiple places. One example are the on-line manual pages at [FreeBSD website](#).

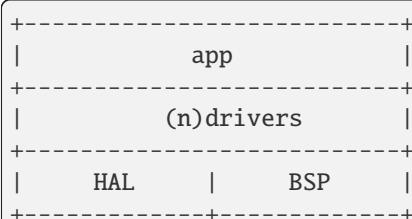
baselibc supports most libc functionality; malloc(), printf-family, string handling, and conversion routines.

There is some limited functionality and support for floating point numbers and ‘long long’, but contributions are always welcome!

6.8 Drivers

6.8.1 Description

Device drivers in the Mynewt context includes libraries that interface with devices external to the CPU. These devices are connected to the CPU via standard peripherals such as SPI, GPIO, I2C etc. Device drivers leverage the base HAL services in Mynewt to provide friendly abstractions to application developers.



- The Board Support Package (BSP) abstracts board specific configurations e.g. CPU frequency, input voltage, LED pins, on-chip flash map etc.

- The Hardware Abstraction Layer (HAL) abstracts architecture-specific functionality. It initializes and enables components within a master processor. It is designed to be portable across all the various MCUs supported in Mynewt (e.g. Nordic’s nRF51, Nordic’s nRF52, NXP’s MK64F12 etc.). It includes code that initializes and manages access to components of the board such as board buses (I2C, PCI, PCMCIA, etc.), off-chip memory (controllers, level 2+ cache, Flash, etc.), and off-chip I/O (Ethernet, RS-232, display, mouse, etc.)
- The driver sits atop the BSP and HAL. It abstracts the common modes of operation for each peripheral device connected via the standard interfaces to the processor. There may be multiple driver implementations of differing complexities for a particular peripheral device. For example, for an Analog to Digital Converter (ADC) peripheral you might have a simple driver that does blocking ADC reads and uses the HAL only. You might have a more complex driver that can deal with both internal and external ADCs, and has chip specific support for doing things like DMA’ing ADC reads into a buffer and posting an event to a task every ‘n’ samples. The drivers are the ones that register with the kernel’s power management APIs, and manage turning on and off peripherals and external chipsets, etc. The Mynewt core repository comes with a base set of drivers to help the user get started.

6.8.2 General design principles

- Device drivers should have a consistent structure and unified interface whenever possible. For example, we have a top-level package, “adc”, which contains the interface for all ADC drivers, and then we have the individual implementation of the driver itself. The following source files point to this:
 - high-level ADC API: `hw/drivers/adc/include/adc/adc.h`
 - implementation of ADC for STM32F4: `hw/drivers/adc/adc_stm32f4/src/adc_stm32f4.c` (As of the 1.0.0-beta release, ADC for nRF51 and nRF52 are available at an external [repo](#). They are expected to be pulled into the core repo on Apache Mynewt after the license terms are clarified.). The only exported call in this example is `int stm32f4_adc_dev_init(struct os_dev *, void *)` which is passed as a function pointer to `os_dev_create()` in `hal_bsp.c`, when the adc device is created.
- Device drivers should be easy to use. In Mynewt, creating a device initializes it as well, making it readily available for the user to open, use (e.g. read, configure etc.) and close. Creating a device is simple using `os_dev_create(struct os_dev *dev, char *name, uint8_t stage, uint8_t priority, os_dev_init_func_t od_init, void *arg)`. The `od_init` function is defined within the appropriate driver directory e.g. `stm32f4_adc_dev_init` in `hw/drivers/adc/adc_stm32f4/src/adc_stm32f4.c` for the ADC device initialization function.
- The implementation should allow for builds optimized for minimal memory usage. Additional functionality can be enabled by writing a more complex driver (usually based on the simple implementation included by default in the core repository) and optionally compiling the relevant packages in. Typically, only a basic driver that addresses a device’s core functionality (covering ~90% of use cases) is included in the Mynewt core repository, thus keeping the footprint small.
- The implementation should allow a user to be able to instantiate multiple devices of a certain kind. In the Mynewt environment the user can, for example, maintain separate contexts for multiple ADCs over different peripheral connections such as SPI, I2C etc. It is also possible for a user to use a single peripheral interface (e.g. SPI) to drive multiple devices (e.g. ADC), and in that case the device driver has to handle the proper synchronization of the various tasks.
- Device drivers should be MCU independent. In Mynewt, device creation and operation functions are independent of the underlying MCU.
- Device drivers should be able to offer high-level interfaces for generic operations common to a particular device group. An example of such a class or group of devices is a group for sensors with generic operations such as channel discovery, configure, and read values. The organization of the driver directory is work in progress - so we encourage you to hop on the `dev@` mailing list and offer your insights!
- Device drivers should be searchable. The plan is to have the `newt` tool offer a `newt pkg search` capability. This is work in progress. You are welcome to join the conversation on the `dev@` mailing list!

6.8.3 Example

The Mynewt core repo includes an example of a driver using the HAL to provide extra functionality - the UART driver. It uses HAL GPIO and UART to provide multiple serial ports on the NRF52 (but allowed on other platforms too.)

The gist of the driver design is that there is an API for the driver (for use by applications), and then sub-packages to that driver that implement that driver API using the HAL and BSP APIs.

6.8.4 Implemented drivers

Drivers live under `hw/drivers`. The current list of supported drivers includes:

Driver	Description
adc	TODO: ADC driver.
flash	SPI/I2C flash drivers.
lwip	TODO: LWIP.
mmc	MMC/SD card driver.
sensors	TODO: sensors.
uart	TODO: UART driver.
chg_ctrl	Charge control drivers.

6.8.5 flash

The flash driver subsystem is a work in progress which aims at supporting common external SPI/I2C flash/eeprom memory chips. This is equivalent to what Linux calls MTD for **Memory Technology Devices**.

At the moment the only `flash` device that is already supported is the AT45DBxxx SPI flash family with the `at45db` driver.

The flash driver aims for full compatibility with the `hal_flash` API, which means initialization and usage can be performed by any `fs` that supports the `hal_flash` interface.

Initialization

To be compatible with the standard `hal_flash` interface, the `at45db` driver embeds a `struct hal_flash` to its own `struct at45db_dev`. The whole `at45db_dev` struct is shown below.

```
struct at45db_dev {
    struct hal_flash hal;
    struct hal_spi_settings *settings;
    int spi_num;
    void *spi_cfg;                      /** Low-level MCU SPI config */
    int ss_pin;
    uint32_t baudrate;
    uint16_t page_size;                  /** Page size to be used, valid: 512 and 528 */
    uint8_t disable_auto_erase;          /** Reads and writes auto-erase by default */
};
```

To ease with initialization a helper function `at45db_default_config` was added. It returns an already initialized `struct at45db_dev` leaving the user with just having to provide the SPI related config.

To initialize the device, pass the `at45db_dev` struct to `at45db_init`.

```
int at45db_init(const struct hal_flash *dev);
```

For low-level access to the device the following functions are provided:

```
int at45db_read(const struct hal_flash *dev, uint32_t addr, void *buf,
                 uint32_t len);
int at45db_write(const struct hal_flash *dev, uint32_t addr, const void *buf,
                  uint32_t len);
int at45db_erase_sector(const struct hal_flash *dev, uint32_t sector_address);
int at45db_sector_info(const struct hal_flash *dev, int idx, uint32_t *address,
                      uint32_t *sz);
```

Also, `nffs` is able to run on the device due to the fact that standard `hal_flash` interface compatibility is provided. Due to current limitations of `nffs`, it can only run on `at45db` if the internal flash of the MCU is not being used.

Dependencies

To include the `at45db` driver on a project, just include it as a dependency in your `pkg.yml`:

```
pkg.deps:
  - "@apache-mynewt-core/hw/drivers/flash/at45db"
```

Header file

The `at45db` SPI flash follows the standard `hal_flash` interface but requires that a special struct

```
#include <at45db/at45db.h>
```

Example

This following examples assume that the `at45db` is being used on a STM32F4 MCU.

```
static const int SPI_SS_PIN    = MCU_GPIO_PORTA(4);
static const int SPI_SCK_PIN   = MCU_GPIO_PORTA(5);
static const int SPI_MISO_PIN = MCU_GPIO_PORTA(6);
static const int SPI_MOSI_PIN = MCU_GPIO_PORTA(7);

struct stm32f4_hal_spi_cfg spi_cfg = {
    .ss_pin    = SPI_SS_PIN,
    .sck_pin   = SPI_SCK_PIN,
    .miso_pin  = SPI_MISO_PIN,
    .mosi_pin  = SPI_MOSI_PIN,
    .irq_prio  = 2
};

struct at45db_dev *my_at45db_dev = NULL;

my_at45db_dev = at45db_default_config();
my_at45db_dev->spi_num = 0;
my_at45db_dev->spi_cfg = &spi_cfg;
```

(continues on next page)

(continued from previous page)

```
my_at45db_dev->ss_pin = spi_cfg.ss_pin;

rc = at45db_init((struct hal_flash *) my_at45db_dev);
if (rc) {
    /* XXX: error handling */
}
```

The enable nffs to run on the at45db, the flash_id 0 needs to map to provide a mapping from 0 to this struct.

```
const struct hal_flash *
hal_bsp_flash_dev(uint8_t id)
{
    if (id != 0) {
        return NULL;
    }
    return &my_at45db_dev;
}
```

6.8.6 mmc

The MMC driver provides support for SPI based MMC/SDcard interfaces. It exports a `disk_ops` struct that can be used by any FS. Currently only `fatfs` can run over MMC.

Initialization

```
int mmc_init(int spi_num, void *spi_cfg, int ss_pin)
```

Initializes the mmc driver to be used by a FS.

MMC uses the `hal_gpio` interface to access the SPI `ss_pin` and the `hal_spi` interface for the communication with the card. `spi_cfg` must be a hw dependent structure used by `hal_spi_init` to initialize the SPI subsystem.

Dependencies

To include the `mmc` driver on a project, just include it as a dependency in your `pkg.yml`:

```
pkg.deps:
  - "@apache-mynewt-core/hw/drivers/mmc"
```

Returned values

MMC functions return one of the following status codes:

Return code	Description
MMC_OK	Success.
MMC_CARD_ERROR	General failure on the card.
MMC_READ_ERROR	Error reading from the card.
MMC_WRITE_ERROR	Error writing to the card.
MMC_TIMEOUT	Timed out waiting for the execution of a command.
MMC_PARAM_ERROR	An invalid parameter was given to a function.
MMC_CRC_ERROR	CRC error reading card.
MMC_DEVICE_ERROR	Tried to use an invalid device.
MMC_RESPONSE_ERROR	A command received an invalid response.
MMC_VOLTAGE_ERROR	The interface doesn't support the requested voltage.
MMC_INVALID_COMMAND	The interface haven't accepted some command.
MMC_ERASE_ERROR	Error erasing the current card.
MMC_ADDR_ERROR	Tried to access an invalid address.

Header file

```
#include "mmc/mmc.h"
```

Example

This example runs on the STM32F4-Discovery and prints out a listing of the root directory on the currently installed card.

```
// NOTE: error handling removed for clarity!

struct stm32f4_hal_spi_cfg spi_cfg = {
    .ss_pin    = SPI_SS_PIN,
    .sck_pin   = SPI_SCK_PIN,
    .miso_pin  = SPI_MISO_PIN,
    .mosi_pin  = SPI_MOSI_PIN,
    .irq_prio  = 2
};

mmc_init(0, &spi_cfg, spi_cfg.ss_pin);
disk_register("mmc0", "fatfs", &mmc_ops);

fs_opendir("mmc0:/", &dir);

while (1) {
    rc = fs_readdir(dir, &dirent);
    if (rc == FS_ENOENT) {
        break;
    }

    fs_dirent_name(dirent, sizeof(out_name), out_name, &u8_len);
    printf("%s\n", out_name);
}

fs_closedir(dir);
```

6.8.7 Charge control drivers

The charge control drivers provides support for various battery chargers. They provide a abstract hw/charge-control Battery Charge Controller IC Interface. The available drivers are for:

- ADP5061
- BQ24040
- DA1469x
- *SGM4056 Charger*

Initialization

Most of the drivers need some driver-specific initialization. This is normally done in the BSP by making a call to `os_dev_create` with a reference to the driver init function and a driver-specific configuration.

For the SGM4056 this will look this:

```
#include "sgm4056/sgm4056.h"

...
static struct sgm4056_dev_config os_bsp_charger_config = {
    .power_presence_pin = CHARGER_POWER_PRESENCE_PIN,
    .charge_indicator_pin = CHARGER_CHARGE_PIN,
};

...
rc = os_dev_create(&os_bsp_charger.dev, "charger",
                   OS_DEV_INIT_KERNEL, OS_DEV_INIT_PRIO_DEFAULT,
                   sgm4056_dev_init, &os_bsp_charger_config);
```

Usage

After initialization the general charge control interface can be used to obtain the data from the charger. First you need to obtain a reference to the charger.

```
#include "charge-control/charge_control.h"

...
struct charge_control *charger;

charger = charge_control_mgr_find_next_bytype(CHARGE_CONTROL_TYPE_STATUS, NULL);
assert(charger);
```

This will find any charger that has the ability to report STATUS. Then you can obtain the status of the charger using `charge_control_read`. This function will retrieve the charger status and call a callback function. Lets first define that callback function:

```
static int
charger_data_callback(struct charge_ctrl *chg_ctrl, void *arg,
                      void *data, charge_control_type_t type)
{
    if (type == CHARGE_CONTROL_TYPE_STATUS) {
        charge_control_status_t charger_status = *(charge_control_status_t*)(data);

        console_printf("Charger state is %i", charger_status);
        console_flush();
    }
    return 0;
}
```

The important arguments are the data pointer and the type of data. This function prints the status to the console as an integer. You need to extend it for your use-case. Now we can call the `charge_control_read` function:

```
rc = charge_control_read(charger, CHARGE_CONTROL_TYPE_STATUS,
                         charger_data_callback, NULL, OS_TIMEOUT_NEVER);
assert(rc == 0);
```

This causes the charge control code to obtain the status and call the callback.

Alternatively you can register a listener that will be called periodically and on change. For this you need to define a `charge_control_listener` and register it with charge control:

```
struct charge_control_listener charger_listener = {
    .ccl_type = CHARGE_CONTROL_TYPE_STATUS,
    .ccl_func = charger_data_callback,
};

...

rc = charge_control_set_poll_rate_ms("charger", 10000);
assert(rc == 0);

rc = charge_control_register_listener(charger, &charger_listener);
assert(rc == 0);
```

This sets the interval to 10 seconds and registers our callback as listener. This means that the callback will be called every 10 seconds and on interrupt of the charger.

Dependencies

To include a charge control driver on a project, just include it as a dependency in your `pkg.yml`. This should be done in the BSP. For example:

```
pkg.deps:
  - "@apache-mynewt-core/hw/drivers/chg_ctrl/sgm4056"
```

SGM4056 Charger

The SGM4056 charger is able to report its status via two output pins. If these pins are connected to GPIO inputs, then the `sgm4056` driver can derive the charger status.

Initialization

Initialization of the `sgm4056` driver is normally done in the BSP. This is done by creating a OS device using `os_dev_create`.

```
#include "sgm4056/sgm4056.h"

...
static struct sgm4056_dev os_bsp_charger;
static struct sgm4056_dev_config os_bsp_charger_config = {
    .power_presence_pin = CHARGER_POWER_PRESENCE_PIN,
    .charge_indicator_pin = CHARGER_CHARGE_PIN,
};

void
hal_bsp_init(void)
{
    int rc;

    ...

    /* Create charge controller */
    rc = os_dev_create(&os_bsp_charger.dev, "charger",
                      OS_DEV_INIT_KERNEL, OS_DEV_INIT_PRIO_DEFAULT,
                      sgm4056_dev_init, &os_bsp_charger_config);
    assert(rc == 0);
}
```

First we create a instance of `struct sgm4056_dev`, this will be initialised later. Then we create a instance of `struct sgm4056_dev_config`, which contains the pin numbers of the gpio connected to the charger. Then we call `os_dev_create` with a pointer to the device, a device name, some defaults, the `sgm4056_dev_init` initializor and a pointer to config.

Usage

There a two ways to use the driver: directly or via `charge-control`.

When using the driver directly, you need to open the OS device and then use the driver functions.

```
#include "sgm4056/sgm4056.h"

...
int
main(int argc, char **argv)
{
```

(continues on next page)

(continued from previous page)

```

struct sgm4056_dev *charger;

charger = (struct sgm4056_dev *) os_dev_open("charger", 0, 0);
assert(charger);

while (1) {
    rc = sgm4056_get_charger_status(charger, &charger_status);
    assert(rc == 0);

    console_printf("Charger state = %i\n", charger_status);
    console_flush();
}
}

```

When using charge-control interface, you need to enable it in syscfg and then follow the general instructions at *Initialization*.

```

syscfg.vals:
# Enable charge control integration
SGM4056_USE_CHARGE_CONTROL: 1

```

6.9 Newt Manager

Newt Manager is the protocol that enables your Mynewt application to communicate remotely with your device running the Mynewt OS in order to configure, manage, conduct maintenance, and monitor it. The core device management module is called `mgmt` and offers multiple options for invoking the appropriate newt manager commands for various operations on the device e.g. enabling and collecting logs, configuring and retrieving stats, resetting the device etc.

1. Use the `newtmgr` package if reduced code footprint is your primary requirement and you do not have interoperability requirements upstream for device information, discovery, and connectivity.
2. Use the `oicmgr` package if interoperability and standards-based connectivity for device interaction is your primary requirement. This package supports the OIC (Open Interconnect Consortium) Specification 1.1.0 framework from Open Connectivity Foundation (OCF).

6.9.1 Invoking Newt Manager commands

The diagram below indicates the two options available to the application developer to issue Newt Manager (`newtmgr`) commands on a Mynewt device. The application may leverage the `newtmgr` framework or the `oicmgr` framework to call the `newtmgr` commands. The latter is described in the next chapter.

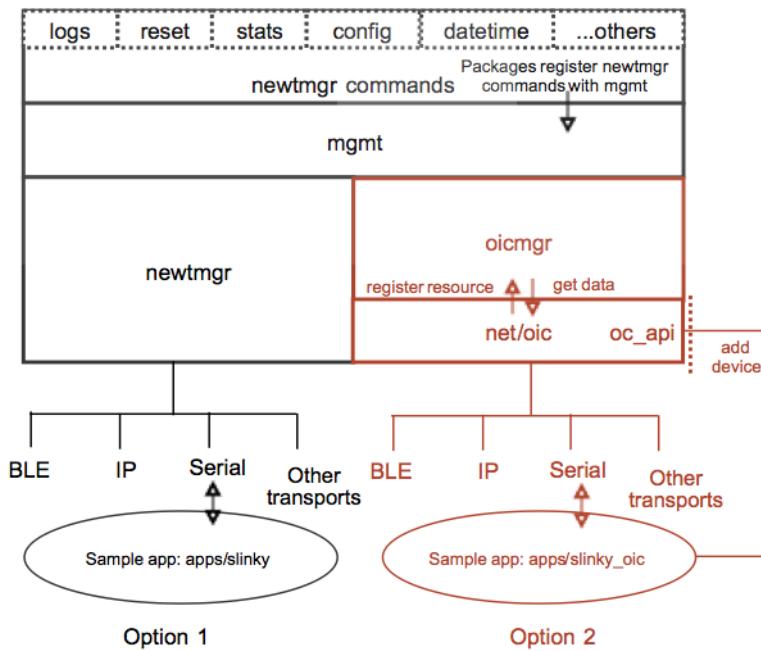


Fig. 4: Device Management

6.9.2 newtmgr

The newtmgr framework uses a simple request and response message format to send commands to the device. A message consists of an eight byte header and the message payload. The message header specifies the newtmgr command. The message payload contains the newtmgr request/response data and is encoded in CBOR (Concise Binary Object Representation) format. newtmgr supports BLE and serial connections.

The newtmgr framework has a smaller code size and memory footprint than oicmgr but does not support open connectivity.

6.9.3 Using the OIC Framework

Apache Mynewt includes support for the OIC interoperability standard through the oicmgr framework. Mynewt defines and exposes oicmgr as an OIC Server resource with the following identity and properties:

URI	/omgr
Resource Type (rt)	x.mynewt.nmgr
Interface (if)	oic.if_rw (default), oic.if.baseline
Discoverable	Yes

The newtmgr application tool uses CoAP (Constrained ApplicationProtocol) requests to send commands to oicmgr. It sends a CoAP request for **/omgr** as follows:

- Specifies the newtmgr command to execute in the URI query string.
- Initially the GET method was used for newtmgr commands that retrieve information from your application, for example, the taskstat and mpstat commands. Now it uses the PUT operation as described below..

- Uses a PUT method for newtmgr commands that send data to or modify the state of your application, for example, the echo or datetime commands.
- Sends the CBOR-encoded command request data in the CoAP message payload.

The oicmgr framework supports transport over BLE, serial, and IP connections to the device.

NewtMgr Protocol Specifics

Requests

1. The NMP op is indicated by the OIC op. The OIC op is always the same: put.
2. There are no URI Query CoAP options.
3. The NMP header is included in the payload as a key-value pair (key="h"). This pair is in the root map of the request and is a sibling of the other request fields. The value of this pair is the big-endian eight-byte NMP header with a length field of 0.

Responses

1. As with requests, the NMP header is included in the payload as a key-value pair (key="h").
2. There is no "r" or "w" field. The response fields are inserted into the root map as a sibling of the "h" pair.
3. Errors encountered during processing of NMP requests are reported identically to other NMP responses (embedded NMP response).

Notes

Keys that start with an underscore are reserved to the OIC manager protocol (e.g., "_h"). NMP requests and responses must not name any of their fields with a leading underscore.

6.9.4 Customizing Newt Manager Usage with mgmt

The **mgmt** package enables you to customize Newt Manager (in either the newtmgr or oicmgr framework) to only process the commands that your application uses. The newtmgr commands are divided into management groups. A manager package implements the commands for a group. It implements the handlers that process the commands for the group and registers the handlers with mgmt. When newtmgr or oicmgr receives a newtmgr command, it looks up the handler for the command (by management group id and command id) from mgmt and calls the handler to process the command.

The system level management groups are listed in following table:

Management Group	newtmgr Commands	Package
MGMT_GROUP_ID_DEFAULT	echo taskstat mpstat datetime reset	mgmt/newtmgr/nmgr_os
MGMT_GROUP_ID_IMAGE	image	mgmt/imgmgr
MGMT_GROUP_ID_STATS	stat	sys/stats
MGMT_GROUP_ID_CONFIG	config	sys/config
MGMT_GROUP_ID_LOGS	log	sys/log
MGMT_GROUP_ID_CRASH	crash	test/crash_test
MGMT_GROUP_ID_RUNTEST	run	test/runtest

Both newtmgr and ocimgr process the MGMT_GROUP_ID_DEFAULT commands by default. You can also use mgmt to add user defined management group commands.

6.10 MCU Manager (mcumgr)

mcumgr, a derivative of newtmgr, is a device/image/embedded OS management library with pluggable transport and encoding components and is, by design, operating system and hardware independent. It relies on hardware porting layers from the operating system it runs on. Currently, mcumgr runs on both the Apache Mynewt and Zephyr operating systems.

So how is it different from newtmgr? There is one substantial difference between the two: newtmgr supports two wire formats - NMP (plain newtmgr protocol) and OMP (CoAP newtmgr protocol). mcumgr only supports NMP (called “SMP” in mcumgr).

NMP is a simple binary format: 8-byte header plus CBOR payload OMP uses CoAP requests to the */omgr* CoAP resource

A request has the same effect on the receiving device regardless of wire format, but OMP fits more cleanly in a system that is already using CoAP. Documentation on mcumgr can be found in the source code repository: [mynewt-mcumgr](#)

There has been some discussion about combining all this functionality into a single library (mcumgr). Your views are welcome on dev@mynewt.apache.org.

6.11 Image Manager

6.11.1 Description

This library accepts incoming image management commands from newtmgr and acts on them.

Images can be uploaded, present images listed, and system can be told to switch to another image.

Currently the package assumes that there are 2 image slots, one active one and another one in standby. When new image is uploaded, it replaces the one in standby slot. This is the model for scenario when MCU has internal flash only, it executes the code from that flash, and there is enough space to store 2 full images.

Image manager interacts with bootloader by telling it to boot to a specific image. At the moment this has to be done by writing a file which contains a version number of the image to boot. Note that image manager itself does not replace the active image.

Image manager also can upload files to filesystem as well as download them.

Note that commands accessing filesystems (next boot target, file upload/download) will not be available unless project includes filesystem implementation.

6.11.2 List of Functions

The functions available in imgmgr are:

Function	Description
<code>imgm_ver_parse()</code>	Parses character string containing specified image version number and writes that to given <code>image_version</code> struct.
<code>imgm_ver_str()</code>	Takes version string from specified <code>image_version</code> struct and formats it into a printable string.

6.11.3 imgmgr_module_init

```
void
imgmgr_module_init(void)
```

Registers the image manager commands with the mgmt package. `sysinit()` automatically calls this function during system initialization.

6.11.4 imgr_ver_parse

```
int
imgr_ver_parse(char *src, struct image_version *ver)
```

Parses character string containing image version number `src` and writes that to `ver`. Version number string should be in format Major and minor numbers should be within range 0-255, revision between 0-65535 and build_number 0-4294967295.

Arguments

Arguments	Description
<code>src</code>	Pointer to C string that contains version number being parsed
<code>ver</code>	Image version number structure containing the returned value

Returned values

0 on success and <0 if version number string could not be parsed.

Notes

Numbers within the string are separated by .. The first number is the major number, and must be provided. Rest of the numbers (minor etc.) are optional.

Example

```
int main(int argc, char **argv)
{
    struct image_version hdr_ver;
    int rc;
    ...

    rc = imgr_ver_parse(argv[3], &hdr_ver);
    if (rc != 0) {
        print_usage(stderr);
        return 1;
    }
    ...
}
```

6.11.5 imgr_ver_str

```
int  
imgr_ver_str(struct image_version *ver, char *dst)
```

Takes the version string from `ver` and formats that into a printable string to `dst`. Caller must make sure that `dst` contains enough space to hold maximum length version string. The convenience definition for max length version string is named `IMGMGR_MAX_VER_STR`.

Arguments

Arguments	Description
ver	Image version number structure containing the value being formatted
dst	Pointer to C string where results will be stored

Returned values

Function returns the number of characters filled into the destination string.

Notes

If build number is 0 in image version structure, it will be left out of the string.

Example

```
static void  
imgr_ver_jsonstr(struct json_encoder *enc, char *key,  
    struct image_version *ver)  
{  
    char ver_str[IMGMGR_MAX_VER_STR];  
    int ver_len;  
    ...  
    ver_len = imgr_ver_str(ver, ver_str)  
    ...  
}
```

6.12 Compile-Time Configuration

This guide describes how Mynewt manages system configuration. It shows you how to tell Mynewt to use default or customized values to configure packages that you develop or use to build a target. This guide:

- Assumes you have read the [Concepts](#) section that describes the Mynewt package hierarchy and its use of the `pkg.yml` and `syscfg.yml` files.
- Assumes you have read the Mynewt [Theory of Operations](#) and are familiar with how newt determines package dependencies for your target build.

- *System Configuration Setting Definitions and Values*
 - *Examples of Configuration Settings*
 - * *Example 1*
 - * *Example 2*
 - * *Example 3*
- *Overriding System Configuration Setting Values*
 - *Resolving Override Conflicts*
 - *Examples of Overrides*
 - * *Example 4*
 - * *Example 5*
- *Conditional Settings*
 - *Examples of Conditional Settings*
 - * *Example 6*
 - * *Example 7*
- *Generated syscfg.h and Referencing System Configuration Settings*
 - *Example of syscfg.h and How to Reference a Setting Name*

Mynewt defines several configuration parameters in the `pkg.yml` and `syscfg.yml` files. The newt tool uses this information to:

- Generate a system configuration header file that contains all the package configuration settings and values.
- Display the system configuration settings and values in the `newt target config` command.

The benefits with this approach include:

- Allows Mynewt developers to reuse other packages and easily change their configuration settings without updating source or header files when implementing new packages.
- Allows application developers to easily view the system configuration settings and values and determine the values to override for a target build.

6.12.1 System Configuration Setting Definitions and Values

A package can optionally:

- Define and expose the system configuration settings to allow other packages to override the default setting values.
- Override the system configuration setting values defined by the packages that it depends on.

You use the `defs` parameter in a `syscfg.yml` file to define the system configuration settings for a package. `defs` is a mapping (or associative array) of system configuration setting definitions. It has the following syntax:

```
syscfg.defs:
  PKGA_SYSCFG_NAME1:
    description:
```

(continues on next page)

(continued from previous page)

```
value:  
type:  
restrictions:  
PKG_A_SYSCFG_NAME2:  
    description:  
    value:  
    type:  
    restrictions:
```

Each setting definition consists of the following key-value mapping:

- A setting name for the key, such as PKG_A_SYSCFG_NAME1 in the syntax example above. **Note:** A system configuration setting name must be unique. The newt tool aborts the build when multiple packages define the same setting.
- A mapping of fields for the value. Each field itself is a key-value pair of attributes. The field keys are **description**, **value**, **type**, and **restrictions**. They are described in following table:

Field	Description
description	Describes the usage for the setting. This field is optional.
value	Specifies the default value for the setting. This field is required. The value depends on the type that you specify and can be an empty string.
type	<p>Specifies the data type for the value field. This field is optional. You can specify one of three types:</p> <p>raw: The value data is uninterpreted. This is the default type.</p> <p>task_priority: Specifies a Mynewt task priority number. The task priority number assigned to each setting must be unique and between 0 and 239. value can be one of the following: A number between 0 and 239 - The task priority number to use for the setting. any - Specify any to have newt automatically assign a priority for the setting. newt alphabetically orders all system configuration settings of this type and assigns the next highest available task priority number to each setting.</p> <p>flash_owner: Specifies a flash area. The value should be the name of a flash area defined in the BSP flash map for your target board.</p>
restrictions	<p>Specifies a list of restrictions on the setting value. This field is optional. You can specify two formats:</p> <p>\$notnull: Specifies that the setting cannot have the empty string for a value. It essentially means that an empty string is not a sensible value and a package must override it with an appropriate value.</p> <p>expression: Specifies a boolean expression of the form <code>[!]&lt;required-setting>[if &lt;base-value>]</code></p> <p>Examples:</p> <p>restrictions: !LOG_FCB - When this setting is enabled, LOG_FCB must be disabled.</p> <p>restrictions: LOG_FCB if 0 - When this setting is disabled, LOG_FCB must be enabled.</p>

Examples of Configuration Settings

Example 1

The following example is an excerpt from the sys/log/full package `syscfg.yml` file. It defines the LOG_LEVEL configuration setting to specify the log level and the LOG_NEWTMGR configuration setting to specify whether to enable or disable the newtmgr logging feature.

```
syscfg.defs:  
  LOG_LEVEL:  
    description: 'Log Level'  
    value: 0  
    type: raw  
  
  ...  
  
  LOG_NEWTMGR:  
    description: 'Enables or disables newtmgr command tool logging'  
    value: 0
```

Example 2

The following example is an excerpt from the net/nimble/controller package `syscfg.yml` file. It defines the BLE_LL_PRIO configuration setting with a task_priority type and assigns task priority 0 to the BLE link layer task.

```
syscfg.defs:  
  BLE_LL_PRIO:  
    description: 'BLE link layer task priority'  
    type: 'task_priority'  
    value: 0
```

Example 3

The following example is an excerpt from the fs/nffs package `syscfg.yml` file.

```
syscfg.defs:  
  NFFS_FLASH_AREA:  
    description: 'The flash area to use for the Newtron Flash File System'  
    type: flash_owner  
    value:  
    restrictions:  
      - $notnull
```

It defines the NFFS_FLASH_AREA configuration setting with a flash_owner type indicating that a flash area needs to be specified for the Newtron Flash File System. The flash areas are typically defined by the BSP in its `bsp.yml` file. For example, the `bsp.yml` for nrf52dk board (hw/bsp/nrf52dk/bsp.yml) defines an area named FLASH_AREA_NFFS:

```
FLASH_AREA_NFFS:  
  user_id: 1  
  device: 0
```

(continues on next page)

(continued from previous page)

```
offset: 0x0007d000
size: 12kB
```

The `syscfg.yml` file for the same board (`hw/bsp/nrf52dk/syscfg.yml`) specifies that the above area be used for `NFFS_FLASH_AREA`.

```
syscfg.vals:
CONFIG_FCB_FLASH_AREA: FLASH_AREA_NFFS
REBOOT_LOG_FLASH_AREA: FLASH_AREA_REBOOT_LOG
NFFS_FLASH_AREA: FLASH_AREA_NFFS
COREDUMP_FLASH_AREA: FLASH_AREA_IMAGE_1
```

Note that the `fs/nffs/syscfg.yml` file indicates that the `NFFS_FLASH_AREA` setting cannot be a null string; so a higher priority package must set a non-null value to it. That is exactly what the BSP package does. For more on priority of packages in setting values, see the next section.

6.12.2 Overriding System Configuration Setting Values

A package may use the `vals` parameter in its `syscfg.yml` file to override the configuration values defined by other packages. This mechanism allows:

- Mynewt developers to implement a package and easily override the system configuration setting values that are defined by the packages it depends on.
- Application developers to easily and cleanly override default configuration settings in a single place and build a customized target. You can use the `newt target config show <target-name>` command to check all the system configuration setting definitions and values in your target to determine the setting values to override. See [newt target](#).

`vals` specifies the mappings of system configuration setting name-value pairs as follows:

```
syscfg.vals:
PKGA_SYSCFG_NAME1: VALUE1
PKGA_SYSCFG_NAME2: VALUE2
...
PKGN_SYSCFG_NAME1: VALUEN
```

Note: The `newt` tool ignores overrides of undefined system configuration settings.

Resolving Override Conflicts

The `newt` tool uses package priorities to determine whether a package can override a value and resolve conflicts when multiple packages override the same system configuration setting. The following rules apply:

- A package can only override the default values of system configuration settings that are defined by lower priority packages.
- When packages with different priorities override the same system configuration setting value, `newt` uses the value from the highest priority package.
- Packages of equal priority cannot override the same system configuration setting with different values. `newt` aborts the build unless a higher priority package also overrides the value.

The following package types are listed from highest to lowest priority:

- Target

- App
- unittest - A target can include either an app or unit test package, but not both.
- BSP
- Lib - Includes all other system level packages such as os, lib, sdk, and compiler. (Note that a Lib package cannot override other Lib package settings.)

It is recommended that you override defaults at the target level instead of updating individual package `syscfg.yml` files.

Examples of Overrides

Example 4

The following example is an excerpt from the `apps/slinky` package `syscfg.yml` file. The application package overrides, in addition to other packages, the `sys/log/full` package system configuration settings defined in [Example 1](#). It changes the `LOG_NEWTMGR` system configuration setting value from 0 to 1.

```
syscfg.vals:  
    # Enable the shell task.  
    SHELL_TASK: 1  
  
    ...  
  
    # Enable newtmgr commands.  
    STATS_NEWTMGR: 1  
    LOG_NEWTMGR: 1
```

Example 5

The following example are excerpts from the `hw/bsp/native` package `bsp.yml` and `syscfg.yml` files. The package defines the flash areas for the BSP flash map in the `bsp.yml` file, and sets the `NFFS_FLASH_AREA` configuration setting value to use the flash area named `FLASH_AREA_NFFS` in the `syscfg.yml` file.

```
bsp.flash_map:  
    areas:  
        # System areas.  
        FLASH_AREA_BOOTLOADER:  
            device: 0  
            offset: 0x00000000  
            size: 16kB  
  
        ...  
  
        # User areas.  
        FLASH_AREA_REBOOT_LOG:  
            user_id: 0  
            device: 0  
            offset: 0x00004000  
            size: 16kB  
        FLASH_AREA_NFFS:
```

(continues on next page)

(continued from previous page)

```

user_id: 1
device: 0
offset: 0x00008000
size: 32kB

syscfg.vals:
NFFS_FLASH_AREA: FLASH_AREA_NFFS

```

6.12.3 Conditional Settings

Settings in most Mynewt YAML files can be made conditional on syscfg settings. For example, a package might depend on a second package *only if a syscfg setting has a particular value*. The condition can be the value of a single syscfg setting or an arbitrary expression involving many settings.

Examples of Conditional Settings

Example 6

In this example, a package depends on lib/pkg2 only if MY_SETTING has a true value.

```

pkg.deps.MY_SETTING:
  # Only depend on pkg2 if MY_SETTING is true.
  - lib/pkg2

```

A setting is “true” if it has a value other than 0 or the empty string. Undefined settings are not true.

Example 7

In this example, a package overrides the setting FOO only if BAR is greater than 5 and BAZ is not true.

```

syscfg.vals.(BAR > 5 && !BAZ):
  # Only override FOO if BAR is greater than 5 and BAZ is untrue.
  FOO: 35

```

6.12.4 Generated syscfg.h and Referencing System Configuration Settings

The newt tool processes all the package `syscfg.yml` files and generates the `bin/<target-path>/generated/include/syscfg/syscfg.h` include file with `#define` statements for each system configuration setting defined. Newt creates a `#define` for a setting name as follows:

- Adds the prefix `MYNEWT_VAL_`.
- Replaces all occurrences of “/”, “-”, and “ “ in the setting name with “_”.
- Converts all characters to upper case.

For example, the `#define` for `my-config-name` setting name is `MYNEWT_VAL_MY_CONFIG_NAME`.

Newt groups the settings in `syscfg.h` by the packages that defined them. It also indicates the package that changed a system configuration setting value.

You must use the `MYNEWT_VAL()` macro to reference a #define of a setting name in your header and source files. For example, to reference the `my-config-name` setting name, you use `MYNEWT_VAL(MY_CONFIG_NAME)`.

Note: You only need to include `syscfg/syscfg.h` in your source files to access the `syscfg.h` file. The newt tool sets the correct include path to build your target.

Example of syscfg.h and How to Reference a Setting Name

Example 6: The following example are excerpts from a sample `syscfg.h` file generated for an app/slinky target and from the `sys/log/full` package `log.c` file that shows how to reference a setting name.

The `syscfg.h` file shows the `sys/log/full` package definitions and also indicates that `app/slinky` changed the value for the `LOG_NEWTMGR` settings.

```
/**  
 * This file was generated by Apache Newt version: 1.0.0-dev  
 */  
  
#ifndef H_MYNEWT_SYSCFG_  
#define H_MYNEWT_SYSCFG_  
  
/**  
 * This macro exists to ensure code includes this header when needed. If code  
 * checks the existence of a setting directly via ifdef without including this  
 * header, the setting macro will silently evaluate to 0. In contrast, an  
 * attempt to use these macros without including this header will result in a  
 * compiler error.  
 */  
#define MYNEWT_VAL(x) MYNEWT_VAL_ ## x  
  
/* ... */  
  
/** kernel/os */  
#ifndef MYNEWT_VAL_MSYS_1_BLOCK_COUNT  
#define MYNEWT_VAL_MSYS_1_BLOCK_COUNT (12)  
#endif  
  
#ifndef MYNEWT_VAL_MSYS_1_BLOCK_SIZE  
#define MYNEWT_VAL_MSYS_1_BLOCK_SIZE (292)  
#endif  
  
/* ... */  
  
/** sys/log/full */  
  
#ifndef MYNEWT_VAL_LOG_LEVEL  
#define MYNEWT_VAL_LOG_LEVEL (0)  
#endif  
  
/* ... */  
  
/* Overridden by apps/slinky (defined by sys/log/full) */  
#ifndef MYNEWT_VAL_LOG_NEWTMGR  
#define MYNEWT_VAL_LOG_NEWTMGR (1)
```

(continues on next page)

(continued from previous page)

`#endif``#endif`

The `log_init()` function in the `sys/log/full/src/log.c` file initializes the `sys/log/full` package. It checks the `LOG_NEWTMGR` setting value, using `MYNEWT_VAL(LOG_NEWTMGR)`, to determine whether the target application has enabled the `newtmgr` log functionality. It only registers the callbacks to process the `newtmgr` log commands when the setting value is non-zero.

```
void
log_init(void)
{
    int rc;

    /* Ensure this function only gets called by sysinit. */
    SYSINIT_ASSERT_ACTIVE();

    (void)rc;

    if (log_initied) {
        return;
    }
    log_initied = 1;

    /* ... */

#if MYNEWT_VAL(LOG_NEWTMGR)
    rc = log_nmgr_register_group();
    SYSINIT_PANIC_ASSERT(rc == 0);
#endif
}
```

6.12.5 Validation and Error Messages

With multiple packages defining and overriding system configuration settings, it is easy to introduce conflicts and violations that are difficult to find. The `newt build <target-name>` command validates the setting definitions and value overrides for all the packages in the target to ensure a valid and consistent build. It aborts the build when it detects violations or ambiguities between packages.

The following sections describe the error conditions that newt detects and the error messages that it outputs. For most errors, newt also outputs the `Setting history` with the order of package overrides to help you resolve the errors.

- *Value Override Violations*
 - *Example: Ambiguity Violation Error Message*
 - *Example: Priority Violation Error Message*
- *Flash Area Violations*
 - *Example: Undefined Flash Area Error Message*

- Example: *Multiple Flash Area Assignment Error Message*
- *Restriction Violations*
 - Example: *\$notnull Restriction Violation Error Message*
 - Example: *Expression Restriction Violation Error Message*
- *Task Priority Violations*
 - Example: *Duplicate Task Priority Assignment Error Message*
 - Example: *Invalid Task Priority Error Message*
- *Duplicate System Configuration Setting Definition*
- *Override of Undefined System Configuration Setting*
 - Example: *Ignoring Override of Undefined Setting Message*
 - *BSP Package Overrides Undefined Configuration Settings*

Note: The `newt target config <target-name>` command also detects errors and outputs error messages at the top of the command output. The command outputs the package setting definitions and values after it outputs the error messages. It is easy to miss the error messages at the top.

Value Override Violations

The newt tool uses package priorities to resolve override conflicts. It uses the value override from the highest priority package when multiple packages override the same setting. Newt checks for the following override violations:

- Ambiguity Violation - Two packages of the same priority override a setting with different values. And no higher priority package overrides the setting.
- Priority Violation - A package overrides a setting defined by a package with higher or equal priority.

Note: A package may override the default value for a setting that it defines. For example, a package defines a setting with a default value but needs to conditionally override the value based on another setting value.

Example: Ambiguity Violation Error Message

The following example shows the error message that newt outputs for an ambiguity violation:

```
Error: Syscfg ambiguities detected:  
Setting: LOG_NEWTMGR, Packages: [apps/slinky, apps/splitty]  
Setting history (newest -> oldest):  
LOG_NEWTMGR: [apps/splitty:0, apps/slinky:1, sys/log/full:0]
```

The above error occurs because the `apps/slinky` and `apps/splitty` packages in the split image target both override the same setting with different values. The `apps/slinky` package sets the `sys/log/full` package `LOG_NEWTMGR` setting to 1, and the `apps/splitty` package sets the setting to 0. The overrides are ambiguous because both are app packages and have the same priority. The following are excerpts of the defintion and the two overrides from the `syscfg.yaml` files that cause the error:

```
#Package: sys/log/full  
syscfg.defs:  
  LOG_NEWTMGR:
```

(continues on next page)

(continued from previous page)

```

description: 'Enables or disables newtmgr command tool logging'
value: 0

#Package: apps/slinky
syscfg.vals:
  LOG_NEWTMGR: 1

#Package: apps/splitty
syscfg.vals:
  LOG_NEWTMGR: 0

```

Example: Priority Violation Error Message

The following example shows the error message that newt outputs for a priority violation where a package tries to change the setting that was defined by another package at the same priority level:

```

Error: Priority violations detected (Packages can only override settings defined by
↳ packages of lower priority):
  Package: mgmt/newtmgr overriding setting: LOG_NEWTMGR defined by sys/log/full

Setting history (newest -> oldest):
  LOG_NEWTMGR: [sys/log/full:0]

```

The above error occurs because the mgmt/newtmgr lib package overrides the LOG_NEWTMGR setting that the sys/log/full lib package defines. The following are excerpts of the definition and the override from the `syscfg.yml` files that cause this error:

```

#Package: sys/log/full
syscfg.defs:
  LOG_NEWTMGR:
    description: 'Enables or disables newtmgr command tool logging'
    value: 0

#Package: mgmt/newtmgr
syscfg.vals:
  LOG_NEWTMGR: 1

```

Flash Area Violations

For `flash_owner` type setting definitions, newt checks for the following violations:

- An undefined flash area is assigned to a setting.
- A flash area is assigned to multiple settings.

Example: Undefined Flash Area Error Message

The following example shows the error message that newt outputs for an undefined flash area.

```
Building target targets/sim_slinky
Error: Flash errors detected:
    Setting REBOOT_LOG_FLASH_AREA specifies unknown flash area: FLASH_AREA_NOEXIST

Setting history (newest -> oldest):
    REBOOT_LOG_FLASH_AREA: [hw/bsp/native:FLASH_AREA_NOEXIST, sys/reboot:]
```

The above error occurs because the hw/bsp/native package assigns the undefined FLASH_AREA_NOEXIST flash area to the sys/reboot package REBOOT_LOG_FLASH_AREA setting. The following are excerpts of the definition and the override from the syscfg.yml files that cause the error:

```
#Package: sys/reboot
syscfg.defs:
    REBOOT_LOG_FLASH_AREA:
        description: 'Flash Area to use for reboot log.'
        type: flash_owner
        value:

#Package: hw/bsp/native
syscfg.vals:
    REBOOT_LOG_FLASH_AREA: FLASH_AREA_NOEXIST
```

Example: Multiple Flash Area Assignment Error Message

The following example shows the error message that newt outputs when multiple settings are assigned the same flash area:

```
Error: Flash errors detected:
    Multiple flash_owner settings specify the same flash area
        settings: REBOOT_LOG_FLASH_AREA, CONFIG_FCB_FLASH_AREA
        flash area: FLASH_AREA_NFFS

Setting history (newest -> oldest):
    CONFIG_FCB_FLASH_AREA: [hw/bsp/native:FLASH_AREA_NFFS, sys/config:]
    REBOOT_LOG_FLASH_AREA: [apps/slinky:FLASH_AREA_NFFS, sys/reboot:]
```

The above error occurs because the hw/bsp/native package assigns the FLASH_AREA_NFFS flash area to the sys/config/ package CONFIG_FCB_FLASH_AREA setting, and the apps/slinky package also assigns FLASH_AREA_NFFS to the sys/reboot package REBOOT_LOG_FLASH_AREA setting. The following are excerpts of the two definitions and the two overrides from the syscfg.yml files that cause the error:

```
# Package: sys/config
syscfg.defs.CONFIG_FCB:
    CONFIG_FCB_FLASH_AREA:
        description: 'The flash area for the Config Flash Circular Buffer'
        type: 'flash_owner'
        value:
```

(continues on next page)

(continued from previous page)

```
# Package: sys/reboot
syscfg.defs:
    REBOOT_LOG_FLASH_AREA:
        description: 'The flash area for the reboot log'
        type: 'flash_owner'
        value:

#Package: hw/bsp/native
syscfg.vals:
    CONFIG_FCB_FLASH_AREA: FLASH_AREA_NFFS

#Package: apps/slinky
syscfg.vals:
    REBOOT_LOG_FLASH_AREA: FLASH_AREA_NFFS
```

Restriction Violations

For setting definitions with `restrictions` specified, newt checks for the following violations:

- A setting with a `$notnull` restriction does not have a value.
- For a setting with expression restrictions, some required setting values in the expressions evaluate to false.

Example: `$notnull` Restriction Violation Error Message

The following example shows the error message that newt outputs when a setting with `$notnull` restriction does not have a value:

```
Error: Syscfg restriction violations detected:
NFFS_FLASH_AREA must not be null

Setting history (newest -> oldest):
NFFS_FLASH_AREA: [fs/nffs:]
```

The above error occurs because the `fs/nffs` package defines the `NFFS_FLASH_AREA` setting with a `$notnull` restriction and no packages override the setting. The following is an excerpt of the definition in the `syscfg.yml` file that causes the error:

```
#Package: fs/nffs
syscfg.defs:
    NFFS_FLASH_AREA:
        description: 'The flash area to use for the Newtron Flash File System'
        type: 'flash_owner'
        value:
        restrictions:
            - $notnull
```

Example: Expression Restriction Violation Error Message

The following example shows the error message that newt outputs for an expression restriction violation:

```
Error: Syscfg restriction violations detected:  
    CONFIG_FCB=1 requires CONFIG_FCB_FLASH_AREA be set, but CONFIG_FCB_FLASH_AREA=  
  
Setting history (newest -> oldest):  
    CONFIG_FCB: [targets/sim_slinky:1, sys/config:0]  
    CONFIG_FCB_FLASH_AREA: [sys/config:]
```

The above error occurs because the `sys/config` package defines the `CONFIG_FCB` setting with a restriction that when set, requires that the `CONFIG_FCB_FLASH_AREA` setting must also be set. The following are excerpts of the definition and the override from the `syscfg.yml` files that cause the error:

```
# Package: sys/config  
syscfg.defs:  
    CONFIG_FCB:  
        description: 'Uses Config Flash Circular Buffer'  
        value: 0  
        restrictions:  
            - '!CONFIG_NFFS'  
            - 'CONFIG_FCB_FLASH_AREA'  
  
# Package: targets/sim_slinky  
syscfg.vals:  
    CONFIG_FCB: 1
```

Task Priority Violations

For `task_priority` type setting definitions, newt checks for the following violations:

- A task priority number is assigned to multiple settings.
- The task priority number is greater than 239.

Example: Duplicate Task Priority Assignment Error Message

The following example shows the error message that newt outputs when a task priority number is assigned to multiple settings.

Note: The settings used in this example are not actual `apps/slinky` and `sys/shell` settings. These settings are created for this example because currently only one Mynewt package defines a `task_priority` type setting.

```
Error: duplicate priority value: setting1=SHELL_TASK_PRIORITY setting2=SLINKY_TASK_  
    ↵PRIORITY pkg1=apps/slinky pkg2=sys/shell value=1
```

The above error occurs because the `apps/slinky` package defines a `SLINKY_TASK_PRIORITY` setting with a default task priority of 1 and the `sys/shell` package also defines a `SHELL_TASK_PRIORITY` setting with a default task priority of 1.

Example: Invalid Task Priority Error Message

The following example shows the error message that newt outputs when a setting is assigned an invalid task priority value:

```
Error: invalid priority value: value too great (> 239); setting=SLINKY_TASK_PRIORITY
  ↵value=240 pkg=apps/slinky
```

The above error occurs because the `apps/slinky` package defines the `SLINKY_TASK_PRIORITY` setting with 240 for the default task priority value.

Note: Newt does not output the `Setting history` with task priority violation error messages.

Duplicate System Configuration Setting Definition

A setting definition must be unique. Newt checks that only one package in the target defines a setting. The following example shows the error message that newt outputs when multiple packages define the `LOG_NEWTMGR` setting:

```
Error: setting LOG_NEWTMGR redefined
```

Note: Newt does not output the `Setting history` with duplicate setting error messages.

Override of Undefined System Configuration Setting

The `newt build` command ignores overrides of undefined system configuration settings. The command does not print a warning when you run it with the default log level. If you override a setting and the value is not assigned to the setting, you may have misspelled the setting name or a package no longer defines the setting. You have two options to troubleshoot this problem:

- Run the `newt target config show` command to see the configuration setting definitions and overrides.
- Run the `newt build -ldebug` command to build your target with DEBUG log level.

Note: The `newt build -ldebug` command generates lots of output and we recommend that you use the `newt target config show` command option.

Example: Ignoring Override of Undefined Setting Message

The following example shows that the `apps/slinky` application overrides the `LOG_NEWTMGR` setting but omits the `T` as an example of an error and overrides the misspelled `LOG_NEWMGR` setting. Here is an excerpt from its `syscfg.yml` file:

```
#package: apps/slinky
syscfg.vals:
    # Enable the shell task.
    SHELL_TASK: 1
    ...
    # Enable newtmgr commands.
    STATS_NEWTMGR: 1
    LOG_NEWMGR: 1
```

The `newt target config show slinky_sim` command outputs the following WARNING message:

```
2017/02/18 17:19:12.119 [WARNING] Ignoring override of undefined settings:  
2017/02/18 17:19:12.119 [WARNING]      LOG_NEWMGR  
2017/02/18 17:19:12.119 [WARNING]      NFFS_FLASH_AREA  
2017/02/18 17:19:12.119 [WARNING] Setting history (newest -> oldest):  
2017/02/18 17:19:12.119 [WARNING]      LOG_NEWMGR: [apps/slinky:1]  
2017/02/18 17:19:12.119 [WARNING]      NFFS_FLASH_AREA: [hw/bsp/native:FLASH_AREA_NFFS]
```

The newt build -ldebug slinky_sim command outputs the following DEBUG message:

```
2017/02/18 17:06:21.451 [DEBUG] Ignoring override of undefined settings:  
2017/02/18 17:06:21.451 [DEBUG]      LOG_NEWMGR  
2017/02/18 17:06:21.451 [DEBUG]      NFFS_FLASH_AREA  
2017/02/18 17:06:21.451 [DEBUG] Setting history (newest -> oldest):  
2017/02/18 17:06:21.451 [DEBUG]      LOG_NEWMGR: [apps/slinky:1]  
2017/02/18 17:06:21.451 [DEBUG]      NFFS_FLASH_AREA: [hw/bsp/native:FLASH_AREA_NFFS]
```

BSP Package Overrides Undefined Configuration Settings

You might see a warning that indicates your application's BSP package is overriding some undefined settings. As you can see from the previous example, the WARNING message shows that the hw/bsp/native package is overriding the undefined NFFS_FLASH_AREA setting. This is not an error because of the way a BSP package defines and assigns its flash areas to packages that use flash memory.

A BSP package defines, in its `bsp.yml` file, a flash area map of the flash areas on the board. A package that uses flash memory must define a flash area configuration setting name. The BSP package overrides the package's flash area setting with one of the flash areas from its flash area map. A BSP package overrides the flash area settings for all packages that use flash memory because it does not know the packages that an application uses. When an application does not include one of these packages, the flash area setting for the package is undefined. You will see a message that indicates the BSP package overrides this undefined setting.

Here are excerpts from the hw/bsp/native package's `bsp.yml` and `syscfg.yml` files for the `slinky_sim` target. The BSP package defines the flash area map in its `bsp.yml` file and overrides the flash area settings for all packages in its `syscfg.yml` file. The `slinky_sim` target does not use the `fs/nffs` package which defines the `NFFS_FLASH_AREA` setting. Newt warns that the `hw/bsp/native` packages overrides the undefined `NFFS_FLASH_AREA` setting.

```
# hw/bsp/native bsp.yml  
bsp.flash_map:  
  areas:  
    # System areas.  
    FLASH_AREA_BOOTLOADER:  
      device: 0  
      offset: 0x00000000  
      size: 16kB  
  
    ...  
  
    FLASH_AREA_IMAGE_SCRATCH:  
      device: 0  
      offset: 0x000e0000  
      size: 128kB  
  
    # User areas.  
    FLASH_AREA_REBOOT_LOG:
```

(continues on next page)

(continued from previous page)

```

user_id: 0
device: 0
offset: 0x00004000
size: 16kB
FLASH_AREA_NFFS:
    user_id: 1
    device: 0
    offset: 0x00008000

# hw/bsp/native syscfg.yml
syscfg.vals:
    NFFS_FLASH_AREA: FLASH_AREA_NFFS
    CONFIG_FCB_FLASH_AREA: FLASH_AREA_NFFS
    REBOOT_LOG_FLASH_AREA: FLASH_AREA_REBOOT_LOG

```

6.13 System Initialization and System Shutdown

Mynewt allows a package to designate startup and shutdown functions. These functions are called automatically by the OS using facilities called *sysinit* and *sysdown*.

This guide:

- Assumes you have read the *Concepts* section that describes the Mynewt package hierarchy and its use of the *pkg.yml* and *syscfg.yml* files.
- Assumes you have read the Mynewt *Theory of Operations* and are familiar with how newt determines package dependencies for your target build.

- *System Initialization*
 - *Specifying Package Initialization Functions*
 - *Generated sysinit_app() Function*
- *System Shutdown*

6.13.1 System Initialization

During system startup, Mynewt creates a default event queue and a main task to process events from this queue. You can override the `OS_MAIN_TASK_PRIO` and `OS_MAIN_TASK_STACK_SIZE` setting values defined by the `kernel/os` package to specify different task priority and stack size values.

Your application's `main()` function executes in the context of the main task and must perform the following:

- At the start of `main()`, call the Mynewt `sysinit()` function to initialize the packages before performing any other processing.
- At the end of `main()`, wait for and dispatch events from the default event queue in an infinite loop.

Note: You must include the `sysinit/sysinit.h` header file to access the `sysinit()` function.

Here is an example of a `main()` function:

```
int
main(int argc, char **argv)
{
    /* First, call sysinit() to perform the system and package initialization */
    sysinit();

    /* ... other application initialization processing ... */

    /* Last, process events from the default event queue. */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }
    /* main never returns */
}
```

Specifying Package Initialization Functions

The `sysinit()` function calls the `sysinit_app()` function to perform system initialization for the packages in the target. You can, optionally, specify one or more package initialization functions that `sysinit_app()` calls to initialize a package.

A package initialization function must have the following prototype:

```
void init_func_name(void)
```

Sysinit functions are not expected to fail. If a sysinit function encounters an unrecoverable failure, it should invoke one of the `SYSINIT_PANIC` macros. By default, these macros trigger a crash, but the behavior can be overridden by configuring the panic function with `sysinit_panic_set()`.

Package initialization functions are called in stages to ensure that lower priority packages are initialized before higher priority packages. A stage is an integer value, 0 or higher, that specifies when an initialization function is called. Mynewt calls the package initialization functions in increasing stage number order. When multiple init functions have the same stage number, they are called in lexicographic order (by function name).

You use the `pkg.init` parameter in the `pkg.yml` file to specify an initialization function and the stage number to call the function. You can specify multiple initialization functions with a different stage number for each function for the parameter values. This feature allows packages with interdependencies to perform initialization in multiple stages.

The `pkg.init` parameter has the following syntax in the `pkg.yml` file:

```
pkg.init:
    pkg_init_func1_name: pkg_init_func1_stage
    pkg_init_func2_name: pkg_init_func2_stage
    ...
    pkg_init_funcN_name: pkg_init_funcN_stage
```

where `pkg_init_func#_name` is the C function name of an initialization function, and `pkg_init_func#_stage` is an integer value, 0 or higher, that indicates the stage when the `pkg_init_func#_name` function is called.

Generated sysinit_app() Function

The newt tool processes the `pkg.init` parameters in all the `pkg.yml` files for a target, generates the `sysinit_app()` function in the `<target-path>/generated/src/<target-name>-sysinit_app.c` file, and includes the file in the build. Here is an example `sysinit_app()` function:

```
/***
 * This file was generated by Apache Newt (incubating) version: 1.0.0-dev
 */

#ifndef SPLIT_LOADER

void split_app_init(void);
void os_pkg_init(void);
void imgmgr_module_init(void);

/* ... */

void stats_module_init(void);

void
sysinit_app(void)
{
    /*** Stage 0 */
    /* 0.0: kernel/os */
    os_pkg_init();

    /*** Stage 2 */
    /* 2.0: sys/flash_map */
    flash_map_init();

    /*** Stage 10 */
    /* 10.0: sys/stats/full */
    stats_module_init();

    /*** Stage 20 */
    /* 20.0: sys/console/full */
    console_pkg_init();

    /*** Stage 100 */
    /* 100.0: sys/log/full */
    log_init();
    /* 100.1: sys/mfg */
    mfg_init();

    /* ... */

    /*** Stage 300 */
    /* 300.0: sys/config */
    config_pkg_init();

    /*** Stage 500 */
}
```

(continues on next page)

(continued from previous page)

```

/* 500.0: sys/id */
id_init();

/* 500.1: sys/shell */
shell_init();

/* ... */

/* 500.4: mgmt/imgmgr */
imgmgr_module_init();

/* Stage 501 */
/* 501.0: mgmt/newtmgr/transport/nmgr_shell */
nmgr_shell_pkg_init();
}

#endif

```

6.13.2 System Shutdown

A Mynewt package can specify a sequence of system shutdown (“sysdown”) function calls. As with sysinit, a stage number is associated with each function. On shutdown, sysdown functions are executed in ascending order of stage number. In the case of a tie, functions are executed in lexicographic order (by function name).

The sysdown procedure is only performed for a controlled shutdown. It is executed when the system processes a newtmgr “reset” command, for example. It is not be executed when the system crashes, browns out, or restarts due to the hardware watchdog.

Sysdown functions are specified in a `pkg.yml` file using the `pkg.down` key. They use the following function type:

```
int func(int reason)
```

The `reason` parameter is currently unused and should be ignored.

Each shutdown callback returns one of the following codes:

- `SYSDOWN_COMPLETE`
- `SYSDOWN_IN_PROGRESS`

If a sysdown function is able to complete its work synchronously, it should return `SYSDOWN_COMPLETE`.

The “in progress” case is a bit more complicated. Sysdown functions should not block on the current task (e.g., they should not tell the default task to do something and then wait for the result). Blocking like this will result in a deadlock since the current task is waiting for the sysdown function to complete.

Instead, if a sysdown function needs to do extra work in the current task, it should do so asynchronously. That is, it should enqueue an event to the task’s event queue, then return `SYSDOWN_IN_PROGRESS`. When the sysdown procedure eventually completes, the task should call `sysdown_release`.

When all sysdown procedures have completed, Mynewt proceeds to reset the device. If it takes longer than `MYNEWT_VAL(SYSDOWN_TIMEOUT_MS)` milliseconds (default: 10s) for all shutdown procedures to complete, a crash is triggered and the device resets.

As an example, the NimBLE BLE host package configures a sysdown function that terminates all open connections. Its `pkg.yml` contains the following map:

```
pkg.down:
  ble_hs_shutdown: 200
```

and `ble_hs_shutdown` is defined as follows:

```
int
ble_hs_shutdown(int reason)
{
    int rc;

    /* Ensure this function only gets called by sysdown. */
    SYSDOWN_ASSERT_ACTIVE();

    /* Initiate a host stop procedure. */
    rc = ble_hs_stop(&ble_hs_shutdown_stop_listener, ble_hs_shutdown_stop_cb,
                     NULL);
    switch (rc) {
        case 0:
            /* Stop initiated. Wait for result to be reported asynchronously. */
            return SYSDOWN_IN_PROGRESS;

        case BLE_HS_EBUSY:
            /* Already stopping. Wait for result to be reported asynchronously. */
            return SYSDOWN_IN_PROGRESS;

        case BLE_HS_EALREADY:
            /* Already stopped. Shutdown complete. */
            return SYSDOWN_COMPLETE;

        default:
            BLE_HS_LOG(ERROR, "ble_hs_shutdown: failed to stop host; rc=%d\n", rc);
            return SYSDOWN_COMPLETE;
    }
}
```

In the cases where this function returns `SYSDOWN_IN_PROGRESS`, the host eventually calls `sysdown_release` when the procedure completes.

6.14 Build-Time Hooks

A package specifies custom commands in its `pkg.yml` file. There are three types of commands:

1. `pre_build_cmds` (run before the build)
2. `pre_link_cmds` (run after compilation, before linking)
3. `post_link_cmds` (run after linking)

6.14.1 Example

Example (apps/blinky/pkg.yml):

```
pkg.pre_build_cmds:  
    scripts/pre_build1.sh: 100  
    scripts/pre_build2.sh: 200  
  
pkg.pre_link_cmds:  
    scripts/pre_link.sh: 500  
  
pkg.post_link_cmds:  
    scripts/post_link.sh: 100
```

For each command, the string on the left specifies the command to run. The number on the right indicates the command's relative ordering. All paths are relative to the project root.

When newt builds this example, it performs the following sequence:

- scripts/pre_build1.sh
- scripts/pre_build2.sh
- [compile]
- scripts/pre_link.sh
- [link]
- scripts/post_link.sh

If other packages specify custom commands, those commands would also be executed during the above sequence. For example, if another package specifies a pre build command with an ordering of 150, that command would run immediately after pre_build1.sh. In the case of a tie, the commands are run in lexicographic order (by path).

All commands are run from the project's base directory. In the above example, the `scripts` directory is a sibling of `targets`.

6.14.2 Custom Build Inputs

A custom pre-build or pre-link command can produce files that get fed into the current build.

Pre-build commands can generate any of the following:

1. .c files for newt to compile.
2. .a files for newt to link.
3. .h files that any package can include.

Pre-link commands can only generate .a files.

.c and .a files should be written to the `$MYNEWT_USER_SRC_DIR` environment variable (defined by newt), or any subdirectory within.

.h files should be written to `$MYNEWT_USER_INCLUDE_DIR`. The directory structure used here is directly reflected by the includer. E.g., if a script writes to `$MYNEWT_USER_INCLUDE_DIR/foo/bar.h`, then a source file can include this header with:

```
#include "foo/bar.h"
```

6.14.3 Details

Environment Variables

In addition to the usual environment variables defined for debug and download scripts, newt defines the following env vars for custom commands:

Environment variable	Description	Notes
MYNEWT_APP_BIN_DIR	The directory where the current target's binary gets written.	
MYNEWT_PKG_BIN_ARCH	The path to the current package's .a file.	
MYNEWT_PKG_BIN_DIR	The directory where the current package's .o and .a files get written.	
MYNEWT_PKG_NAME	The full name of the current package.	
MYNEWT_USER_INCLUDE	Path where globally-accessible headers get written.	Pre-build only.
MYNEWT_USER_SRC_DIR	Path where build inputs get written.	Pre-build and pre-link only.
MYNEWT_USER_WORK_DIR	Shared temp directory; used for communication between commands.	

These environment variables are defined for each process that a custom command runs in. They are *not* defined in the newt process itself. So, the following snippet will not produce the expected output:

BAD Example (apps/blinky/pkg.yml):

```
pkg.pre_cmds:
  'echo $MYNEWT_USER_SRC_DIR': 100
```

You can execute `sh` here instead if you need access to the environment variables, but it is probably saner to just use a script.

Detect Changes in Custom Build Inputs

To avoid unnecessary rebuilds, newt detects if custom build inputs have changed since the previous build. If none of the inputs have changed, then they do not get rebuilt. If any of them have changed, they all get rebuilt.

The `$MYNEWT_USER_[...]` directories are actually temp directories. After the pre-build commands have run, newt compares the contents of the temp directory with those of the actual user directory. If any differences are detected, newt replaces the user directory with the temp directory, triggering a rebuild of its contents. The same procedure is used for pre-link commands.

Paths

Custom build inputs get written to the following directories:

- bin/targets/<target>/user/pre_build/src
- bin/targets/<target>/user/pre_build/include
- bin/targets/<target>/user/pre_link/src

Custom commands should not write to these directories. They should use the `$MYNEWT_USER_[...]` environment variables instead.

6.15 File System Abstraction

Mynewt provides a file system abstraction layer (`fs/fs`) to allow client code to be file system agnostic. By accessing the file system via the `fs/fs` API, client code can perform file system operations without being tied to a particular implementation. When possible, library code should use the `fs/fs` API rather than accessing the underlying file system directly.

- *Description*
- *Support for multiple filesystems*
 - `struct disk_ops`
- *Thread Safety*
- *Header Files*
- *Data Structures*
- *Examples*
- *API*

6.15.1 Description

Applications should aim to minimize the amount of code which depends on a particular file system implementation. When possible, only depend on the `fs/fs` package. In terms of the Mynewt hierarchy, an **app** package must depend on a specific file system package, while **library** packages should only depend on `fs/fs`.

Applications wanting to access a filesystem are required to include the necessary packages in their applications `pkg.yml` file. In the following example, the *Newtron Flash File System* is used.

```
# repos/apache-mynewt-core/apps/slinky/pkg.yml

pkg.name: repos/apache-mynewt-core/apps/slinky
pkg.deps:
  - "@apache-mynewt-core/fs/fs"          # include the file operations interfaces
  - "@apache-mynewt-core/fs/nffs"         # include the NFFS filesystem implementation
```

```
# repos/apache-mynewt-core/apps/slinky/syscfg.yml
# [...]
# Package: apps/<example app>
# [...]
  CONFIG_NFFS: 1  # initialize and configure NFFS into the system
#  NFFS_DETECT_FAIL: 1  # Ignore NFFS detection issues
#  NFFS_DETECT_FAIL: 2  # Format a new NFFS file system on failure to detect

# [...]
```

Consult the `nffs` *documentation* for a more detailed explanation of `NFFS_DETECT_FAIL`

Code which uses the file system after the system has been initialized need only depend on `fs/fs`. For example, the `libs/imgmgr` package is a library which provides firmware upload and download functionality via the use of a file system. This library is only used after the system has been initialized, and therefore only depends on the `fs/fs` package.

```
# repos/apache-mynewt-core/libs/imgmgr/pkg.yml
pkg.name: libs/imgmgr
pkg.deps:
- "@apache-mynewt-core/fs/fs"

# [...]
```

The `libs/imgmgr` package uses the `fs/fs` API for all file system operations.

6.15.2 Support for multiple filesystems

When using a single filesystem/disk, it is valid to provide paths in the standard unix way, eg, `/<dir-name>/<file-name>`. When trying to run more than one filesystem or a single filesystem in multiple devices simultaneously, an extra name has to be given to the disk that is being used. The abstraction for that was added as the `fs/disk` package which is a dependency of `fs/fs`. It adds the following extra user function:

```
int disk_register(const char *disk_name, const char *fs_name, struct disk_ops *dops)
```

As an example of usage:

```
disk_register("mmc0", "fatfs", &mmc_ops);
disk_register("flash0", "nffs", NULL);
```

This registers the name `mmc0` to use `fatfs` as the filesystem and `mmc_ops` for the low-level disk driver and also registers `flash0` to use `nffs`. `nffs` is currently strongly bound to the `hal_flash` interface, ignoring any other possible `disk_ops` given.

struct disk_ops

To support a new low-level disk interface, the `struct disk_ops` interface must be implemented by the low-level driver. Currently only `read` and `write` are effectively used (by `fatfs`).

```
struct disk_ops {
    int (*read)(uint8_t, uint32_t, void *, uint32_t);
    int (*write)(uint8_t, uint32_t, const void *, uint32_t);
    int (*ioctl)(uint8_t, uint32_t, void *);
    SLIST_ENTRY(disk_ops) sc_next;
}
```

6.15.3 Thread Safety

All `fs/fs` functions are thread safe.

6.15.4 Header Files

All code which uses the fs/fs package needs to include the following header:

```
#include "fs/fs.h"
```

6.15.5 Data Structures

All fs/fs data structures are opaque to client code.

```
struct fs_file;
struct fs_dir;
struct fs_dirent;
```

6.15.6 Examples

Example 1 below opens the file /settings/config.txt for reading, reads some data, and then closes the file.

```
int
read_config(void)
{
    struct fs_file *file;
    uint32_t bytes_read;
    uint8_t buf[16];
    int rc;

    /* Open the file for reading. */
    rc = fs_open("/settings/config.txt", FS_ACCESS_READ, &file);
    if (rc != 0) {
        return -1;
    }

    /* Read up to 16 bytes from the file. */
    rc = fs_read(file, sizeof buf, buf, &bytes_read);
    if (rc == 0) {
        /* buf now contains up to 16 bytes of file data. */
        console_printf("read %u bytes\n", bytes_read)
    }

    /* Close the file. */
    fs_close(file);

    return rc == 0 ? 0 : -1;
}
```

Example 2 below iterates through the contents of a directory, printing the name of each child node. When the traversal is complete, the code closes the directory handle.

```
int
traverse_dir(const char *dirname)
{
```

(continues on next page)

(continued from previous page)

```

struct fs_dirent *dirent;
struct fs_dir *dir;
char buf[64];
uint8_t name_len;
int rc;

rc = fs_opendir(dirname, &dir);
if (rc != 0) {
    return -1;
}

/* Iterate through the parent directory, printing the name of each child
* entry. The loop only terminates via a function return.
*/
while (1) {
    /* Retrieve the next child node. */
    rc = fs_readdir(dir, &dirent);
    if (rc == FS_ENOENT) {
        /* Traversal complete. */
        return 0;
    } else if (rc != 0) {
        /* Unexpected error. */
        return -1;
    }

    /* Read the child node's name from the file system. */
    rc = fs_dirent_name(dirent, sizeof buf, buf, &name_len);
    if (rc != 0) {
        return -1;
    }

    /* Print the child node's name to the console. */
    if (fs_dirent_is_dir(dirent)) {
        console_printf(" dir: ");
    } else {
        console_printf("file: ");
    }
    console_printf("%s\n", buf);
}
}

```

Example 3 below demonstrates creating a series of nested directories.

```

int
create_path(void)
{
    int rc;

    rc = fs_mkdir("/data");
    if (rc != 0) goto err;

    rc = fs_mkdir("/data/logs");

```

(continues on next page)

(continued from previous page)

```

if (rc != 0) goto err;

rc = fs_mkdir("/data/logs/temperature");
if (rc != 0) goto err;

rc = fs_mkdir("/data/logs/temperature/current");
if (rc != 0) goto err;

return 0;

err:
/* Clean up the incomplete directory tree, if any. */
fs_unlink("/data");
return -1;
}

```

Example 4 below demonstrates reading a small text file in its entirety and printing its contents to the console.

```

int
print_status(void)
{
    uint32_t bytes_read;
    uint8_t buf[16];
    int rc;

    /* Read up to 15 bytes from the start of the file. */
    rc = fsutil_read_file("/cfg/status.txt", 0, sizeof buf - 1, buf,
                          &bytes_read);
    if (rc != 0) return -1;

    /* Null-terminate the string just read. */
    buf[bytes_read] = '\0';

    /* Print the file contents to the console. */
    console_printf("%s\n", buf);

    return 0;
}

```

Example 5 creates a 4-byte file.

```

int
write_id(void)
{
    int rc;

    /* Create the parent directory. */
    rc = fs_mkdir("/cfg");
    if (rc != 0 && rc != FS_EALREADY) {
        return -1;
    }
}

```

(continues on next page)

(continued from previous page)

```
/* Create a file and write four bytes to it. */
rc = fsutil_write_file("/cfg/id.txt", "1234", 4);
if (rc != 0) {
    return -1;
}

return 0;
}
```

6.15.7 API

Defines

FS_ACCESS_READ

File access flags.

FS_ACCESS_WRITE

FS_ACCESS_APPEND

FS_ACCESS_TRUNCATE

FS_EOK

File access return codes.

FS_ECORRUPT

FS_EHW

FS_EOFFSET

FS_EINVAL

FS_ENOMEM

FS_ENOENT

FS_EEMPTY

FS_EFULL

FS_EUNEXP

FS_EOS

FS_EEXIST

FS_EACCESS

FS_EUNINIT

FS_MGMT_ID_FILE

FS_MGMT_MAX_NAME

Functions

```
int fs_open(const char *filename, uint8_t access_flags, struct fs_file**)
int fs_close(struct fs_file*)
int fs_read(struct fs_file*, uint32_t len, void *out_data, uint32_t *out_len)
int fs_write(struct fs_file*, const void *data, int len)
int fs_seek(struct fs_file*, uint32_t offset)
uint32_t fs_getpos(const struct fs_file*)
int fs_filelen(const struct fs_file*, uint32_t *out_len)
int fs_unlink(const char *filename)
int fs_rename(const char *from, const char *to)
int fs_mkdir(const char *path)
int fs_opendir(const char *path, struct fs_dir**)
int fs_readdir(struct fs_dir*, struct fs_dirent**)
int fs_closedir(struct fs_dir*)
int fs_dirent_name(const struct fs_dirent*, size_t max_len, char *out_name, uint8_t *out_name_len)
int fs_dirent_is_dir(const struct fs_dirent*)
int fs_flush(struct fs_file*)
```

Functions

```
int fsutil_read_file(const char *path, uint32_t offset, uint32_t len, void *dst, uint32_t *out_len)
int fsutil_write_file(const char *path, const void *data, uint32_t len)
```

6.15.8 Newtron Flash Filesystem (nffs)

Mynewt includes the Newtron Flash File System (nffs). This file system is designed with two priorities that makes it suitable for embedded use:

- Minimal RAM usage
- Reliability

Mynewt also provides an abstraction layer API (fs) to allow you to swap out nffs with a different file system of your choice.

Description

Areas

At the top level, an nffs disk is partitioned into *areas*. An area is a region of disk with the following properties:

1. An area can be fully erased without affecting any other areas.
2. Writing to one area does not restrict writes to other areas.

Regarding property 1: Generally, flash hardware divides its memory space into “blocks.” When erasing flash, entire blocks must be erased in a single operation; partial erases are not possible.

Regarding property 2: Furthermore, some flash hardware imposes a restriction with regards to writes: writes within a block must be strictly sequential. For example, if you wish to write to the first 16 bytes of a block, you must write bytes 1 through 15 before writing byte 16. This restriction only applies at the block level; writes to one block have no effect on what parts of other blocks can be written.

Thus, each area must comprise a discrete number of blocks.

Initialization

As part of overall system initialization, mynewt re-initialized the filesystem as follows:

1. Restores an existing file system via detection.
2. Creates a new file system via formatting.

A typical initialization sequence is the following:

1. Detect an nffs file system in a specific region of flash.
2. If no file system was detected, if configured to do so, format a new file system in the same flash region.

Note that in the latter case, the behavior is controlled with a variable in the syscfg.yml file. If NFFS_DETECT_FAIL is set to 1, the system ignores NFFS filesystem detection issues, but unless a new filesystem is formatted manually, all filesystem access will fail. If NFFS_DETECT_FAIL is set to 2, the system will format a new filesystem - note however this effectively deletes all existing data in the NFFS flash areas.

Both methods require the user to describe how the flash memory should be divided into nffs areas. This is accomplished with an array of `struct nffs_area_desc` configured as part of the BSP configuration.

After nffs has been initialized, the application can access the file system via the file system abstraction layer.

Data Structures

The `fs/nffs` package exposes the following data structures:

Struct	Description
<code>struct nffs_area_desc</code>	Descriptor for a single nffs area.
<code>struct nffs_config</code>	Configuration struct for nffs.

Miscellaneous measures

- RAM usage:
 - 24 bytes per inode
 - 12 bytes per data block
 - 36 bytes per inode cache entry
 - 32 bytes per data block cache entry
- Maximum filename size: 256 characters (no null terminator required)
- Disallowed filename characters: / and \0

Internals

nffs implementation details can be found here:

- [nffs_internals](#)

Future enhancements

- Error correction.
- Encryption.
- Compression.

API

Defines

`NFFS_FILENAME_MAX_LEN`

`NFFS_MAX_AREAS`

Functions

```
int nffs_init(void)

int nffs_detect(const struct nffs_area_desc *area_descs)

int nffs_format(const struct nffs_area_desc *area_descs)

int nffs_misc_desc_from_flash_area(int idx, int *cnt, struct nffs_area_desc *nad)
```

Variables

struct *nffs_config* **nffs_config**

```
struct nffs_config
#include <nffs.h>
```

Public Members

uint32_t nc_num_inodes

Maximum number of inodes; default=1024.

uint32_t nc_num_blocks

Maximum number of data blocks; default=4096.

uint32_t nc_num_files

Maximum number of open files; default=4.

uint32_t nc_num_dirs

Maximum number of open directories; default=4.

uint32_t nc_num_cache_inodes

Inode cache size; default=4.

uint32_t nc_num_cache_blocks

Data block cache size; default=64.

struct *nffs_area_desc*

```
#include <nffs.h>
```

Public Members

uint32_t **nad_offset**

uint32_t **nad_length**

uint8_t **nad_flash_id**

6.15.9 The FAT File System

Mynewt provides an implementation of the FAT filesystem which is currently supported on MMC/SD cards.

Description

File Allocation Table (FAT) is a computer file system architecture and a family of industry-standard file systems utilizing it. The FAT file system is a legacy file system which is simple and robust. It offers good performance even in lightweight implementations, but cannot deliver the same performance, reliability and scalability as some modern file systems.

Configuration

`fatfs` configuration can be tweaked by editing `fs/fatfs/include/fatfs/ffconf.h`. The current configuration was chosen to minimize memory use and some options address limitations existing in the OS:

- Write support is enabled by default (can be disabled to minimize memory use).
- Long filename (up to 255) support is disabled.
- When writing files, time/dates are not persisted due to current lack of a standard `hal_rtc` interface.
- No unicode support. Vanilla config uses standard US codepage 437.
- Formatting of new volumes is disabled.
- Default number of volumes is configured to 1.

API

To include `fatfs` on a project just include it as a dependency in your project:

```
pkg.deps:  
  - "@apache-mynewt-core/fs/fatfs"
```

It can now be used through the standard file system abstraction functions as described in FS API.

Example

An example of using `fatfs` on a MMC card is provided on the [MMC](#) documentation.

6.15.10 Other File Systems

Libraries use Mynewt's file system abstraction layer (`fs/fs`) for all file operations. Because clients use an abstraction layer, the underlying file system can be swapped out without affecting client code. This page documents the procedure for plugging a custom file system into the Mynewt file system abstraction layer.

1. Specify `fs/fs` as a dependency of your file system package.

The file system package must register itself with the `fs/fs` package, so it must specify `fs/fs` as a dependency. As an example, part of the Newtron Flash File System (`nffs`) `pkg.yml` is reproduced below. Notice the first item in the `pkg.deps` list.

```
pkg.name: fs/nffs
pkg.deps:
- "@apache-mynewt-core/fs/fs"
- "@apache-mynewt-core/hw/hal"
- "@apache-mynewt-core/libs/os"
- "@apache-mynewt-core/libs/testutil"
- "@apache-mynewt-core/sys/log"
```

2. Register your package's API with the `fs/fs` interface.

The `fs/fs` package calls into the underlying file system via a collection of function pointers. To plug your file system into the `fs/fs` API, you must assign these function pointers to the corresponding routines in your file system package.

For example, `nffs` registers itself with `fs/fs` as follows (from `fs/nffs/src/nffs.c`):

```
static const struct fs_ops nffs_ops = {
    .f_open = nffs_open,
    .f_close = nffs_close,
    .f_read = nffs_read,
    .f_write = nffs_write,

    .f_seek = nffs_seek,
    .f_getpos = nffs_getpos,
    .f_filelen = nffs_file_len,

    .f_unlink = nffs_unlink,
    .f_rename = nffs_rename,
    .f_mkdir = nffs_mkdir,

    .f_opendir = nffs_opendir,
    .f_readdir = nffs_readdir,
    .f_closedir = nffs_closedir,

    .f_dirent_name = nffs_dirent_name,
    .f_dirent_is_dir = nffs_dirent_is_dir,
```

(continues on next page)

(continued from previous page)

```
.f_name = "nffs"  
};  
  
int  
nffs_init(void)  
{  
    /* [...] */  
    fs_register(&nffs_ops);  
}
```

Header Files

To gain access to `fs/fs`'s registration interface, include the following header:

```
#include "fs/fs_if.h"
```

6.15.11 Adding a new file system

This section describes the common interface provided by any filesystem.

API

Functions

`int fs_register(struct fs_ops *fops)`

Registers a new filesystem interface.

Parameters

- **fops** – filesystem operations table

Returns

0 on success, non-zero on failure

`struct fs_ops *fs_ops_try_unique(void)`

Will look for the number of registered filesystems and will return the fops if there is only one.

Returns

fops if there's only one registered filesystem, NULL otherwise.

`struct fs_ops *fs_ops_for(const char *name)`

Retrieve a filesystem's operations table.

Parameters

- **name** – Name of the filesystem to retrieve `fs_ops` for

Returns

valid pointer on success, NULL on failure

`struct fs_ops *fs_ops_from_container(struct fops_container *container)`

```
struct fs_ops
#include <fs_if.h>
```

Public Functions**SLIST_ENTRY (fs_ops) sc_next****Public Members**int (***f_open**)(const char *filename, uint8_t access_flags, struct fs_file **out_file)int (***f_close**)(struct fs_file *file)int (***f_read**)(struct fs_file *file, uint32_t len, void *out_data, uint32_t *out_len)int (***f_write**)(struct fs_file *file, const void *data, int len)int (***f_flush**)(struct fs_file *file)int (***f_seek**)(struct fs_file *file, uint32_t offset)uint32_t (***f_getpos**)(const struct fs_file *file)int (***f_filelen**)(const struct fs_file *file, uint32_t *out_len)int (***f_unlink**)(const char *filename)int (***f_rename**)(const char *from, const char *to)int (***f_mkdir**)(const char *path)int (***f_opendir**)(const char *path, struct fs_dir **out_dir)int (***f_readdir**)(struct fs_dir *dir, struct fs_dirent **out_dirent)int (***f_closedir**)(struct fs_dir *dir)int (***f_dirent_name**)(const struct fs_dirent *dirent, size_t max_len, char *out_name, uint8_t *out_name_len)int (***f_dirent_is_dir**)(const struct fs_dirent *dirent)

```
const char *f_name

struct fops_container
#include <fs_if.h>
```

Public Members

```
struct fs_ops *fops
```

6.16 Flash Circular Buffer (FCB)

Flash circular buffer provides an abstraction through which you can treat flash like a FIFO. You append entries to the end, and read data from the beginning.

- *Description*
- *Usage*
- *Data structures*
- *API*

6.16.1 Description

Elements in the flash contain the length of the element, the data within the element, and checksum over the element contents.

Storage of elements in flash is done in a FIFO fashion. When user requests space for the next element, space is located at the end of the used area. When user starts reading, the first element served is the oldest element in flash.

Elements can be appended to the end of the area until storage space is exhausted. User has control over what happens next; either erase oldest block of data, thereby freeing up some space, or stop writing new data until existing data has been collected. FCB treats underlying storage as an array of flash sectors; when it erases old data, it does this a sector at a time.

Elements in the flash are checksummed. That is how FCB detects whether writing element to flash completed ok. It will skip over entries which don't have a valid checksum.

6.16.2 Usage

To add an element to circular buffer:

- Call fcb_append() to get the location where data can be written. If this fails due to lack of space, you can call fcb_rotate() to make some. And then call fcb_append() again.
- Use flash_area_write() to write element contents.
- Call fcb_append_finish() when done. This completes the entry by calculating the checksum.

To read contents of the circular buffer:

- * Call fcb_walk() with a pointer to your callback function.
- * Within callback function copy in data from the element using flash_area_read(). You can tell when all data from within a sector has been read by monitoring returned element's area pointer. Then you can call fcb_rotate(), if you're done with that data.

Alternatively:

- * Call fcb_getnext() with 0 in element offset to get the pointer to oldest element.
- * Use flash_area_read() to read element contents.
- * Call fcb_getnext() with pointer to current element to get the next one. And so on.

6.16.3 Data structures

This data structure describes the element location in the flash. You would use it figure out what parameters to pass to flash_area_read() to read element contents. Or to flash_area_write() when adding a new element.

```
struct fcb_entry {
    struct flash_area *fe_area;
    uint32_t fe_elem_off;
    uint32_t fe_data_off;
    uint16_t fe_data_len;
};
```

Element	Description
fe_area	Pointer to info about the flash sector. Pass this to flash_area_x x() routines.
fe_elem_off	Byte offset from the start of the sector to beginning of element.
fe_data_off	Byte offset from start of the sector to beginning of element data. Pass this to to flash_area_x x() routines.
fe_data_len	Number of bytes in the element.

The following data structure describes the FCB itself. First part should be filled in by the user before calling fcb_init(). The second part is used by FCB for its internal bookkeeping.

```
struct fcb {
    /* Caller of fcb_init fills this in */
    uint32_t f_magic;           /* As placed on the disk */
    uint8_t f_version;          /* Current version number of the data */
    uint8_t f_sector_cnt;       /* Number of elements in sector array */
    uint8_t f_scratch_cnt;      /* How many sectors should be kept empty */
    struct flash_area *f_sectors; /* Array of sectors, must be contiguous */

    /* Flash circular buffer internal state */
    struct os_mutex f_mtx;      /* Locking for accessing the FCB data */
    struct flash_area *f_oldest;
    struct fcb_entry f_active;
    uint16_t f_active_id;
    uint8_t f_align;            /* writes to flash have to aligned to this */
};
```

Element	Description
f_magic	Magic number in the beginning of FCB flash sector. FCB uses this when determining whether sector contains valid data or not.
f_version	Current version number of the data. Also stored in flash sector header.
f_sector_cnt	Number of elements in the f_sectors array.
f_scratch_cnt	Number of sectors to keep empty. This can be used if you need to have scratch space for garbage collecting when FCB fills up.
f_sectors	Array of entries describing flash sectors to use.
f_mtx	Lock protecting access to FCBs internal data.
f_oldest	Pointer to flash sector containing the oldest data. This is where data is served when read is started.
f_active	Flash location where the newest data is. This is used by fcb_append() to figure out where the data should go to.
f_active_id	Flash sectors are assigned ever-increasing serial numbers. This is how FCB figures out where oldest data is on system restart.
f_align	Some flashes have restrictions on alignment for writes. FCB keeps a copy of this number for the flash here.

6.16.4 API

typedef int (***fcb_walk_cb**)(struct *fcb_entry* *loc, void *arg)

Walk over all entries in FCB.

cb gets called for every entry. If cb wants to stop the walk, it should return non-zero value.

Entry data can be read using flash_area_read(), using loc->fe_area, loc->fe_data_off, and loc->fe_data_len as arguments.

int **fcb_cache_init**(struct *fcb* *fcb, struct *fcb_entry_cache* *cache, int initial_entry_count)

Init FCB cache that can be used for walking back optimization.

Note: Cache can be used to speed up walk back functionality. Cache uses memory allocated by os_malloc/os_realloc and must be freed after use with *fcb_cache_free()*.

To setup cache: struct *fcb_entry_cache* cache; struct *fcb_entry* loc = {};

loc.fe_cache = &cache; fcb_cache_init(fcb, &cache, 100); loc.fe_step_back = true;

fcb_getnext(fcb, &loc);

fcb_cache_free(fcb, cache);

Parameters

- **fcb** -- fcb to init cache for
- **cache** -- cache to init
- **initial_entry_count** -- initial number of entries in the cache

Returns

0 - on success SYS_ENOMEM - when cache memory can't be allocated.

void **fcb_cache_free**(struct *fcb* *fcb, struct *fcb_entry_cache* *cache)

Free memory allocate by fcb cache.

Parameters

- **fcb** -- fcb associated with cache

- **cache** -- cache to free

`int fcb_init(struct fcb *fcb)`

`int fcb_append(struct fcb*, uint16_t len, struct fcb_entry *loc)`

`fcb_append()` appends an entry to circular buffer.

When writing the contents for the entry, use loc->fl_area and loc->fl_data_off with flash_area_write(). When you're finished, call `fcb_append_finish()` with loc as argument.

`int fcb_append_finish(struct fcb*, struct fcb_entry *append_loc)`

`int fcb_write(struct fcb *fcb, struct fcb_entry *loc, const uint8_t *buf, size_t len)`

Write to flash user data.

Function should be called after fcb_append is called and before fcb_finish This is wrapper for flash_area_write() function and uses loc for starting location. loc is modified and can be used for subsequent writes.

Parameters

- **fcb** -- fcb to write entry to
- **loc** -- location of the entry
- **buf** -- data to write
- **len** -- number of bytes to write to fcb

Returns

0 on success, non-zero on failure

`int fcb_walk(struct fcb*, struct flash_area*, fcb_walk_cb cb, void *cb_arg)`

`int fcb_getnext(struct fcb*, struct fcb_entry *loc)`

`int fcb_rotate(struct fcb*)`

Erases the data from oldest sector.

`int fcb_append_to_scratch(struct fcb*)`

Start using the scratch block.

`int fcb_free_sector_cnt(struct fcb *fcb)`

How many sectors are unused.

`int fcb_is_empty(struct fcb *fcb)`

Whether FCB has any data.

`int fcb_offset_last_n(struct fcb *fcb, uint8_t entries, struct fcb_entry *last_n_entry)`

Element at offset *entries* from last position (backwards).

`int fcb_clear(struct fcb *fcb)`

Clears FCB passed to it.

`int fcb_area_info(struct fcb *fcb, struct flash_area *fa, int *elemsp, int *bytesp)`

Usage report for a given FCB area.

Returns number of elements and the number of bytes stored in them.

FCB_MAX_LEN

FCB_OK

Error codes.

FCB_ERR_ARGS

FCB_ERR_FLASH

FCB_ERR_NOVAR

FCB_ERR_NOSPACE

FCB_ERR_NOMEM

FCB_ERR_CRC

FCB_ERR_MAGIC

FCB_ERR_VERSION

FCB_ERR_NEXT_SECT

```
struct fcb_entry_cache  
{  
    #include <fcb.h>  
}
```

struct fcb_entry

#include <fcb.h> Entry location is pointer to area (within fcb->f_sectors), and offset within that area.

struct fcb

```
struct fcb  
{  
    #include <fcb.h>  
}
```

Public Members

uint16_t f_active_sector_entry_count

Number of element in active sector (f_active)

6.16.5 fcb_append

```
int fcb_append(struct fcb *fcb, uint16_t len, struct fcb_entry *append_loc);
```

Start writing a new element to flash. This routine reserves the space in the flash by writing out the element header.

When writing the contents for the entry, use append_loc->fl_area and append_loc->fl_data_off as arguments to flash_area_write(). When finished, call fcb_append_finish() with append_loc as argument.

Arguments

Argu- ments	Description
fcb	Points to FCB where data is written to.
len	Number of bytes to reserve for the element.
loc	Pointer to fcb_entry. fcb_append() will fill this with info about where the element can be written to.

Returned values

Returns 0 on success; nonzero on failure. FCB_ERR_NOSPACE is returned if FCB is full.

Notes

If FCB is full, you need to make more space. This can be done by calling fcb_rotate(). Or if you've reserved scratch sectors, you can take those into use by calling fcb_append_to_scratch().

Example

6.16.6 fcb_append_finish

```
int fcb_append_finish(struct fcb *fcb, struct fcb_entry *append_loc);
```

Finalizes the write of new element. FCB computes the checksum over the element and updates it in flash.

Arguments

Arguments	Description
fcb	Points to FCB where data is written to.
append_loc	Pointer to fcb_entry. Use the fcb_entry returned by fcb_append().

Returned values

Returns 0 on success; nonzero on failure.

Notes

You need to call fcb_append_finish() after writing the element contents. Otherwise FCB will consider this entry to be invalid, and skips over it when reading.

Example

6.16.7 fcb_append_to_scratch

```
int fcb_append_to_scratch(struct fcb *fcb);
```

This can be used if FCB created to have scratch block(s). Once FCB fills up with data, fcb_append() will fail. This routine can be called to start using the reserve block.

Arguments

Arguments	Description
fcb	Points to FCB.

Returned values

Returns 0 on success; nonzero on failure.

Notes

Example

6.16.8 fcb_clear

```
int fcb_clear(struct fcb *fcb);
```

Wipes out all data in FCB.

Arguments

Arguments	Description
fcb	Points to FCB.

Returned values

Returns 0 on success; non-zero otherwise.

Notes**Example****6.16.9 fcb_getnext**

```
int fcb_getnext(struct fcb *, struct fcb_entry *loc);
```

Given element in location in loc, return with loc filled in with information about next element.

If loc->le_elem_off is set to 0, fcb_getnext() will return info about the oldest element in FCB.

Entry data can be read within the callback using flash_area_read(), using loc->fe_area, loc->fe_data_off, and loc->fe_data_len as arguments.

Arguments

Arguments	Description
fcb	Points to FCB where data is written to.
loc	Info about element. On successful call

Returned values

Returns 0 on success; nonzero on failure. Returns FCB_ERR_NOVAR when there are no more elements left.

Notes**Example****6.16.10 fcb_init**

```
int fcb_init(struct fcb *);
```

Initializes FCB. This function walks through the given sectors, finding out how much data already exists in the flash. After calling this, you can start reading/writing data from FCB.

Arguments

Arguments	Description
fcb	Structure describing the FCB.

Returned values

Returns 0 on success; nonzero on failure.

Notes

User should fill in their portion of fcb before calling this function.

Example

6.16.11 fcb_is_empty

```
int fcb_is_empty(struct fcb *fcb);
```

Returns 1 if there are no elements stored in FCB, otherwise returns 0.

Arguments

Arguments	Description
fcb	Points to FCB.

Returned values

See description.

Notes

Example

6.16.12 fcb_offset_last_n

```
int fcb_offset_last_n(struct fcb *fcb, uint8_t entries, uint32_t *last_n_off);
```

Returns the offset of n-th last element.

Arguments

Arguments	Description
fcb	Points to FCB.
entries	How many entries to leave.
last_n_off	Returned offset.

Returned values

0 on success; non-zero on failure.

Notes

Returned offset is relative to beginning of the sector where the element is. Therefore, n-th last element must be found within the last sector of FCB.

Example

6.16.13 fcb_rotate

```
int fcb_rotate(struct fcb *fcb);
```

Erase the oldest sector in FCB.

Arguments

Arguments	Description
fcb	Points to FCB.

Returned values

Returns 0 on success; nonzero on failure.

Notes

Example

6.16.14 fcb_walk

```
typedef int (*fcb_walk_cb)(struct fcb_entry *loc, void *arg);

int fcb_walk(struct fcb *fcb, struct flash_area *area, fcb_walk_cb cb,
             void *cb_arg);
```

Walks over all log entries in FCB. Callback function cb gets called for every entry. If cb wants to stop the walk, it should return a non-zero value.

If specific flash_area is specified, only entries within that sector are walked over.

Entry data can be read within the callback using flash_area_read(), using loc->fe_area, loc->fe_data_off, and loc->fe_data_len as arguments.

Arguments

Arguments	Description
fcb	Points to FCB where data is written to.
area	Optional. Pointer to specific entry in fcb's array of sectors.
cb	Callback function which gets called for every valid entry fcb_walk encounters.
cb_arg	Optional. Parameter which gets passed to callback function.

Returned values

Returns 0 on success; nonzero on failure.

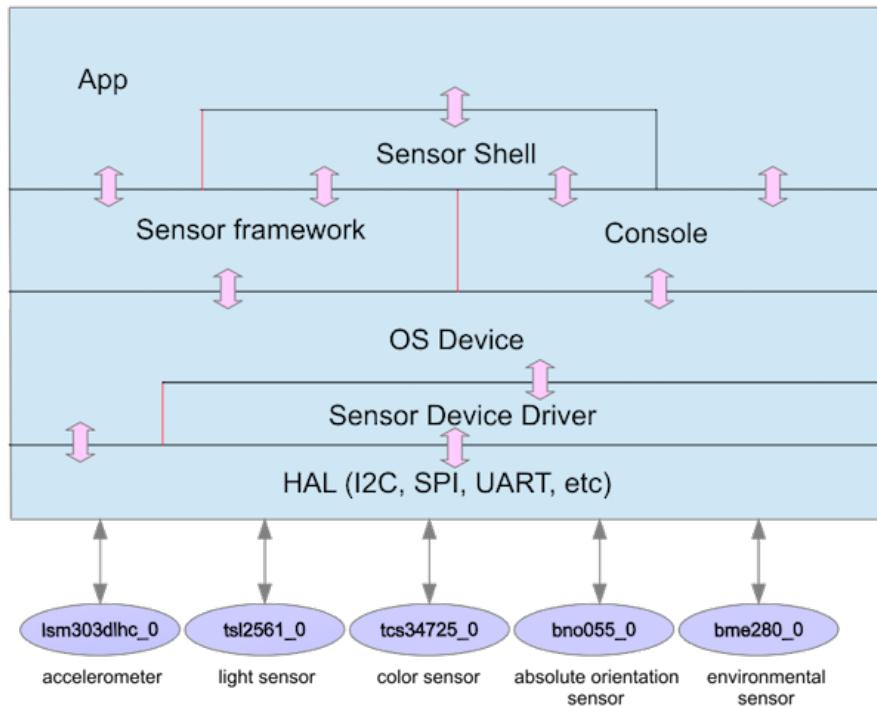
Notes

Example

6.17 Mynewt Sensor Framework Overview

The Mynewt sensor framework is an abstraction layer between an application and sensor devices. The sensor framework provides the following support:

- A set of APIs that allows developers to develop sensor device drivers within a common framework and enables application developers to develop applications that can access sensor data from any Mynewt sensor device using a common interface.
- Support for onboard and off-board sensors.
- An OIC sensor server that exposes sensors as OIC resources and handles OIC CoAP requests for the sensor resources. A developer can easily develop a MyNewt OIC sensor enabled application that serves sensor resource requests from OIC client applications.
- A sensor shell command and optional sensor device driver shell commands to view sensor data and manage sensor device settings for testing and debugging sensor enabled applications and sensor device drivers.



6.17.1 Overview of Sensor Support Packages

In this guide and the package source code, we use `SENSORNAME` (all uppercase) to refer to a sensor product name. For each sensor named `SENSORNAME`:

- The package name for the sensor device driver is `<sensornname>`. All functions and data structures that the sensor device driver exports are prefixed with `<sensornname>`.
- All syscfg settings defined for the sensor are prefixed with `<SENSORNAME>`. For example:
 - The `<SENSORNAME>_CLI` syscfg setting defined in the device driver package to specify whether to enable the sensor device shell command.
 - The `<SENSORNAME>_ONB` syscfg setting defined in the BSP to specify whether the onboard sensor is enabled.
 - The `<SENSORNAME>_OFB` syscfg setting defined in the sensor creator package to specify whether to enable the off-board sensor.

The following Mynewt packages provide sensor support and are needed to build a sensor enabled application.

- `hw/sensor`: The sensor framework package. This package implements the [sensor framework APIs](#), the [OIC sensor server](#), and the [sensor shell command](#).
- `hw/sensor/creator`: The sensor creator package. This package supports off-board sensor devices. It creates the OS devices for the sensor devices that are enabled in an application and configures the sensor devices with

default values. See the [Creating and Configuring a Sensor Device](#) page for more information.

Note: This package is only needed if you are building an application with off-board sensors enabled.

- `hw/bsp/`: The BSP for boards that support onboard sensors. The BSP creates the OS devices for the onboard sensors that the board supports and configures the sensors with default values. See the [Creating and Configuring a Sensor Device](#) page for more information.
- `hw/drivers/sensors/*`: These are the sensor device driver packages. The `hw/drivers/sensors/<sensornname>` package is the device driver for a sensor named `SENSORMNAME`. See the [Sensor Device Driver](#) page for more information.

6.17.2 Sensor API

The sensor API implements the sensor abstraction and the functions:

- For a sensor device driver package to initialize a sensor object with the device specific information.
- For an application to read sensor data from a sensor and to configure a sensor for polling.

A sensor is represented by the `struct sensor` object.

Sensor API Functions Used by a Sensor Device Driver Package

A sensor device driver package must use the sensor API to initialize device specific information for a sensor object and to change sensor configuration types.

Initializing a Sensor Object

When the BSP or the sensor creator package creates an OS device for a sensor named `SENSORMNAME`, it specifies the `<sensornname>_init()` callback function, that the device driver exports, for the `os_dev_create()` function to call to initialize the device. The `<sensornname>_init()` function must use the following sensor API functions to set the driver and interface information in the sensor object:

- The `sensor_init()` function to initialize the `struct sensor` object for the device.
- The `sensor_set_driver()` function to set the types that the sensor device supports and the sensor driver functions to read the sensor data from the device and to retrieve the value type for a given sensor type.
- The `sensor_set_interface()` function to set the interface to use to communicate with the sensor device.

Notes:

- See the [Sensor Device Driver](#) page for the functions and data structures that a sensor driver package exports.
- The `<sensornname>_init()` function must also call the `sensor_mgr_register()` function to register the sensor with the sensor manager. See the [Sensor Manager API](#) for details.

Setting the Configured Sensor Types

The BSP, or the sensor creator package, also calls the <sensorname>_config() function to configure the sensor device with default values. The <sensorname>_config() function is exported by the sensor device driver and must call the sensor API sensor_set_type_mask() function to set the configured sensor types in the sensor object. The configured sensor types are a subset of the sensor types that the device supports. The sensor framework must know the sensor types that a sensor device is configured for because it only reads sensor data for the configured sensor types.

Note: An application may also call the <sensorname>_config() function to configure the sensor device.

Sensor API Functions Used By an Application

The sensor API provides the functions for an application to read sensor data from a sensor and to configure a sensor for polling.

Reading Sensor Data

An application calls the sensor_read() function to read sensor data from a sensor device. You specify a bit mask of the configured sensor types to read from a sensor device and a callback function to call when the sensor data is read. The callback is called for each specified configured type and the data read for that sensor type is passed to the callback.

Setting a Poll Rate for A Sensor

The sensor manager implements a poller that reads sensor data from a sensor at specified poll intervals. An application must call the sensor_set_poll_rate_ms() function to set the poll rate for a sensor in order for poller to poll the sensor.

Note: An application needs to register a *sensor listener* to receive the sensor data that the sensor manager poller reads from a sensor.

Data Structures

We list the main data structures that the sensor API uses and mention things to note. For more details, see the sensor.h include file.

Sensor Object

The `struct sensor` data structure represents the sensor device. The sensor API, the *sensor manager API*, and the *sensor listener API* all operate on the sensor object abstraction. A sensor is maintained in the sensor manager global sensors list.

```
struct sensor {
    /* The OS device this sensor inherits from, this is typically a sensor
     * specific driver.
     */
    struct os_dev *s_dev;

    /* The lock for this sensor object */
    struct os_mutex s_lock;
```

(continues on next page)

(continued from previous page)

```

/* A bit mask describing the types of sensor objects available from this
 * sensor. If the bit corresponding to the sensor_type_t is set, then this
 * sensor supports that variable.
 */
sensor_type_t s_types;

/* Sensor mask of the configured sensor type s*/
sensor_type_t s_mask;
/***
 * Poll rate in MS for this sensor.
 */
uint32_t s_poll_rate;

/* The next time at which we want to poll data from this sensor */
os_time_t s_next_run;

/* Sensor driver specific functions, created by the device registering the
 * sensor.
 */
struct sensor_driver *s_funcs;

/* Sensor last reading timestamp */
struct sensor_timestamp s_sts;

/* Sensor interface structure */
struct sensor_ifc s_ifc;

/* A list of listeners that are registered to receive data off of this
 * sensor
 */
SLIST_HEAD(, sensor_listener) s_listener_list;
/* The next sensor in the global sensor list. */
SLIST_ENTRY(sensor) s_next;
};

```

Note: There are two fields, `s_types` and `s_mask`, of type `sensor_type_t`. The `s_types` field is a bit mask that specifies the sensor types that the sensor device supports. The `s_mask` field is a bit mask that specifies the sensor types that the sensor device is configured for. Only sensor data for a configured sensor type can be read.

Sensor Types

The `sensor_type_t` type is an enumeration of a bit mask of sensor types, with each bit representing one sensor type. Here is an excerpt of the enumeration values. See the `sensor.h` for details:

```

typedef enum {
    /* No sensor type, used for queries */
    SENSOR_TYPE_NONE          = 0,
    /* Accelerometer functionality supported */
    SENSOR_TYPE_ACCELEROMETER = (1 << 0),
    /* Magnetic field supported */

```

(continues on next page)

(continued from previous page)

```

SENSOR_TYPE_MAGNETIC_FIELD      = (1 << 1),
/* Gyroscope supported */
SENSOR_TYPE_GYROSCOPE          = (1 << 2),
/* Light supported */
SENSOR_TYPE_LIGHT                = (1 << 3),
/* Temperature supported */
SENSOR_TYPE_TEMPERATURE         = (1 << 4),

....
```

...

```

SENSOR_TYPE_USER_DEFINED_6      = (1 << 31),
/* A selector, describes all sensors */
SENSOR_TYPE_ALL                 = 0xFFFFFFFF

} sensor_type_t;

```

Sensor Interface

The `struct sensor_itf` data structure represents the interface the sensor device driver uses to communicate with the sensor device.

```

struct sensor_itf {

    /* Sensor interface type */
    uint8_t si_type;

    /* Sensor interface number */
    uint8_t si_num;

    /* Sensor CS pin */
    uint8_t si_cs_pin;

    /* Sensor address */
    uint16_t si_addr;
};

```

The `si_cs_pin` specifies the chip select pin and is optional. The `si_type` field must be of the following types:

```

#define SENSOR_ITF_SPI      (0)
#define SENSOR_ITF_I2C      (1)
#define SENSOR_ITF_UART     (2)

```

Sensor Value Type

The `struct sensor_cfg` data structure represents the configuration sensor type:

```
/**  
 * Configuration structure, describing a specific sensor type off of  
 * an existing sensor.  
 */  
struct sensor_cfg {  
    /* The value type for this sensor (e.g. SENSOR_VALUE_TYPE_INT32).  
     * Used to describe the result format for the value corresponding  
     * to a specific sensor type.  
     */  
    uint8_t sc_valtype;  
    /* Reserved for future usage */  
    uint8_t _reserved[3];  
};
```

Only the `sc_valtype` field is currently used and specifies the data value type of the sensor data. The valid value types are:

```
/**  
 * Opaque 32-bit value, must understand underlying sensor type  
 * format in order to interpret.  
 */  
#define SENSOR_VALUE_TYPE_OPAQUE (0)  
/**  
 * 32-bit signed integer  
 */  
#define SENSOR_VALUE_TYPE_INT32 (1)  
/**  
 * 32-bit floating point  
 */  
#define SENSOR_VALUE_TYPE_FLOAT (2)  
/**  
 * 32-bit integer triplet.  
 */  
#define SENSOR_VALUE_TYPE_INT32_TRIPLET (3)  
/**  
 * 32-bit floating point number triplet.  
 */  
#define SENSOR_VALUE_TYPE_FLOAT_TRIPLET (4)
```

Sensor Driver Functions

The `struct sensor_device` data structure represents the device driver functions. The sensor device driver must implement the functions and set up the function pointers.

```
struct sensor_driver {  
    sensor_read_func_t sd_read;  
    sensor_get_config_func_t sd_get_config;  
};
```

API

enum **sensor_type_t**

Values:

enumerator **SENSOR_TYPE_NONE**

enumerator **SENSOR_TYPE_ACCELEROMETER**

enumerator **SENSOR_TYPE_MAGNETIC_FIELD**

enumerator **SENSOR_TYPE_GYROSCOPE**

enumerator **SENSOR_TYPE_LIGHT**

enumerator **SENSOR_TYPE_TEMPERATURE**

enumerator **SENSOR_TYPE_AMBIENT_TEMPERATURE**

enumerator **SENSOR_TYPE_PRESSURE**

enumerator **SENSOR_TYPE_PROXIMITY**

enumerator **SENSOR_TYPE_RELATIVE_HUMIDITY**

enumerator **SENSOR_TYPE_ROTATION_VECTOR**

enumerator **SENSOR_TYPE_ALTITUDE**

enumerator **SENSOR_TYPE_WEIGHT**

enumerator **SENSOR_TYPE_LINEAR_ACCEL**

enumerator **SENSOR_TYPE_GRAVITY**

enumerator **SENSOR_TYPE_EULER**

enumerator **SENSOR_TYPE_COLOR**

enumerator **SENSOR_TYPE_VOLTAGE**

enumerator **SENSOR_TYPE_CURRENT**

enumerator **SENSOR_TYPE_USER_DEFINED_1**

enumerator **SENSOR_TYPE_USER_DEFINED_2**

enumerator **SENSOR_TYPE_USER_DEFINED_3**

enumerator **SENSOR_TYPE_USER_DEFINED_4**

enumerator **SENSOR_TYPE_USER_DEFINED_5**

enumerator **SENSOR_TYPE_USER_DEFINED_6**

enumerator **SENSOR_TYPE_ALL**

enum **sensor_event_type_t**

Values:

enumerator **SENSOR_EVENT_TYPE_DOUBLE_TAP**

enumerator **SENSOR_EVENT_TYPE_SINGLE_TAP**

enumerator **SENSOR_EVENT_TYPE_FREE_FALL**

enumerator **SENSOR_EVENT_TYPE_SLEEP_CHANGE**

enumerator **SENSOR_EVENT_TYPE_WAKEUP**

enumerator **SENSOR_EVENT_TYPE_SLEEP**

enumerator **SENSOR_EVENT_TYPE_ORIENT_CHANGE**

enumerator **SENSOR_EVENT_TYPE_ORIENT_X_CHANGE**

enumerator **SENSOR_EVENT_TYPE_ORIENT_Y_CHANGE**

enumerator **SENSOR_EVENT_TYPE_ORIENT_Z_CHANGE**

enumerator **SENSOR_EVENT_TYPE_ORIENT_X_L_CHANGE**

enumerator **SENSOR_EVENT_TYPE_ORIENT_Y_L_CHANGE**

enumerator **SENSOR_EVENT_TYPE_ORIENT_Z_L_CHANGE**

enumerator **SENSOR_EVENT_TYPE_ORIENT_X_H_CHANGE**

enumerator **SENSOR_EVENT_TYPE_ORIENT_Y_H_CHANGE**

enumerator **SENSOR_EVENT_TYPE_ORIENT_Z_H_CHANGE**

enumerator **SENSOR_EVENT_TYPE_TILT_CHANGE**

enumerator **SENSOR_EVENT_TYPE_TILT_POS**

enumerator **SENSOR_EVENT_TYPE_TILT_NEG**

typedef int (*sensor_data_func_t)(struct *sensor, void*, void*, *sensor_type_t*)**

Callback for handling sensor data, specified in a sensor listener.

Param sensor

The sensor for which data is being returned

Param arg

The argument provided to *sensor_read()* function.

Param data

A single sensor reading for that sensor listener

Param type

The sensor type for the data function

Return

0 on success, non-zero error code on failure.

typedef int (*sensor_trigger_notify_func_t)(struct *sensor, void*, *sensor_type_t*)**

Callback for sending trigger notification.

Param sensor

Ptr to the sensor

Param data

Ptr to sensor data

Param type

The sensor type

typedef int (*sensor_trigger_cmp_func_t)(*sensor_type_t*, *sensor_data_t, *sensor_data_t**, void*)**

Callback for trigger compare functions.

Param type

Type of sensor

Param low_thresh

The sensor low threshold

Param high_thresh

The sensor high threshold

Param arg

Ptr to data

```
typedef int (*sensor_notifier_func_t)(struct sensor*, void*, sensor_event_type_t)
```

Callback for event notifications.

Param sensor

The sensor that observed the event

Param arg

The opaque argument provided during registration

Param event

The sensor event type that was observed

```
typedef void (*sensor_error_func_t)(struct sensor *sensor, void *arg, int status)
```

Callback for reporting a sensor read error.

Param sensor

The sensor for which a read failed.

Param arg

The optional argument registered with the callback.

Param status

Indicates the cause of the read failure. Determined by the underlying sensor driver.

```
typedef int (*sensor_read_func_t)(struct sensor*, sensor_type_t, sensor_data_func_t, void*, uint32_t)
```

Read a single value from a sensor, given a specific sensor type (e.g.

SENSOR_TYPE_PROXIMITY).

Param sensor

The sensor to read from

Param type

The *type(s)* of sensor values to read. Mask containing that type, provide all, to get all values.

Param data_func

The function to call with each value read. If NULL, it calls all sensor listeners associated with this function.

Param arg

The argument to pass to the read callback.

Param timeout

Timeout. If block until result, specify OS_TIMEOUT_NEVER, 0 returns immediately (no wait.)

Return

0 on success, non-zero error code on failure.

```
typedef int (*sensor_get_config_func_t)(struct sensor*, sensor_type_t, struct sensor_cfg*)
```

Get the configuration of the sensor for the sensor type.

This includes the value type of the sensor.

Param sensor

Ptr to the sensor

Param type

The type of sensor value to get configuration for

Param cfg

A pointer to the sensor value to place the returned result into.

Return

0 on success, non-zero error code on failure.

```
typedef int (*sensor_set_config_func_t)(struct sensor*, void*)
```

Send a new configuration register set to the sensor.

Param sensor

Ptr to the sensor-specific structure

Param arg

Ptr to the sensor-specific configuration structure

Return

0 on success, non-zero error code on failure.

```
typedef int (*sensor_set_trigger_thresh_t)(struct sensor*, sensor_type_t, struct sensor_type_traits *stt)
```

Set the trigger and threshold values for a specific sensor for the sensor type.

Param sensor

Ptr to the sensor

Param type

type of sensor

Param stt

Ptr to teh sensor traits

Return

0 on success, non-zero error code on failure.

```
typedef int (*sensor_clear_trigger_thresh_t)(struct sensor *sensor, sensor_type_t type)
```

Clear the high/low threshold values for a specific sensor for the sensor type.

Param sensor

Ptr to the sensor

Param type

Type of sensor

Return

0 on success, non-zero error code on failure.

```
typedef int (*sensor_set_notification_t)(struct sensor*, sensor_event_type_t)
```

Set the notification expectation for a targeted set of events for the specific sensor.

After this function returns successfully, the implementer shall post corresponding event notifications to the sensor manager.

Param sensor

The sensor to expect notifications from.

Param event

The mask of event types to expect notifications from.

Return

0 on success, non-zero error code on failure.

`typedef int (*sensor_unset_notification_t)(struct sensor*, sensor_event_type_t)`

Unset the notification expectation for a targeted set of events for the specific sensor.

Param sensor

The sensor.

Param event

The mask of event types.

Return

0 on success, non-zero error code on failure.

`typedef int (*sensor_handle_interrupt_t)(struct sensor *sensor)`

Let driver handle interrupt in the sensor context.

Param sensor

Ptr to the sensor

Return

0 on success, non-zero error code on failure.

`typedef int (*sensor_reset_t)(struct sensor*)`

Reset Sensor function Ptr.

Param Ptr

to the sensor

Return

0 on success, non-zero on failure

`void sensor_pkg_init(void)`

Package init function.

Remove when we have post-kernel init stages.

`int sensor_itf_lock(struct sensor_itf *si, os_time_t timeout)`

Lock access to the `sensor_itf` specified by si.

Blocks until lock acquired.

Parameters

- **si** – The `sensor_itf` to lock
- **timeout** – The timeout

Returns

0 on success, non-zero on failure.

`void sensor_itf_unlock(struct sensor_itf *si)`

Unlock access to the `sensor_itf` specified by si.

Parameters

- **si** – The `sensor_itf` to unlock access to

Returns

0 on success, non-zero on failure.

`int sensor_init(struct sensor *sensor, struct os_dev *dev)`

Initialize a sensor.

Parameters

- **sensor** – The sensor to initialize
- **dev** – The device to associate with this sensor.

Returns

0 on success, non-zero error code on failure.

`int sensor_lock(struct sensor *sensor)`

Lock access to the sensor specified by sensor.

Blocks until lock acquired.

Parameters

- **sensor** – The sensor to lock

Returns

0 on success, non-zero on failure.

`void sensor_unlock(struct sensor *sensor)`

Unlock access to the sensor specified by sensor.

Parameters

- **sensor** – The sensor to unlock access to.

`int sensor_read(struct sensor *sensor, sensor_type_t type, sensor_data_func_t data_func, void *arg, uint32_t timeout)`

Read the data for sensor type “type,” from the given sensor and return the result into the “value” parameter.

Parameters

- **sensor** – The sensor to read data from
- **type** – The type of sensor data to read from the sensor
- **data_func** – The callback to call for data returned from that sensor
- **arg** – The argument to pass to this callback.
- **timeout** – Timeout before aborting sensor read

Returns

0 on success, non-zero on failure.

`static inline int sensor_set_driver(struct sensor *sensor, sensor_type_t type, struct sensor_driver *driver)`

Set the driver functions for this sensor, along with the type of sensor data available for the given sensor.

Parameters

- **sensor** – The sensor to set the driver information for
- **type** – The types of sensor data available for this sensor
- **driver** – The driver functions for this sensor

Returns

0 on success, non-zero error code on failure

```
static inline int sensor_set_type_mask(struct sensor *sensor, sensor_type_t mask)
```

Set the sensor driver mask so that the developer who configures the sensor tells the sensor framework which sensor data to send back to the user.

Parameters

- **sensor** – The sensor to set the mask for
- **mask** – The mask

```
static inline sensor_type_t sensor_check_type(struct sensor *sensor, sensor_type_t type)
```

Check if sensor type is supported by the sensor device.

Parameters

- **sensor** – The sensor object
- **type** – Type to be checked

Returns

type bitfield, if supported, 0 if not supported

```
static inline int sensor_set_interface(struct sensor *sensor, struct sensor_itf *s_itf)
```

Set interface type and number.

Parameters

- **sensor** – The sensor to set the interface for
- **s_itf** – The interface type to set

```
static inline int sensor_get_config(struct sensor *sensor, sensor_type_t type, struct sensor_cfg *cfg)
```

Read the configuration for the sensor type “type,” and return the configuration into “cfg.”.

Parameters

- **sensor** – The sensor to read configuration for
- **type** – The type of sensor configuration to read
- **cfg** – The configuration structure to point to.

Returns

0 on success, non-zero error code on failure.

SENSOR_VALUE_TYPE_OPAQUE

Opaque 32-bit value, must understand underlying sensor type format in order to interpret.

SENSOR_VALUE_TYPE_INT32

32-bit signed integer

SENSOR_VALUE_TYPE_FLOAT

32-bit floating point

SENSOR_VALUE_TYPE_INT32_TRIPLET

32-bit integer triplet.

SENSOR_VALUE_TYPE_FLOAT_TRIPLET

32-bit floating point number triplet.

SENSOR_ITF_SPI

Sensor interfaces.

SENSOR_ITF_I2C**SENSOR_ITF_UART****SENSOR_THRESH_ALGO_WINDOW**

Sensor threshold constants.

SENSOR_THRESH_ALGO_WATERMARK**SENSOR_THRESH_ALGO_USERDEF****SENSOR_IGN_LISTENER**

Sensor listener constants.

STANDARD_ACCEL_GRAVITY

Useful constants.

SENSOR_GET_DEVICE(__s)**SENSOR_GET_ITF(__s)****SENSOR_DATA_CMP_GT(__d, __t, __f)****SENSOR_DATA_CMP_LT(__d, __t, __f)****struct sensor_cfg**

#include <sensor.h> Configuration structure, describing a specific sensor type off of an existing sensor.

union sensor_data_t

#include <sensor.h>

Public Members

struct sensor_mag_data ***smd**

struct sensor_accel_data ***sad**

struct sensor_euler_data ***sed**

struct sensor_quat_data ***sqd**

struct sensor_accel_data ***slad**

```
struct sensor_accel_data *sgrd

struct sensor_gyro_data *sgd

struct sensor_temp_data *std

struct sensor_temp_data *satd

struct sensor_light_data *sld

struct sensor_color_data *scd

struct sensor_press_data *spd

struct sensor_humid_data *srhd

struct sensor_listener
    #include <sensor.h>

struct sensor_notifier
    #include <sensor.h> Registration for sensor event notifications.

struct sensor_read_ev_ctx
    #include <sensor.h> Context for sensor read events.

struct sensor_type_traits
    #include <sensor.h> Sensor type traits list.

struct sensor_notify_ev_ctx
    #include <sensor.h>

struct sensor_notify_os_ev
    #include <sensor.h>

struct sensor_driver
    #include <sensor.h>

struct sensor_timestamp
    #include <sensor.h>

struct sensor_int
    #include <sensor.h>
```

```
struct sensor_itf
#include <sensor.h>
```

```
struct sensor
#include <sensor.h>
```

Public Members

uint32_t s_poll_rate
Poll rate in MS for this sensor.

```
struct sensor_read_ctx
#include <sensor.h> Read context for calling user function with argument.
```

6.17.3 Sensor Manager API

The sensor manager API manages the sensors that are enabled in an application. The API allows a sensor device to register itself with the sensor manager and an application to look up the sensors that are enabled. The sensor manager maintains a list of sensors, each represented by a `struct sensor` object defined in the the [Sensor API](#).

Registering Sensors

When the BSP or the sensor creator package creates a sensor device in the kernel, via the `os_dev_create()` function, the sensor device init function must initialize a `struct sensor` object and call the `sensor_mgr_register()` function to register itself with the sensor manager.

Looking Up Sensors

An application uses the sensor manager API to look up the `struct sensor` object for a sensor. The [sensor API](#) and the [sensor listener API](#) perform operations on a sensor object. The sensor manager API provides the following sensor lookup functions:

- `sensor_mgr_find_next_bytype()`: This function returns the next sensor, on the sensors list, whose configured sensor types match one of the specified sensor types to search for. The sensor types to search are specified as a bit mask. You can use the function to search for the first sensor that matches or to iterate through the list and search for all sensors that match. If you are iterating through the list to find the next match, you must call the `sensor_mgr_lock()` function to lock the sensors list before you start the iteration and call the `sensor_mgr_unlock()` unlock the list when you are done. You do not need to lock the sensor list if you use the function to find the first match because the `sensor_mgr_find_next_bytype()` locks the sensors list before the search.
- `sensor_mgr_find_next_bydevname()`: This function returns the sensor that matches the specified device name.

Note: There should only be one sensor that matches a specified device name even though the function name suggests that there can be multiple sensors with the same device name.

Checking Configured Sensor Types

An application may configure a sensor device to only support a subset of supported sensor types. The `sensor_mngr_match_bytype()` function allows an application to check whether a sensor is configured for one of the specified sensor types. The type to check is a bit mask and can include multiple types. The function returns a match when there is a match for one, not all, of the specified types in the bit mask.

Polling Sensors

The sensor manager implements a poller that reads sensor data from sensors at specified poll rates. If an application configures a sensor to be polled, using the `sensor_set_poll_rate_ms()` function defined in the [sensor API](#), the sensor manager poller will poll and read the configured sensor data from the sensor at the specified interval.

The sensor manager poller uses an OS callout to set up a timer event to poll the sensors, and uses the default OS event queue and the OS main task to process timer events. The `SENSOR_MGR_WAKEUP_RATE` syscfg setting specifies the default wakeup rate the sensor manager poller wakes up to poll sensors. The sensor manager poller uses the poll rate for a sensor if the sensor is configured for a higher poll rate than the `SENSOR_MGR_WAKEUP_RATE` setting value.

Note: An application needs to register a [sensor listener](#) to receive the sensor data that the sensor manager poller reads from a sensor.

Data Structures

The sensor manager API uses the `struct sensor` and `sensor_type_t` types.

API

```
typedef int (*sensor_mngr_compare_func_t)(struct sensor*, void*)
```

```
int sensor_mngr_lock(void)
```

Lock sensor manager to access the list of sensors.

```
void sensor_mngr_unlock(void)
```

Unlock sensor manager once the list of sensors has been accessed.

```
int sensor_mngr_register(struct sensor *sensor)
```

Register the sensor with the global sensor list.

This makes the sensor searchable by other packages, who may want to look it up by type.

Parameters

- **sensor** – The sensor to register

Returns

0 on success, non-zero error code on failure.

```
struct os_eventq *sensor_mngr_evq_get(void)
```

Get the current eventq, the system is misconfigured if there is still no parent eventq.

Returns

Ptr OS eventq that the sensor mgr is set to

```
struct sensor *sensor_mgr_find_next(sensor_mgr_compare_func_t, void*, struct sensor*)
```

The sensor manager contains a list of sensors, this function returns the next sensor in that list, for which compare_func() returns successful (one).

If prev_cursor is provided, the function starts at that point in the sensor list.

@warn This function MUST be locked by sensor_mgr_lock/unlock() if the goal is to iterate through sensors (as opposed to just finding one.) As the “prev_cursor” may be resorted in the sensor list, in between calls.

Parameters

- **compare_func** – The comparison function to use against sensors in the list.
- **arg** – The argument to provide to that comparison function
- **prev_cursor** – The previous sensor in the sensor manager list, in case of iteration. If desire is to find first matching sensor, provide a NULL value.

Returns

A pointer to the first sensor found from prev_cursor, or NULL, if none found.

```
struct sensor *sensor_mgr_find_next_bytype(sensor_type_t type, struct sensor *sensor)
```

Find the “next” sensor available for a given sensor type.

If the sensor parameter, is present find the next entry from that parameter. Otherwise, find the first matching sensor.

Parameters

- **type** – The type of sensor to search for
- **sensor** – The cursor to search from, or NULL to start from the beginning.

Returns

A pointer to the sensor object matching that sensor type, or NULL if none found.

```
struct sensor *sensor_mgr_find_next_bydevname(const char *devname, struct sensor *prev_cursor)
```

Search the sensor list and find the next sensor that corresponds to a given device name.

Parameters

- **devname** – The device name to search for
- **sensor** – The previous sensor found with this device name

Returns

0 on success, non-zero error code on failure

```
int sensor_mgr_match_bytype(struct sensor *sensor, void*)
```

Check if sensor type matches.

Parameters

- **sensor** – The sensor object
- **arg** – type to check

Returns

1 if matches, 0 if it doesn’t match.

```
int sensor_set_poll_rate_ms(const char *devname, uint32_t poll_rate)
```

Set the sensor poll rate.

Parameters

- **devname** – Name of the sensor

- **poll_rate** – The poll rate in milli seconds

```
int sensor_set_n_poll_rate(const char *devname, struct sensor_type_traits *stt)
```

Set the sensor poll rate multiple based on the device name, sensor type.

Parameters

- **devname** – Name of the sensor
- **stt** – The sensor type trait

```
int sensor_oic_tx_trigger(struct sensor *sensor, void *arg, sensor_type_t type)
```

Transmit OIC trigger.

Parameters

- **sensor** – Ptr to the sensor
- **arg** – Ptr to sensor data
- **type** – The sensor type

Returns

0 on sucess, non-zero on failure

```
void sensor_trigger_init(struct sensor *sensor, sensor_type_t type, sensor_trigger_notify_func_t notify)
```

Sensor trigger initialization.

Parameters

- **sensor** – Ptr to the sensor
- **type** – Sensor type to enable trigger for
- **notify** – the function to call if the trigger condition is satisfied

```
struct sensor_type_traits *sensor_get_type_traits_bytype(sensor_type_t type, struct sensor *sensor)
```

Search the sensor type traits list for specific type of sensor.

Parameters

- **type** – The sensor type to search for
- **sensor** – Ptr to a sensor

Returns

NULL when no sensor type is found, ptr to *sensor_type_traits* structure when found

```
struct sensor *sensor_get_type_traits_byname(const char*, struct sensor_type_traits**, sensor_type_t)
```

Get the type traits for a sensor.

Parameters

- **devname** – Name of the sensor
- **stt** – Ptr to sensor types trait struct
- **type** – The sensor type

Returns

NULL on failure, sensor struct on success

```
int sensor_set_thresh(const char *devname, struct sensor_type_traits *stt)
```

Set the thresholds along with the comparison algo for a sensor.

Parameters

- **devname** – Name of the sensor
- **sst** – Ptr to sensor type traits containing thresholds

Returns

0 on success, non-zero on failure

```
int sensor_clear_low_thresh(const char *devname, sensor_type_t type)
```

Clears the low threshold for a sensor.

Parameters

- **devname** – Name of the sensor
- **type** – The sensor type

Returns

0 on success, non-zero on failure

```
int sensor_clear_high_thresh(const char *devname, sensor_type_t type)
```

Clears the high threshold for a sensor.

Parameters

- **devname** – Name of the sensor
- **type** – The sensor type

Returns

0 on success, non-zero on failure

```
void sensor_mgr_put_notify_evt(struct sensor_notify_ev_ctx *ctx, sensor_event_type_t evtype)
```

Puts a notification event on the sensor manager evq.

Parameters

- **ctx** – Notification event context
- **evtype** – The notification event type

```
void sensor_mgr_put_interrupt_evt(struct sensor *sensor)
```

Puts a interrupt event on the sensor manager evq.

Parameters

- **sensor** – Sensor Ptr as interrupt event context

```
void sensor_mgr_put_read_evt(void *arg)
```

Puts read event on the sensor manager evq.

Parameters

- **arg** – Argument

```
int sensor_reset(struct sensor *sensor)
```

Resets the sensor.

Parameters

- **Ptr** – to sensor

```
char *sensor_ftostr(float, char*, int)
```

Convinience API to convert floats to strings, might loose some precision due to rounding.

Parameters

- **num** – Floating point number to print
- **fltstr** – Output float string
- **len** – Length of the string to print

```
int sensor_shell_register(void)
```

API to register sensor shell.

```
void sensor_oic_init(void)
```

Iterates through the sensor list and initializes OIC resources based on each sensor type.

6.17.4 Sensor Listener API

The sensor listener API allows an application to register listeners for sensors and get notified whenever sensor data are read from the sensor devices. An application calls the `sensor_register_listener()` function to register a listener that specifies the callback function and the types of sensor data to listen for from a sensor device.

When the `sensor_read()` function defined in the [sensor API](#) is called to read the sensor data for the specified sensor types from a sensor, the `sensor_read()` function calls the listener callback, passing it the sensor data that is read from the sensor.

An application calls the `sensor_unregister_listener()` function to unregister a listener if it no longer needs notifications when data is read from a sensor.

An application can use listeners in conjunction with the sensor manager poller. An application can configure a polling interval for a sensor and register a listener for the sensor types it wants to listen for from the sensor. When the sensor manager poller reads the sensor data from the sensor at each polling interval, the listener callback is called with the sensor data passed to it.

An application can also use listeners for other purposes. For example, an application that uses the OIC sensor server may want to register listeners. The OIC sensor server handles all the OIC requests for the sensor resources and an application does not know about the OIC requests. An application can install a listener if it wants to know about a request for a sensor resource or use the sensor data that the OIC sensor server reads from the sensor.

Data Structures

The `struct sensor_listener` data structure represents a listener. You must initialize a listener structure with a bit mask of the sensor types to listen for from a sensor, a callback function that is called when sensor data is read for one of the sensor types, and an opaque argument to pass to the callback before you call the `sensor_register_listener()` function to register a listener.

```
struct sensor_listener {
{
    /* The type of sensor data to listen for, this is interpreted as a
     * mask, and this listener is called for all sensor types on this
     * sensor that match the mask.
    */
    sensor_type_t sl_sensor_type;

    /* Sensor data handler function, called when has data */
    sensor_data_func_t sl_func;

    /* Argument for the sensor listener */
    void *sl_arg;
```

(continues on next page)

(continued from previous page)

```

/* Next item in the sensor listener list.  The head of this list is
 * contained within the sensor object.
 */
SLIST_ENTRY(sensor_listener) sl_next;
};

```

API

int `sensor_register_listener`(struct *sensor* *sensor, struct *sensor_listener* *listener)

Register a sensor listener.

This allows a calling application to receive callbacks for data from a given sensor object.

For more information on the type of callbacks available, see the documentation for the sensor listener structure.

Parameters

- **sensor** – The sensor to register a listener on
- **listener** – The listener to register onto the sensor

Returns

0 on success, non-zero error code on failure.

int `sensor_unregister_listener`(struct *sensor* *sensor, struct *sensor_listener* *listener)

Un-register a sensor listener.

This allows a calling application to clear callbacks for a given sensor object.

Parameters

- **sensor** – The sensor object
- **listener** – The listener to remove from the sensor listener list

Returns

0 on success, non-zero error code on failure.

int `sensor_register_err_func`(struct *sensor* *sensor, *sensor_error_func_t* err_fn, void *arg)

Register a sensor error callback.

The callback is executed when the sensor manager fails to read from the given sensor.

Parameters

- **sensor** – The sensor to register an error callback on.
- **err_fn** – The function to execute when a read fails.
- **arg** – Optional argument to pass to the callback.

Returns

0 on success, non-zero error code on failure.

6.17.5 Sensor Notifier API

```
int sensor_register_notifier(struct sensor *sensor, struct sensor_notifier *notifier)
```

Register a sensor notifier.

This allows a calling application to receive callbacks any time a requested event is observed.

Parameters

- **sensor** – The sensor to register the notifier on
- **notifier** – The notifier to register

Returns

0 on success, non-zero error code on failure.

```
int sensor_unregister_notifier(struct sensor *sensor, struct sensor_notifier *notifier)
```

Un-register a sensor notifier.

This allows a calling application to stop receiving callbacks for events on the sensor object.

Parameters

- **sensor** – The sensor object to un-register the notifier on
- **notifier** – The notifier to remove from the notification list

Returns

0 on success, non-zero error code on failure.

6.17.6 OIC Sensor Support

The sensor framework provides support for an OIC enabled application to host the sensor devices as OIC resources. The sensor framework provides the following OIC support:

- Creates OIC resources for each sensor device that is enabled in the application. It creates an OIC discoverable and observable resource for each sensor type that the sensor device is configured for.
- Processes CoAP GET requests for the sensor OIC resources. It reads the sensor data samples, encodes the data, and sends back a response.

The sensor package (`hw/sensor`) defines the following syscfg settings for OIC support:

- **SENSOR_OIC**: This setting specifies whether to enable sensor OIC server support. The setting is enabled by default. The sensor package includes the `net/oic` package for the OIC support when this setting is enabled. The `OC_SERVER` syscfg setting that specifies whether to enable OIC server support in the `net/oic` package must also be enabled.
- **SENSOR_OIC_OBS_RATE**: Sets the OIC server observation rate.

An application defines an OIC application initialization handler that sets up the OIC resources it supports and calls the `oc_main_init()` function to initialize the OC server. The application must call the `sensor_oic_init()` function from the the OIC application initialization handler. The `sensor_oic_init()` function creates all the OIC resources for the sensors and registers request callbacks to process CoAP GET requests for the sensor OIC resources.

See the [Enabling OIC Sensor Data Monitoring Tutorials](#) to run and develop sample OIC sensor server applications.

6.17.7 Sensor Shell Command

The sensor framework, `hw/sensor`, package implements a `sensor` shell command. The command allows a user to view the sensors that are enabled in an application and to read the sensor data from the sensors. See the [Enabling an Off-Board Sensor in an Existing Application Tutorial](#)

The package defines the `SENSOR_CLI` syscfg setting to specify whether to enable the `sensor` shell command and enables the setting by default.

6.17.8 Sensor Device Driver

A Mynewt sensor device driver uses the sensor framework abstraction and API to enable applications to access sensor data from any Mynewt sensor device using a common interface. The sensor device driver must also use the Mynewt HAL interface to communicate with and control a sensor device.

This guide describes what a sensor device driver must implement to enable a sensor device within the sensor framework. For information on using the HAL API to communicate with a sensor device, see the [Hardware Layer Abstraction Guide](#).

The `hw/drivers/sensors/<sensorname>` package implements the device driver for the sensor named `SENSORNAME`.

Note: All example excerpts are from the BNO055 sensor device driver package.

Initializing and Configuring a Sensor Device

A driver package for a sensor named `SENSORNAME` must define and export the following data structures and functions to initialize and configure a device:

- `struct <sensorname>`: This data structure represents a sensor device. The structure must include a `dev` field of type `struct os_dev` and a `sensor` field of type `struct sensor`. For example:

```
struct bno055 {
    struct os_dev dev;
    struct sensor sensor;
    struct bno055_cfg cfg;
    os_time_t last_read_time;
};
```

- `struct <sensorname>_cfg`: This data structure defines the configuration for a sensor device. The structure fields are specific to the device. This is the data structure that a BSP, the sensor creator package, or an application sets to configure a sensor device. For example:

```
struct bno055_cfg {
    uint8_t bc_opr_mode;
    uint8_t bc_pwr_mode;
    uint8_t bc_units;
    uint8_t bc_placement;
    uint8_t bc_acc_range;
    uint8_t bc_acc_bw;

    ...
    uint32_t bc_mask;
};
```

- `<sensorname>_init()`: This is the os device initialization callback of type `int (*os_dev_init_func_t)(struct os_dev *, void *)` that the `os_dev_create()` function calls to initialize the device. For example, the bno055 device driver package defines the following function:

```
int bno055_init(struct os_dev *dev, void *arg)
```

The BSP, which creates a device for an onboard sensor, and the sensor creator package, which creates a device for an off-board sensor, calls the `os_dev_create()` function and passes:

- A pointer to a `struct <sensorname>` variable.
- The `<sensorname>_init()` function pointer.
- A pointer to a `struct sensor_itf` variable that specifies the interface the driver uses to communicate with the sensor device.

See the [Creating Sensor Devices](#) page for more details.

The `os_dev_create()` function calls the `<sensorname>_init()` function with a pointer to the `struct <sensorname>` for the `dev` parameter and a pointer to the `struct sensor_itf` for the `arg` parameter.

- `<sensorname>_config()`: This is the sensor configuration function that the BSP, sensor creator package, or an application calls to configure the sensor device. For example:

```
int bno055_config(struct bno055 *bno055, struct bno055_cfg *cfg)
```

Defining Functions to Read Sensor Data and Get Sensor Value Type

A device driver must implement the following functions that the sensor API uses to read sensor data and to get the configuration value type for a sensor:

- A function of type `int (*sensor_read_func_t)(struct sensor *, sensor_type_t, sensor_data_func_t, void *, uint32_t)` that the sensor framework uses to read a single value from a sensor for the specified sensor types. The device driver must implement this function such that it reads, for each sensor type set in the bit mask, a single value from the sensor and calls the `sensor_data_func_t` callback with the opaque callback argument, the sensor data, and the sensor type.
- A function of type `int (*sensor_get_config_func_t)(struct sensor *, sensor_type_t, struct sensor_cfg *)` that returns the value type for the specified sensor type. For example, the value type for a `SENSOR_VALUE_TYPE_TEMPERATURE` sensor might be `SENSOR_VALUE_TYPE_FLOAT` and the value type for a `SENSOR_TYPE_ACCELEROMETER` sensor might be `SENSOR_VALUE_TYPE_FLOAT_TRIPLET`.

The driver initializes a `sensor_driver` structure, shown below, with the pointers to these functions:

```
struct sensor_driver {
    sensor_read_func_t sd_read;
    sensor_get_config_func_t sd_get_config;
};
```

For example:

```
static int bno055_sensor_read(struct sensor *, sensor_type_t,
    sensor_data_func_t, void *, uint32_t);
static int bno055_sensor_get_config(struct sensor *, sensor_type_t,
    struct sensor_cfg *);

static const struct sensor_driver g_bno055_sensor_driver = {
```

(continues on next page)

(continued from previous page)

```
bno055_sensor_read,
bno055_sensor_get_config
};
```

Registering the Sensor in the Sensor Framework

The device driver must initialize and register a `struct sensor` object with the sensor manager. See the [Sensor API](#) and the [Sensor Manager API](#) pages for more details.

The device driver `<sensorname>_init()` function initializes and registers a sensor object as follows:

- Calls the `sensor_init()` function to initialize the `struct sensor` object.
- Calls the `sensor_set_driver()` function to specify the sensor types that the sensor device supports, and the pointer to the `struct sensor_driver` variable that specifies the driver functions to read the sensor data and to get the value type for a sensor.
- Calls the `sensor_set_interface()` function to set the interface that the device driver uses to communicate with the sensor device. The BSP, or sensor creator package for an off-board sensors, sets up the `sensor_itf` and passes it to the `<sensorname>_init()` function. The `sensor_set_interface()` functions saves this information in the sensor object. The device driver uses the `SENSOR_GET_ITF()` macro to retrieve the `sensor_itf` when it needs to communicate with the sensor device.
- Calls the `sensor_mgr_register()` function to register the sensor with the sensor manager.

For example:

```
int
bno055_init(struct os_dev *dev, void *arg)
{
    struct bno055 *bno055;
    struct sensor *sensor;
    int rc;

    if (!arg || !dev) {
        rc = SYS_ENODEV;
        goto err;
    }

    bno055 = (struct bno055 *) dev;

    rc = bno055_default_cfg(&bno055->cfg);
    if (rc) {
        goto err;
    }

    sensor = &bno055->sensor;

    /* Code to setup logging and stats may go here */
    ...

    rc = sensor_init(sensor, dev);
    if (rc != 0) {
        goto err;
    }
}
```

(continues on next page)

(continued from previous page)

```

}

/* Add the accelerometer/magnetometer driver */
rc = sensor_set_driver(sensor, SENSOR_TYPE_ACCELEROMETER |
    SENSOR_TYPE_MAGNETIC_FIELD | SENSOR_TYPE_GYROSCOPE |
    SENSOR_TYPE_TEMPERATURE | SENSOR_TYPE_ROTATION_VECTOR |
    SENSOR_TYPE_GRAVITY | SENSOR_TYPE_LINEAR_ACCEL |
    SENSOR_TYPE_EULER, (struct sensor_driver *) &g_bno055_sensor_driver);
if (rc != 0) {
    goto err;
}

/* Set the interface */
rc = sensor_set_interface(sensor, arg);
if (rc) {
    goto err;
}

rc = sensor_mgr_register(sensor);
if (rc != 0) {
    goto err;
}

return (0);

err:
    return (rc);
}

```

Configuring the Sensor Device and Setting the Configured Sensor Types

After the BSP, or the sensor creator package for an off-board sensor, creates the OS device for a sensor, it calls the <sensorname>_config() function to configure sensor device information such as mode, power mode, and to set the configured sensor types. The <sensorname>_config() function configures the settings on the sensor device. It must also call the sensor_set_type_mask() function to set the configured sensor types in the sensor object. The configured sensor types are a subset of the sensor types that the sensor device supports and the sensor framework only reads sensor data for configured sensor types.

Notes:

- The device driver uses the SENSOR_GET_ITF() macro to retrieve the sensor interface to communicate with the sensor.
- If a sensor device has a chip ID that can be queried, we recommend that the device driver read and verify the chip ID with the data sheet.
- An application may call the <sensorname>_config() function to configure the sensor device.

For example:

```

int
bno055_config(struct bno055 *bno055, struct bno055_cfg *cfg)
{

```

(continues on next page)

(continued from previous page)

```

int rc;
uint8_t id;
uint8_t mode;
struct sensor_if *itf;

itf = SENSOR_GET_ITF(&(bno055->sensor));

/* Check if we can read the chip address */
rc = bno055_get_chip_id(itf, &id);
if (rc) {
    goto err;
}

if (id != BNO055_ID) {
    os_time_delay((OS_TICKS_PER_SEC * 100)/1000 + 1);

    rc = bno055_get_chip_id(itf, &id);
    if (rc) {
        goto err;
    }

    if(id != BNO055_ID) {
        rc = SYS_EINVAL;
        goto err;
    }
}

.....

/* Other code to set the configuration on the sensor device. */

.....


rc = sensor_set_type_mask(&(bno055->sensor), cfg->bc_mask);
if (rc) {
    goto err;
}

bno055->cfg.bc_mask = cfg->bc_mask;

return 0;

err:
    return rc;
}

```

Implementing a Sensor Device Shell Command

A sensor device driver package may optionally implement a sensor device shell command that retrieves and sets sensor device information to aid in testing and debugging. While the sensor framework [sensor shell command](#) reads sensor data for configured sensor types and is useful for testing an application, it does not access low level device information, such as reading register values and setting hardware configurations, that might be needed to test a sensor device or to debug the sensor device driver code. A sensor device shell command implementation is device specific but should minimally support reading sensor data for all the sensor types that the device supports because the sensor framework sensor shell command only reads sensor data for configured sensor types.

The package should:

- Name the sensor device shell command <sensornname>. For example, the sensor device shell command for the BNO055 sensor device is `bno055`.
- Define a <SENSORNAME>_CLI syscfg setting to specify whether the shell command is enabled and disable the setting by default.
- Export a <sensornname>_shell_init() function that an application calls to initialize the sensor shell command when the SENSORNAME_CLI setting is enabled.

For an example on how to implement a sensor device shell command, see the [bno055 shell command](#) source code. See the [Enabling an Off-Board Sensor in an Existing Application Tutorial](#) for examples of the bno055 shell command.

Defining Logs

A sensor device driver should define logs for testing purposes. See the [Log OS Guide](#) for more details on how to add logs. The driver should define a <SENSORNAME>_LOG syscfg setting to specify whether logging is enabled and disable the setting by default.

Here is an example from the BNO055 sensor driver package:

```
#if MYNEWT_VAL(BNO055_LOG)
#include "log/log.h"
#endif

#if MYNEWT_VAL(BNO055_LOG)
#define LOG_MODULE_BNO055 (305)
#define BNO055_INFO(...) LOG_INFO(&_log, LOG_MODULE_BNO055, _VA_ARGS_)
#define BNO055_ERR(...) LOG_ERROR(&_log, LOG_MODULE_BNO055,_VA_ARGS_)
static struct log _log;
#else
#define BNO055_INFO(...)
#define BNO055_ERR(...)
#endif

...

int
bno055_init(struct os_dev *dev, void *arg)
{
    ...

    rc = bno055_default_cfg(&bno055->cfg);
    if (rc) {
```

(continues on next page)

(continued from previous page)

```

    goto err;
}

#if MYNEWT_VAL(BNO055_LOG)
    log_register(dev->od_name, &_log, &log_console_handler, NULL, LOG_SYSLEVEL);
#endif

...
}
```

Defining Stats

A sensor device driver may also define stats for the sensor. See the [Stats OS Guide](#) for more details on how to add stats. The driver should define a <SENSORNAME>_STATS syscfg setting to specify whether stats is enabled and disable the setting by default.

Here is an example from the BNO055 sensor driver package:

```

#if MYNEWT_VAL(BNO055_STATS)
#include "stats/stats.h"
#endif

#if MYNEWT_VAL(BNO055_STATS)

/* Define the stats section and records */
STATS_SECT_START(bno055_stat_section)
STATS_SECT_ENTRY(errors)
STATS_SECT_END

/* Define stat names for querying */
STATS_NAME_START(bno055_stat_section)
STATS_NAME(bno055_stat_section, errors)
STATS_NAME_END(bno055_stat_section)

/* Global variable used to hold stats data */
STATS_SECT_DECL(bno055_stat_section) g_bno055stats;
#endif

...
int
bno055_init(struct os_dev *dev, void *arg)
{

...

#if MYNEWT\_VAL(BNO055\_STATS)

/* Initialise the stats entry */
rc = stats_init(
```

(continues on next page)

(continued from previous page)

```

STATS_HDR(g_bno055stats),
STATS_SIZE_INIT_PARMS(g_bno055stats, STATS_SIZE_32),
STATS_NAME_INIT_PARMS(bno055_stat_section));
SYSINIT_PANIC_ASSERT(rc == 0);
/* Register the entry with the stats registry */
rc = stats_register(dev->od_name, STATS_HDR(g_bno055stats));
SYSINIT_PANIC_ASSERT(rc == 0);

#endif

...
}

```

6.17.9 Creating and Configuring a Sensor Device

The steps to create and configure OS devices for onboard and off-board sensors are very similar. The BSP creates the OS devices for onboard sensors and the `hw/sensor/creator/` package creates the OS devices for off-board sensors.

We discuss how a BSP creates a device for an onboard sensor and then discuss what the `hw/sensor/creator` package does differently to create an off-board sensor. We also discuss how an application can change the default configuration for a sensor device.

Creating an Onboard Sensor

To create and initialize a sensor device named `SENSORNAME`, the BSP implements the following in the `hal_bsp.c` file.

Note: All example excerpts are from the code that creates the LIS2DH12 onboard sensor in the nrf52_thingy BSP.

1. Define a `<SENSORNAME>_ONB` syscfg setting to specify whether the onboard sensor named `SENSORNAME` is enabled. The setting is disabled by default. The setting is used to conditionally include the code to create a sensor device for `SENSORNAME` when it is enabled by the application. For example:

```
syscfg.defs:
    LIS2DH12_ONB:
        description: 'NRF52 Thingy onboard lis2dh12 sensor'
        value: 0
```

2. Include the “`<sensorname>/<sensorname>.h`” header file. The BSP uses the functions and data structures that a device driver package exports. See the [Sensor Device Driver](#) page for details.
3. Declare a variable named `sensorname` of type `struct sensorname`. For example:

```
#if MYNEWT_VAL(LIS2DH12_ONB)
#include <lis2dh12/lis2dh12.h>
static struct lis2dh12 lis2dh12;
#endif
```

4. Declare and specify the values for a variable of type `struct sensor_itf` that the sensor device driver uses to communicate with the sensor device. For example:

```
#if MYNEWT_VAL(LIS2DH12_ONB)
static struct sensor_if i2c_0_itf_lis = {
    .si_type = SENSOR_ITF_I2C,
    .si_num = 0,
    .si_addr = 0x19
<br>
```

5. In the `hal_bsp_init()` function, create an OS device for the sensor device. Call the `os_dev_create()` function and pass the following to the function:

- A pointer to the `sensornname` variable from step 3.
- A pointer to the `<sensornname>_init()` callback function. Note that the device driver package exports this function.
- A pointer to the `struct sensor_if` variable from step 4.

For example:

```
static void sensor\_\dev\_\create(void)
{
    int rc;
    (void)rc;

#ifndef MYNEWT\_VAL(LIS2DH12\_ONB)

    rc = os_dev_create((struct os_dev *) &lis2dh12, "lis2dh12_0",
        OS_DEV_INIT_PRIMARY, 0, lis2dh12_init, (void *)&i2c_0_itf_lis);
    assert(rc == 0);

#endif
}

void hal_bsp_init(void) { int rc;

    ...

sensor_dev_create();

}
```

6. Define a `config_<sensornname>_sensor()` function to set the default configuration for the sensor. This function opens the OS device for the sensor device, initializes the `a``cfg``` variable of type ```struct <sensornname>_cfg ``` with the default settings, calls the `<sensornname>_config()` driver function to configure the device, and closes the device. This function is called when the BSP is initialized during `sysinit()`. For example:

```
int
config_lis2dh12_sensor(void)
{
#ifndef MYNEWT_VAL(LIS2DH12_ONB)
    int rc;
    struct os_dev *dev;
    struct lis2dh12_cfg cfg;
```

(continues on next page)

(continued from previous page)

```

dev = (struct os_dev *) os_dev_open("lis2dh12_0", OS_TIMEOUT_NEVER, NULL);
assert(dev != NULL);

memset(&cfg, 0, sizeof(cfg));

cfg.lc_s_mask = SENSOR_TYPE_ACCELEROMETER;
cfg.lc_rate = LIS2DH12_DATA_RATE_HN_1344HZ_L_5376HZ;
cfg.lc_fs = LIS2DH12_FS_2G;
cfg.lc_pull_up_disc = 1;

rc = lis2dh12_config((struct lis2dh12 *)dev, &cfg);
SYSINIT_PANIC_ASSERT(rc == 0);

os_dev_close(dev);
#endif
return 0;
}

```

7. Add the following in the BSP pkg.yml file:

- A conditional package dependency for the hw/drivers/sensors/<sensornname> package when the <SENSEORNAME>_ONB setting is enabled.
- The config_<sensornname>_sensor function with an init stage of 400 to the pkg.init parameter.

For example:

```

pkg.deps.LIS2DH12_ONB:
  - "@apache-mynewt-core/hw/drivers/sensors/lis2dh12"

pkg.init:
  config_lis2dh12_sensor: 400

```

Creating an Off-Board Sensor

The steps to create an off-board sensor is very similar to the steps for a BSP. The hw/sensor/creator/ package also declares the variables and implements the config_<sensornname>_sensor() function described for a BSP. The package does the following differently.

Note: All example excerpts are from the code that creates the BNO055 off-board sensor in hw/sensor/creator package.

1. Define a <SENSEORNAME>_OFB syscfg setting to specify whether the off-board sensor named SENSEORNAME is enabled. This setting is disabled by default. The hw/sensor/creator package uses the setting to conditionally include the code to create the sensor device when it is enabled by the application.

```

# Package: hw/sensor/creator

syscfg.defs:
  ...

BNO055_OFB:
  description: 'BNO055 is present'
  value : 0

```

(continues on next page)

(continued from previous page)

2. Add the calls to the `os_dev_create()` and the `config_<sensornname>_sensor()` functions in the `sensor_dev_create()` function defined in the `sensor_creator.c` file . The `sensor_dev_create()` function is the hw/sensor/creator package initialization function that `sysinit()` calls.

For example:

```
void
sensor_dev_create(void)
{
    int rc;

    ...

#if MYNEWT_VAL(BNO055_OFB)
    rc = os_dev_create((struct os_dev *) &bno055, "bno055_0",
        OS_DEV_INIT_PRIMARY, 0, bno055_init, (void *)&i2c_0_itf_bno);
    assert(rc == 0);

    rc = config_bno055_sensor();
    assert(rc == 0);
#endif

    ...
}

}
```

3. Add a conditional package dependency for the `hw/drivers/sensors/<sensornname>` package when the `<SENSORNAME>_OFB` setting is enabled. For example:

```
pkg.deps.BNO055_OFB:
- "@apache-mynewt-core/hw/drivers/sensors/bno055"
```

Reconfiguring A Sensor Device by an Application

The BSP and sensor creator package use a default configuration and enable all supported sensors on a sensor device by default. If the default configuration does not meet your application requirements, you may change the default configuration for a sensor device. As in the `config_<sensornname>_sensor` function, an application must open the OS device for the sensor, set up the values for the `<sensornname>_cfg` structure, call the `<sensornname>_config()` device driver function to change the configuration in the device, and close the OS device.

We recommend that you copy the `config_<sensornname>_sensor()` function from the BSP or the sensor creator package in to your application code and change the desired settings. Note that you must keep all the fields in the `<sensornname>_cfg` structure initialized with the default values for the settings that you do not want to change.

See the [Changing the Default Configuration for a Sensor Tutorial](#) for more details on how to change the default sensor configuration from an application.

6.18 testutil

The testutil package is a test framework that provides facilities for specifying test cases and recording test results.

You would use it to build regression tests for your library.

- *Description*
- *Example*
- *Data structures*
- *API*

6.18.1 Description

A package may optionally contain a set of test cases. Test cases are not normally compiled and linked when a package is built; they are only included when the “test” identity is specified. All of a package’s test code goes in its `src/test` directory. For example, the nffs package’s test code is located in the following directory:

```
* fs/nffs/src/test/
```

This directory contains the source and header files that implement the nffs test code.

The test code has access to all the header files in the following directories:

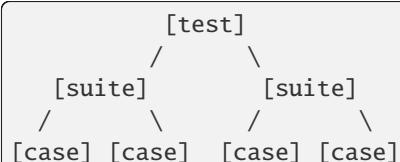
```
* src
* src/arch/<target-arch>
* include
* src/test
* src/test/arch/<target-arch>
* include directories of all package dependencies
```

Package test code typically depends on the testutil package, described later in this document.

Some test cases or test initialization code may be platform-specific. In such cases, the platform-specific function definitions are placed in arch subdirectories within the package test directory.

While building the test code (i.e., when the `test` identity is specified), the newt tool defines the `TEST` macro. This macro is defined during compilation of all C source files in all projects and packages.

Tests are structured according to the following hierarchy:



I.e., a test consists of test suites, and a test suite consists of test cases.

The test code uses testutil to define test suites and test cases.

Regression test can then be executed using ‘newt target test’ command, or by including a call to your test suite from `project/test/src/test.c`.

6.18.2 Example

This Tutorial shows how to create a test suite for a Mynewt package.

6.18.3 Data structures

```
struct ts_config {
    int ts_print_results;
    int ts_system_assert;

    const char *ts_suite_name;

    /*
     * Called prior to the first test in the suite
     */
    tu_init_test_fn_t *ts_suite_init_cb;
    void *ts_suite_init_arg;

    /*
     * Called after the last test in the suite
     */
    tu_init_test_fn_t *ts_suite_complete_cb;
    void *ts_suite_complete_arg;

    /*
     * Called before every test in the suite
     */
    tu_pre_test_fn_t *ts_case_pre_test_cb;
    void *ts_case_pre_arg;

    /*
     * Called after every test in the suite
     */
    tu_post_test_fn_t *ts_case_post_test_cb;
    void *ts_case_post_arg;

    /*
     * Called after test returns success
     */
    tu_case_report_fn_t *ts_case_pass_cb;
    void *ts_case_pass_arg;

    /*
     * Called after test fails (typically through a failed test assert)
     */
    tu_case_report_fn_t *ts_case_fail_cb;
    void *ts_case_fail_arg;

    /*
     * restart after running the test suite - self-test only
     */
    tu_suite_restart_fn_t *ts_restart_cb;
```

(continues on next page)

(continued from previous page)

```
    void *ts_restart_arg;  
};
```

The global `ts_config` struct contains all the testutil package's settings.

6.18.4 API

```
typedef void tu_case_report_fn_t(const char *msg, void *arg)  
  
typedef void tu_pre_test_fn_t(void *arg)  
  
typedef void tu_post_test_fn_t(void *arg)  
  
typedef void tu_testsuite_fn_t(void)  
  
struct ts_testsuite_list g_ts_suites  
  
struct tu_config tu_config  
  
const char *tu_suite_name  
  
const char *tu_case_name  
  
int tu_any_failed  
  
int tu_suite_failed  
  
int tu_case_reported  
  
int tu_case_failed  
  
int tu_case_idx  
  
jmp_buf tu_case_jb  
  
void tu_set_pass_cb(tu_case_report_fn_t *cb, void *cb_arg)  
void tu_set_fail_cb(tu_case_report_fn_t *cb, void *cb_arg)  
void tu_suite_init(const char *name)  
void tu_suite_pre_test(void)  
void tu_suite_complete(void)  
int tu_suite_register(tu_testsuite_fn_t *ts, const char *name)  
  
SLIST_HEAD (ts_testsuite_list, ts_suite)
```

```

void tu_restart(void)

void tu_start_os(const char *test_task_name, os_task_func_t test_task_handler)

void tu_suite_set_pre_test_cb(tu_pre_test_fn_t *cb, void *cb_arg)

void tu_case_set_post_test_cb(tu_post_test_fn_t *cb, void *cb_arg)

void tu_case_init(const char *name)

void tu_case_complete(void)

void tu_case_pass(void)

void tu_case_fail(void)

void tu_case_fail_assert(int fatal, const char *file, int line, const char *expr, const char *format, ...)

void tu_case_write_pass_auto(void)

void tu_case_pass_manual(const char *file, int line, const char *format, ...)

void tu_case_post_test(void)

TEST_SUITE_DECL(suite_name)

TEST_SUITE_REGISTER(suite_name)

TEST_SUITE(suite_name)

TEST_CASE_DECL(case_name)

TEST_CASE_DEFN(case_name, do_sysinit, body)

TEST_CASE(case_name)

```

Defines a test case suitable for running in an application.

The ***TEST_CASE()*** macro should not be used for self-tests (i.e., tests that are run with **newt test**). Instead, ***TEST_CASE_SELF()*** or ***TEST_CASE_TASK()*** should be preferred; those macros perform system clean up before the test runs.

```
TEST_CASE_SELF_EMIT_(case_name)
```

```
TEST_CASE_TASK_EMIT_(case_name)
```

```
TEST_CASE_SELF(case_name)
```

Defines a test case for self-test mode (i.e., suitable for **newt test**).

Test cases defined with ***TEST_CASE_SELF()*** execute **sysinit()** before the test body.

```
TEST_CASE_TASK(case_name)
```

Defines a test case that runs inside a temporary task.

Most tests don't utilize the OS scheduler; they simply run in **main()**, outside the context of a task. However, sometimes the OS is required to fully test a package, e.g., to verify timeouts and other timed events.

The ***TEST_CASE_TASK()*** macro simplifies the implementation of test cases that require the OS. This macro is identical in usage to ***TEST_CASE_SELF()***, except the test case it defines performs some additional preliminary work:

- a. Creates the default task.
- b. Creates the “test task” (the task where the test itself runs).
- c. Starts the OS.

The body following the macro invocation is what actually runs in the test task. The test task has a priority of `OS_MAIN_TASK_PRIO + 1`, so it yields to the main task. Thus, tests using this macro typically have the following form:

```
TEST_CASE_TASK(my_test)
{
    enqueue_event_to_main_task();
    // Event immediately runs to completion.
    TEST_ASSERT(expected_event_result);

    //
}
```

The `TEST_CASE_TASK()` macro is only usable in self-tests (i.e., tests that are run with `newt test`).

`FIRST_AUX(first, ...)`

`FIRST(...)`

`NUM(...)`

`ARG10(a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, ...)`

`REST_OR_0(...)`

`REST_OR_0_AUX(qty, ...)`

`REST_OR_0_AUX_INNER(qty, ...)`

`REST_OR_0_AUX_1(first)`

`REST_OR_0_AUX_N(first, ...)`

`XSTR(s)`

`STR(s)`

`TEST_ASSERT_FULL(fatal, expr, ...)`

`TEST_ASSERT(...)`

`TEST_ASSERT_FATAL(...)`

`TEST_PASS(...)`

`ASSERT_IF_TEST(expr)`

`struct ts_suite`

`#include <testutil.h>`

`struct tu_config`

`#include <testutil.h>`

6.19 JSON

JSON is a data interchange format. The description of this format can be found from IETF RFC 4627.

- *Description*
- *Data structures*
 - *Encoding*
 - *Decoding*
- *API*

6.19.1 Description

This package helps in converting between C data types and JSON data objects. It supports both encoding and decoding.

6.19.2 Data structures

Encoding

```
/* Encoding functions */
typedef int (*json_write_func_t)(void *buf, char *data,
                                 int len);

struct json_encoder {
    json_write_func_t je_write;
    void *je_arg;
    int je_wr_commas:1;
    char je_encode_buf[64];
};
```

Here's the data structure encoder funtions use, and it must be initialized by the caller. The key element is *je_write*, which is a function pointer which gets called whenever encoding routine is ready with encoded data. The element *je_arg* is passed to *je_write* as the first argument. The rest of the structure contents are for internal state management. This function should collect all the data encoder function generates. It can collect this data to a flat buffer, chain of mbufs or even stream through.

```
/**
 * For encode.  The contents of a JSON value to encode.
 */
struct json_value {
    uint8_t jv_pad1;
    uint8_t jv_type;
    uint16_t jv_len;

    union {
        uint64_t u;
        float f;
    };
};
```

(continues on next page)

(continued from previous page)

```

char *str;
struct {
    char **keys;
    struct json_value **values;
} composite;
} jv_val;
};

```

This data structure is filled with data to be encoded. It is best to fill this using the macros `JSON_VALUE_STRING()` or `JSON_VALUE_STRINGN()` when value is string, `JSON_VALUE_INT()` when value is an integer, and so forth.

Decoding

```

/* when you implement a json buffer, you must implement these functions */

/* returns the next character in the buffer or '\0' */
typedef char (*json_buffer_read_next_byte_t)(struct json_buffer *);
/* returns the previous character in the buffer or '\0' */
typedef char (*json_buffer_read_prev_byte_t)(struct json_buffer *);
/* returns the number of characters read or zero */
typedef int (*json_buffer_readn_t)(struct json_buffer *, char *buf, int n);

struct json_buffer {
    json_buffer_readn_t jb_readn;
    json_buffer_read_next_byte_t jb_read_next;
    json_buffer_read_prev_byte_t jb_read_prev;
};

```

Function pointers within this structure are used by decoder when it is reading in more data to decode.

```

struct json_attr_t {
    char *attribute;
    json_type type;
    union {
        int *integer;
        unsigned int *uinteger;
        double *real;
        char *string;
        bool *boolean;
        char *character;
        struct json_array_t array;
        size_t offset;
    } addr;
    union {
        int integer;
        unsigned int uinteger;
        double real;
        bool boolean;
        char character;
        char *check;
    } dflt;
}

```

(continues on next page)

(continued from previous page)

```

size_t len;
const struct json_enum_t *map;
bool nodefault;
};

```

This structure tells the decoder about a particular name/value pair. Structure must be filled in before calling the decoder routine `json_read_object()`.

Element	Description
attribute	Name of the value
type	The type of the variable; see enum <code>json_type</code>
addr	Contains the address where value should be stored
dflt	Default value to fill in, if this name is not found
len	Max number of bytes to read in for value
nodefault	If set, default value is not copied name

6.19.3 API

Defines

`JSON_VALUE_TYPE_BOOL`

`JSON_VALUE_TYPE_UINT64`

`JSON_VALUE_TYPE_INT64`

`JSON_VALUE_TYPE_STRING`

`JSON_VALUE_TYPE_ARRAY`

`JSON_VALUE_TYPE_OBJECT`

`JSON_VALUE_STRING(__jv, __str)`

`JSON_VALUE_STRINGN(__jv, __str, __len)`

`JSON_VALUE_BOOL(__jv, __v)`

`JSON_VALUE_INT(__jv, __v)`

`JSON_VALUE_UINT(__jv, __v)`

`JSON_NITEMS(x)`

`JSON_ATTR_MAX`

`JSON_VAL_MAX`

`JSON_ERR_OBSTART`

`JSON_ERR_ATTRSTART`

`JSON_ERR_BADATTR`

`JSON_ERR_ATTRLEN`

`JSON_ERR_NOARRAY`

`JSON_ERR_NOBRAK`

`JSON_ERR_STRLONG`

`JSON_ERR_TOKLONG`

`JSON_ERR_BADTRAIL`

`JSON_ERR_ARRAYSTART`

`JSON_ERR_OBJARR`

`JSON_ERR_SUBTOOLONG`

`JSON_ERR_BADSUBTRAIL`

`JSON_ERR_SUBTYPE`

`JSON_ERR_BADSTRING`

`JSON_ERR_CHECKFAIL`

`JSON_ERR_NOPARSTR`

`JSON_ERR_BADENUM`

`JSON_ERR_QNONSTRING`

`JSON_ERR_NONQSTRING`

`JSON_ERR_MISC`

`JSON_ERR_BADNUM`

`JSON_ERR_NULLPTR`

`JSON_STRUCT_OBJECT(s, f)`

`JSON_STRUCT_ARRAY(a, e, n)`

Typedefs

`typedef int (*json_write_func_t)(void *buf, char *data, int len)`

`typedef char (*json_buffer_read_next_byte_t)(struct json_buffer*)`

`typedef char (*json_buffer_read_prev_byte_t)(struct json_buffer*)`

`typedef int (*json_buffer_readn_t)(struct json_buffer*, char *buf, int n)`

Enums

enum `json_type`

Values:

enumerator `t_integer`

enumerator `t_uinteger`

enumerator `t_real`

enumerator `t_string`

enumerator `t_boolean`

enumerator `t_character`

enumerator `t_object`

enumerator `t_structobject`

enumerator `t_array`

enumerator `t_check`

enumerator **t_ignore**

Functions

```
int json_encode_object_start(struct json_encoder*)
int json_encode_object_key(struct json_encoder *encoder, char *key)
int json_encode_object_entry(struct json_encoder*, char*, struct json_value*)
int json_encode_object_finish(struct json_encoder*)
int json_encode_array_name(struct json_encoder *encoder, char *name)
int json_encode_array_start(struct json_encoder *encoder)
int json_encode_array_value(struct json_encoder *encoder, struct json_value *val)
int json_encode_array_finish(struct json_encoder *encoder)
int json_read_object(struct json_buffer*, const struct json_attr_t*)
int json_read_array(struct json_buffer*, const struct json_array_t*)

struct json_value
#include <json.h>
```

Public Members

```
uint8_t jv_pad1
uint8_t jv_type
uint16_t jv_len
union json_value jv_val
```

```
struct json_encoder
#include <json.h>
```

Public Members

```
json_write_func_t je_write
void *je_arg
int je_wr_commas
```

```
char je_encode_buf[64]

struct json_enum_t
#include <json.h>
```

Public Members

```
char *name

long long int value

struct json_array_t
#include <json.h>
```

Public Members

```
json_type element_type

union json_array_t arr

int *count

int maxlen

struct json_attr_t
#include <json.h>
```

Public Members

```
char *attribute

json_type type

union json_attr_t addr

union json_attr_t dflt

size_t len

const struct json_enum_t *map
```

```
bool nodefault

struct json_buffer
#include <json.h>
```

Public Members

```
json_buffer_readn_t jb_readn
json_buffer_read_next_byte_t jb_read_next
json_buffer_read_prev_byte_t jb_read_prev
```

```
union jv_val
```

Public Members

```
uint64_t u
float f1
char *str
struct json_value composite
struct composite
```

Public Members

```
char **keys
struct json_value **values
union arr
```

Public Members

```
struct json_array_t objects  
  
struct json_array_t strings  
  
struct json_array_t integers  
  
struct json_array_t uintegers  
  
struct json_array_t reals  
  
struct json_array_t booleans  
  
struct objects
```

Public Members

```
const struct json_attr_t *subtype  
  
char *base  
  
size_t stride  
  
struct strings
```

Public Members

```
char **ptrs  
  
char *store  
  
int storelen  
  
struct integers
```

Public Members

long long int ***store**

struct **uintegers**

Public Members

long long unsigned int ***store**

struct **reals**

Public Members

double ***store**

struct **booleans**

Public Members

bool ***store**

union **addr**

Public Members

long long int ***integer**

long long unsigned int ***uinteger**

double ***real**

char ***string**

bool ***boolean**

char ***character**

struct *json_array_t* **array**

```
size_t offset

union dflt
```

Public Members

long long int **integer**

long long unsigned int **uinteger**

double **real**

bool **boolean**

char **character**

char ***check**

6.20 Manufacturing Support

6.20.1 Description

An mfgimage is a binary that gets written to a Mynewt device at manufacturing time. Unlike a Mynewt target which corresponds to a single executable image, an mfgimage represents the entire contents of a flash device.

6.20.2 Definitions

Term	Long Name	Meaning
Flashdev	Flash device	A single piece of flash hardware. A typical device might contain two flashdevs: 1) internal flash, and 2) external SPI flash.
Mfgimage	Manufacturing image	A file with the entire contents of a single flashdev. At manufacturing time, a separate mfgimage is written to each of the device's flashdevs.
Boot Mfgimage	Boot manufacturing image	The mfgimage containing the boot loader; always written to internal flash.
MMR	Manufacturing Meta Region	A chunk of read-only data included in every mfgimage. Contains identifying information for the mfgimage and other data that stays with the device until end of life.
TLV	Type Length Value	A simple extensible means of representing data. Contains three fields: 1) type of data, 2) length of data, and 3) the data itself.
MfgID	Manufacturing ID	Identifies which set of mfgimages a device was built with. Expressed as a list of SHA256 hashes.

6.20.3 Details

Typically, an mfgimage consists of:

- 1 boot loader.
 - 1 or 2 Mynewt images.
 - Extra configuration (e.g., a pre-populated sys/config region).

In addition, each mgfimage contains a manufacturing meta region (MMR). The MMR consists of read-only data that resides in flash for the lifetime of the device. There is currently support for three MMR TLV types:

- Hash of mfgimage
 - Flash map
 - Device / offset of next MMR

The manufacturing hash indicates which manufacturing image a device was built with. A management system may need this information to determine which images a device can be upgraded to, for example. A Mynewt device exposes its manufacturing hash via the `id/mfghash` config setting.

Since MMRs are not intended to be modified or erased, they must be placed in unmodifiable areas of flash. In the boot mfgimage, the MMR *must* be placed in the flash area containing the boot loader. For non-boot mfgimages, the MMR can go in any unused area in the relevant flashdev.

6.20.4 Manufacturing ID

Each mfgimage has its own MMR containing a hash.

The MMR at the end of the boot mfgimage (“boot MMR”) must be present. The boot MMR indicates the flash locations of other MMRs via the `mmr_ref` TLV type.

At startup, the firmware reads the boot MMR. Next, it reads any additional MMRs indicated by `mmr_ref` TLVs. An `mmr_ref` TLV contains one field: The ID of the flash area where the next MMR is located.

After all MMRs have been read, the firmware populates the `id/mfghash` setting with a colon-separated list of hashes. By reading and parsing this setting, a client can derive the full list of mfgimages that the device was built with.

One important implication is that MMR areas should never be moved in a BSP's flash map. Such a change would produce firmware that is incompatible with older devices.

6.20.5 MMR Structure

An MMR is always located at the end its flash area. Any other placement is invalid.

An MMR has the following structure:

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
TLV type TLV size TLV data ("TLV size" bytes) ~			
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
~			~
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
TLV type TLV size TLV data ("TLV size" bytes) ~			
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+			
~			~

(continues on next page)

(continued from previous page)

Region size	Version	0xff padding
Magic (0x3bb2a269)		

+-----+-----+-----+-----+-----+-----+-----+
| end of flash area |

The number of TLVs is variable; two are shown above for illustrative purposes.

Fields:

<TLVs>

1. TLV type: Indicates the type of data to follow.
2. TLV size: The number of bytes of data to follow.
3. TLV data: “TLV size” bytes of data.

<Footer>

4. Region size: The size, in bytes, of the entire manufacturing meta region; includes TLVs and footer.
5. Version: Manufacturing meta version number; always 0x02.
6. Magic: Indicates the presence of the manufacturing meta region.

API

Defines

MFG_HASH_SZ

MFG_META_TLV_TYPE_HASH

MFG_META_TLV_TYPE_FLASH_AREA

MFG_META_TLV_TYPE_FLASH_TRAITS

Informational only; not read by firmware.

MFG_META_TLV_TYPE_MMR_REF

Functions

void **mfg_open**(struct *mfg_reader* *out_reader)

Opens the manufacturing space for reading.

The resulting *mfg_reader* object should be passed to subsequent seek and read functions.

int **mfg_seek_next**(struct *mfg_reader* *reader)

Seeks to the next mfg TLV.

The caller must initialize the supplied *mfg_reader* with *mfg_open()* prior to calling this function.

Parameters

- **reader** – The reader to seek with.

Returns

0 if the next TLV was successfully seeked to. SYS_EDONE if there are no additional TLVs available. Other MFG error code on failure.

```
int mfg_seek_next_with_type(struct mfg_reader *reader, uint8_t type)
```

Seeks to the next mfg TLV with the specified type.

The caller must initialize the supplied *mfg_reader* with *mfg_open()* prior to calling this function.

Parameters

- **reader** – The reader to seek with.
- **type** – The type of TLV to seek to; one of the MFG_META_TLV_TYPE_[...] constants.

Returns

0 if the next TLV was successfully seeked to. SYS_EDONE if there are no additional TLVs with the specified type available. Other MFG error code on failure.

```
int mfg_read_tlv_hash(const struct mfg_reader *reader, void *out_hash)
```

Reads a hash TLV from the manufacturing space.

This function should only be called when the provided reader is pointing at a TLV with the MFG_META_TLV_TYPE_HASH type.

Parameters

- **reader** – The reader to read with.
- **out_mr** – (out) On success, the retrieved MMR reference information gets written here.

Returns

0 on success; MFG error code on failure.

```
int mfg_read_tlv_flash_area(const struct mfg_reader *reader, struct mfg_meta_flash_area *out_mfa)
```

Reads a flash-area TLV from the manufacturing space.

This function should only be called when the provided reader is pointing at a TLV with the MFG_META_TLV_TYPE_FLASH_AREA type.

Parameters

- **reader** – The reader to read with.
- **out_mfa** – (out) On success, the retrieved flash area information gets written here.

Returns

0 on success; MFG error code on failure.

```
int mfg_read_tlv_mmr_ref(const struct mfg_reader *reader, struct mfg_meta_mmr_ref *out_mr)
```

Reads an MMR ref TLV from the manufacturing space.

This function should only be called when the provided reader is pointing at a TLV with the MFG_META_TLV_TYPE_MMR_REF type.

Parameters

- **reader** – The reader to read with.
- **out_mr** – (out) On success, the retrieved MMR reference information gets written here.

Returns

0 on success; MFG error code on failure.

```
void mfg_init(void)
    Initializes the mfg package.
```

Variables

```
uint8_t type
```

```
uint8_t size
```

```
uint8_t area_id
```

```
uint8_t device_id
```

```
uint32_t offset
```

```
uint8_t min_write_sz
```

```
struct mfg_meta_tlv
    #include <mfg.h>
```

Public Members

```
uint8_t type
```

```
uint8_t size
```

```
struct mfg_meta_flash_area
    #include <mfg.h>
```

Public Members

```
uint8_t area_id
```

```
uint8_t device_id
```

```
uint32_t offset
```

```
uint32_t size
```

```
struct mfg_meta_flash_traits
    #include <mfg.h> Informational only; not read by firmware.
```

Public Members

```
uint8_t device_id  
  
uint8_t min_write_sz  
  
struct mfg_meta_mmr_ref  
#include <mfg.h>
```

Public Members

```
uint8_t area_id
```

struct mfg_reader

#include <mfg.h> Object used for reading records from the manufacturing space.

The [mfg_open\(\)](#) function should be used to construct a reader object.

Public Members

```
struct mfg_meta_tlv cur_tlv
```

Public (read-only).

```
uint8_t mmr_idx
```

Private.

```
uint32_t offset
```

6.21 Board support

This section lists some of the supported boards, their current status and links to tutorial specific to the board. A list of all supported boards can be found in [@apache-mynewt-core/hw/bsp](#).

6.21.1 PineTime smartwatch

This page is about the board support package for the Pine64 PineTime smartwatch. You can find some general documentation at the [device wiki](#). You could buy a dev kit in the [store](#).

- [Status](#)
- [Tutorials](#)

Status

Currently the status is: incomplete.

The board support package contains the code for booting the device and the pin definitions. This means you can load an application like `blinky`, but no peripherals can be used. New drivers will be added in the future.

Tutorials

- *Blinky, your “Hello World!”, on a PineTime smartwatch*

BLE USER GUIDE

Apache Mynewt offers the world's first fully open-source Bluetooth Low Energy (BLE) or Bluetooth Smart stack fully compliant with Bluetooth 5 specifications with support for Bluetooth Mesh. It is called NimBLE.

BLE technology operates in the unlicensed industrial, scientific and medical (ISM) band at 2.4 to 2.485 GHz which is available in most countries. It uses a spread spectrum, frequency hopping, full-duplex signal. BLE FHSS employs 40 2-MHz-wide channels to ensure greater reliability over longer distances. It also features 0-dBm (1 mW) power output and a typical maximum range of 50 meters. With Bluetooth 5 specification range can be increased 4 times and speed 2 time.

- *Features*
- *Bluetooth Mesh features*
- *Components*
- *Example NimBLE projects*

Note that BLE is not compatible with standard Bluetooth.

7.1 Features

NimBLE complies with Bluetooth Core Specification 5.0 which makes it an ideal wireless technology for the Internet of Things (IoT).

- LE Advertising Extensions
- 2Msym/s PHY for higher throughput
- Coded PHY for LE Long Range
- High Duty Cycle Non-Connectable Advertising
- Channel Selection Algorithm #2 to utilize channels in more efficient way.
- LE Privacy 1.2 for frequent changes to the device address to make it difficult to track for outsiders
- LE Secure Connections featuring FIPS-compliant algorithms.
- LE Data Length Extension for higher throughput
- **Coming Soon:** Assigning an Internet Protocol (IP) address (compliant with the IPv6 or 6LoWPAN standard) to a Bluetooth device through Internet Protocol Support Profile (IPSP)

The Bluetooth 5 is backward compatible with previous Bluetooth version 4.2 which is also supported by Apache Mynewt.

7.2 Bluetooth Mesh features

Bluetooth Mesh is a great addition to and opens a wide range of new possibilities for the IoT connectivity space. NimBLE fully supports the following Bluetooth Mesh features:

- Advertising and GATT bearers
- PB-GATT and PB-ADV provisioning
- Foundation Models (server role)
- Relay support
- GATT Proxy

7.3 Components

A Bluetooth low energy stack (NimBLE included) consists of two components with several subcomponents:

- **Controller**
 - **Physical Layer**: adaptive frequency-hopping Gaussian Frequency Shift Keying (GFSK) radio using 40 RF channels (0-39), with 2 MHz spacing.
 - **Link Layer**: with one of five states (Standby, Advertising, Scanning, Initiating, Connection states) active at any time
- **Host**
 - **Logical Link Control and Adaptation Protocol (L2CAP)**: provides logical channels, named L2CAP channels, which are multiplexed over one or more logical links to provide packet segmentation and re-assembly, flow control, error control, streaming, QoS etc.
 - **Security Manager (SM)**: uses Security Manager Protocol (SMP) for pairing and transport specific key distribution for securing radio communication
 - **Attribute protocol (ATT)**: allows a device (*Server*) to expose certain pieces of data, known as *Attributes*, to another device (*Client*)
 - **Generic Attribute Profile (GATT)**: a framework for using the ATT protocol to exchange attributes encapsulated as *Characteristics* or *Services*
 - **Generic Access Profile (GAP)**: a base profile which all Bluetooth devices implement, which in the case of LE, defines the Physical Layer, Link Layer, L2CAP, Security Manager, Attribute Protocol and Generic Attribute Profile.
 - **Host Controller Interface (HCI)**: the interface between the host and controller either through software API or by a hardware interface such as SPI, UART or USB.

Subsequent chapters in this manual will go into the details of the implementation of each component, APIs available, and things to consider while designing a NimBLE app.

7.4 Example NimBLE projects

Mynewt comes with two built-in projects that allow users to play with NimBLE, try the tutorials out with, and see how to use available services.

1. **btshell**: A simple shell application which provides a basic interface to the host-side of the BLE stack.
2. **bleprph**: A basic peripheral device with no user interface. It advertises automatically on startup, and resumes advertising whenever a connection is terminated. It supports a maximum of one connection.
3. **blemesh**: A sample application for Bluetooth Mesh Node using on/off model.

7.5 NimBLE Security

The Bluetooth Low Energy security model includes five distinct security concepts as listed below. For detailed specifications, see BLUETOOTH SPECIFICATION Version 4.2 [Vol 1, Part A].

- **Pairing**: The process for creating one or more shared secret keys. In LE a single link key is generated by combining contributions from each device into a link key used during pairing.
- **Bonding**: The act of storing the keys created during pairing for use in subsequent connections in order to form a trusted device pair.
- **Device authentication**: Verification that the two devices have the same keys (verify device identity)
- **Encryption**: Keeps message confidential. Encryption in Bluetooth LE uses AES-CCM cryptography and is performed in the *Controller*.
- **Message integrity**: Protects against message forgeries.

Bluetooth LE uses four association models depending on the I/O capabilities of the devices.

- **Just Works**: designed for scenarios where at least one of the devices does not have a display capable of displaying a six digit number nor does it have a keyboard capable of entering six decimal digits.
- **Numeric Comparison**: designed for scenarios where both devices are capable of displaying a six digit number and both are capable of having the user enter “yes” or “no”. A good example of this model is the cell phone / PC scenario.
- **Out of Band**: designed for scenarios where an Out of Band mechanism is used to both discover the devices as well as to exchange or transfer cryptographic numbers used in the pairing process.
- **Passkey Entry**: designed for the scenario where one device has input capability but does not have the capability to display six digits and the other device has output capabilities. A good example of this model is the PC and keyboard scenario.

7.5.1 Key Generation

Key generation for all purposes in Bluetooth LE is performed by the *Host* on each LE device independent of any other LE device.

7.5.2 Privacy Feature

Bluetooth LE supports an optional feature during connection mode and connection procedures that reduces the ability to track a LE device over a period of time by changing the Bluetooth device address on a frequent basis.

There are two variants of the privacy feature.

- In the first variant, private addresses are resolved and generated by the *Host*.
- In the second variant, private addresses are resolved and generated by the *Controller* without involving the Host after the Host provides the Controller device identity information. The Host may provide the Controller with a complete resolving list or a subset of the resolving list. Device filtering becomes possible in the second variant when address resolution is performed in the Controller because the peer's device identity address can be resolved prior to checking whether it is in the white list.

Note: When address resolution is performed exclusively in the Host, a device may experience increased power consumption because device filtering must be disabled. For more details on the privacy feature, refer to BLUETOOTH SPECIFICATION Version 4.2 [Vol 3, Part C] (Published 02 December 2014), Page 592.

7.6 NimBLE Setup

Most NimBLE initialization is done automatically by `sysinit`. This section documents the few bits of initialization that an application must perform manually.

7.6.1 Configure clock for controller

The NimBLE stack uses OS cputime for scheduling various events inside controller. Since the code of controller is optimized to work with 32768 Hz clock, the OS cputime has to be configured accordingly.

To make things easier, controller package (`net/nimble/controller`) defines new system configuration setting `BLE_LP_CLOCK` as sets it to 1 so other packages can be configured if necessary. The next section describes configuration required for controller to work properly.

System configuration

Note: All BSPs based on nRF5x have below settings automatically applied when `BLE_LP_CLOCK` is set, there is no need to configure this in application.

The following things need to be configured for NimBLE controller to work properly:

- OS cputime frequency shall be set to 32768
- OS cputime timer source shall be set to 32768 Hz clock source
- Default 1 MHz clock source can be disabled if not used by application
- 32768 Hz clock source shall be enabled
- Crystal settling time shall be set to non-zero value (see below)

For example, on nRF52 platform timer 5 can be used as source for 32768 Hz clock. Also, timer 0 can be disabled since this is the default source for OS cputime clock and is no longer used. The configuration will look as below:

```
syscfg.vals:  
OS_CPUTIME_FREQ: 32768  
OS_CPUTIME_TIMER_NUM: 5
```

(continues on next page)

(continued from previous page)

```
TIMER_0: 0
TIMER_5: 1
BLE_XTAL_SETTLE_TIME: 1500
```

On nRF51 platform the only difference is to use timer 3 instead of timer 5.

On platforms without 32768 Hz crystal available it usually can be synthesized by setting XTAL_32768_SYNTH to 1 - this is also already configured in existing BSPs.

Crystal settle time configuration

The configuration variable BLE_XTAL_SETTLE_TIME is used by the controller to turn on the necessary clock source(s) for the radio and associated peripherals prior to Bluetooth events (advertising, scanning, connections, etc). For the nRF5x platforms, the HFXO needs to be turned on prior to using the radio and the BLE_XTAL_SETTLE_TIME must be set to accommodate this time. The amount of time required is board dependent, so users must characterize their hardware and set BLE_XTAL_SETTLE_TIME accordingly. The current value of 1500 microseconds is a fairly long time and was intended to work for most, if not all, platforms.

Note that changing this time will impact battery life with the amount depending on the application. The HFXO draws a fairly large amount of current when running so keeping this time as small as possible will reduce overall current drain.

7.6.2 Configure device address

A BLE device needs an address to do just about anything. For information on the various types of Bluetooth addresses, see the *NimBLE Host Identity Reference* :doc:```<./ble_hs/ble_hs_id/ble_hs_id>`.

There are several methods for assigning an address to a NimBLE device. The available options are documented below:

Method 1: Configure nRF hardware with a public address

When Mynewt is running on a Nordic nRF platform, the NimBLE controller will attempt to read a public address out of the board's FICR or UICR registers. The controller uses the following logic while trying to read an address from hardware:

1. If the DEVICEADDRTYPE FICR register is written, read the address programmed in the DEVICEADDR[0] and DEVICEADDR[1] FICR registers.
2. Else if the upper 16 bits of the CUSTOMER[1] UICR register are 0, read the address programmed in the CUSTOMER[0] and CUSTOMER[1] UCI registers.
3. Else, no address available.

Method 2: Hardcode a public address in the Mynewt target

The NimBLE controller package exports a `syscfg` setting called BLE_PUBLIC_DEV_ADDR. This setting can be overridden at the application or target level to configure a public Bluetooth address. For example, a target can assign the public address 11:22:33:44:55:66 as follows:

```
syscfg.vals:
    BLE_PUBLIC_DEV_ADDR: '(uint8_t[6]){0x66, 0x55, 0x44, 0x33, 0x22, 0x11}'
```

This setting takes the form of a C expression. Specifically, the value is a designated initializer expressing a six-byte array. Also note that the bytes are reversed, as an array is inherently little-endian, while addresses are generally expressed in big-endian.

Note: this method takes precedence over method 1. Whatever is written to the `BLE_PUBLIC_DEV_ADDR` setting is the address that gets used.

Method 3: Configure a random address at runtime

Random addresses get configured through the NimBLE host. The following two functions are used in random address configuration:

- `ble_hs_id_gen_rnd()`: Generates a new random address.
- `ble_hs_id_set_rnd()`: Sets the device's random address.

For an example of how this is done, see the [BLE iBeacon](#).

Note: A NimBLE device can be configured with multiple addresses; at most one of each address type.

7.6.3 Respond to sync and reset events

sync

The NimBLE stack is inoperable while the host and controller are out of sync. In a combined host-controller app, the sync happens immediately at startup. When the host and controller are separate, sync typically occurs in under a second after the application starts. An application learns when sync is achieved by configuring the host's *sync callback*: `ble_hs_cfg.sync_cb`. The host calls the sync callback whenever sync is acquired. The sync callback has the following form:

```
typedef void ble_hs_sync_fn(void);
```

Because the NimBLE stack begins in the unsynced state, the application should delay all BLE operations until the sync callback has been called.

reset

Another event indicated by the host is a *controller reset*. The NimBLE stack resets itself when a catastrophic error occurs, such as loss of communication between the host and controller. Upon resetting, the host drops all BLE connections and loses sync with the controller. After a reset, the application should refrain from using the host until sync is again signaled via the sync callback.

An application learns of a host reset by configuring the host's *reset callback*: `ble_hs_cfg.reset_cb`. This callback has the following form:

```
typedef void ble_hs_reset_fn(int reason);
```

The `reason` parameter is a [NimBLE host return code](#).

Example

The following example demonstrates the configuration of the sync and reset callbacks.

```
#include "sysinit/sysinit.h"
#include "console/console.h"
#include "host/ble_hs.h"

static void
on_sync(void)
{
    /* Begin advertising, scanning for peripherals, etc. */
}

static void
on_reset(int reason)
{
    console_printf("Resetting state; reason=%d\n", reason);
}

int
mynewt_main(int argc, char **argv)
{
    /* Initialize all packages. */
    sysinit();

    ble_hs_cfg.sync_cb = on_sync;
    ble_hs_cfg.reset_cb = on_reset;

    /* As the last thing, process events from default event queue. */
    while (1) {
        os_eventq_run(os_eventq_dflt_get());
    }
}
```

7.7 NimBLE Host

7.7.1 Introduction

At a high level, the NimBLE stack is divided into two components:

- Host
- Controller

This document is an API reference for the host component. If you are interested in the general structure of the NimBLE stack and its non-host components, you might want to read the [BLE User Guide](#).

The host sits directly below the application, and it serves as the interface to the application for all BLE operations.

7.7.2 NimBLE Host Return Codes

- *Introduction*
 - *Summary*
 - *Example*
- *Return Code Reference*
 - *Header*
 - *Return codes - Core*
 - *Return codes - ATT*
 - *Return codes - HCI*
 - *Return codes - L2CAP*
 - *Return codes - Security manager (us)*
 - *Return codes - Security manager (peer)*

Introduction

Summary

The NimBLE host reports status to the application via a set of return codes. The host encompasses several layers of the Bluetooth specification that each defines its own set of status codes. Rather than “abstract away” information from lower layers that the application developer might find useful, the NimBLE host aims to indicate precisely what happened when something fails. Consequently, the host utilizes a rather large set of return codes.

A return code of 0 indicates success. For failure conditions, the return codes are partitioned into five separate sets:

Set	Condition
Core	Errors detected internally by the NimBLE host.
ATT	The ATT server has reported a failure via the transmission of an ATT Error Response. The return code corresponds to the value of the Error Code field in the response.
HCI	The controller has reported an error to the host via a command complete or command status HCI event. The return code corresponds to the value of the Status field in the event.
L2CAP	An L2CAP signaling procedure has failed and an L2CAP Command Reject was sent as a result. The return code corresponds to the value of the Reason field in the command.
Security manager (us)	The host detected an error during a security manager procedure and sent a Pairing Failed command to the peer. The return code corresponds to the value of the Reason field in the Pairing Failed command.
Security manager (peer)	A security manager procedure failed because the peer sent us a Pairing Failed command. The return code corresponds to the value of the Reason field in the Pairing Failed command.

The return codes in the core set are defined by the NimBLE Host. The other sets are defined in the Bluetooth specification; the codes in this latter group are referred to as *formal status codes*. As defined in the Bluetooth specification, the formal status code sets are not disjoint. That is, they overlap. For example, the spec defines a status code of 1 to have all of the following meanings:

Layer	Meaning
ATT	Invalid handle.
HCI	Unknown HCI command.
L2CAP	Signalling MTU exceeded.
SM	Passkey entry failed.

Clearly, the host can't just return an unadorned formal status code and expect the application to make sense of it. To resolve this ambiguity, the NimBLE host divides the full range of an int into several subranges. Each subrange corresponds to one of the five return code sets. For example, the ATT set is mapped onto the subrange [0x100, 0x200). To indicate an ATT error of 3 (write not permitted), the NimBLE host returns a value 0x103 to the application.

The host defines a set of convenience macros for converting from a formal status code to NimBLE host status code. These macros are documented in the table below.

Macro	Status code set	Base value
BLE_HS_ATT_ERR()	ATT	0x100
BLE_HS_HCI_ERR()	HCI	0x200
BLE_HS_L2C_ERR()	L2CAP	0x300
BLE_HS_SM_US_ERR()	Security manager (us)	0x400
BLE_HS_SM_PEER_ERR()	Security manager (peer)	0x500

Example

The following example demonstrates how an application might determine which error is being reported by the host. In this example, the application performs the GAP encryption procedure and checks the return code. To simplify the example, the application uses a hypothetical *my_blocking_enc_proc()* function, which blocks until the pairing operation has completed.

```
void
encrypt_connection(uint16_t conn_handle)
{
    int rc;

    /* Perform a blocking GAP encryption procedure. */
    rc = my_blocking_enc_proc(conn_handle);
    switch (rc) {
    case 0:
        console_printf("success - link successfully encrypted\n");
        break;

    case BLE_HS_ENOTCONN:
        console_printf("failure - no connection with handle %d\n",
                      conn_handle);
        break;

    case BLE_HS_ERR_SM_US_BASE(BLE_SM_ERR_CONFIRM_MISMATCH):
        console_printf("failure - mismatch in peer's confirm and random "
                      "commands.\n");
        break;
    }
```

(continues on next page)

(continued from previous page)

```
case BLE_HS_ERR_SM_PEER_BASE(BLE_SM_ERR_CONFIRM_MISMATCH):
    console_printf("failure - peer reports mismatch in our confirm and "
                   "random commands.\n");
    break;

default:
    console_printf("failure - other error: 0x%04x\n", rc);
    break;
}
```

Return Code Reference

Header

All NimBLE host return codes are made accessible by including the following header:

```
#include "host/ble_hs.h"
```

Return codes - Core

The precise meaning of each of these error codes depends on the function that returns it. The API reference for a particular function indicates the conditions under which each of these codes are returned.

Table 1: Return codes - Core

Value	Name	Condition
0x00	N/A	Success
0x01	BLE_HS_EAGAIN	Temporary failure; try again.
0x02	BLE_HS_EALREADY	Operation already in progress or completed.
0x03	BLE_HS_EINVAL	One or more arguments are invalid.
0x04	BLE_HS_EMSGSIZE	The provided buffer is too small.
0x05	BLE_HS_ENOENT	No entry matching the specified criteria.
0x06	BLE_HS_ENOMEM	Operation failed due to resource exhaustion.
0x07	BLE_HS_ENOTCONN	No open connection with the specified handle.
0x08	BLE_HS_ENOTSUP	Operation disabled at compile time.
0x09	BLE_HS_EAPP	Application callback behaved unexpectedly.
0x0a	BLE_HS_EBADDATA	Command from peer is invalid.
0x0b	BLE_HS_EOS	Mynewt OS error.
0x0c	BLE_HS_ECONTROLLER	Event from controller is invalid.
0x0d	BLE_HSETIMEOUT	Operation timed out.
0x0e	BLE_HS_EDONE	Operation completed successfully.
0x0f	BLE_HS_EBUSY	Operation cannot be performed until procedure completes.
0x10	BLE_HS_EREJECT	Peer rejected a connection parameter update request.
0x11	BLE_HS_EUNKNOWN	Unexpected failure; catch all.
0x12	BLE_HS_EROLE	Operation requires different role (e.g., central vs. peripheral).
0x13	BLE_HSETIMEOUT_HCI	HCI request timed out; controller unresponsive.
0x14	BLE_HS_ENOMEM_EVT	Controller failed to send event due to memory exhaustion (combined host-controller only).
0x15	BLE_HS_ENOADDR	Operation requires an identity address but none configured.
0x16	BLE_HS_ENOTSYNCED	Attempt to use the host before it is synced with controller.
0x17	BLE_HS_EAUTHEN	Insufficient authentication.
0x18	BLE_HS_EAUTHOR	Insufficient authorization.
0x19	BLE_HS_EENCRYPT	Insufficient encryption level.
0x1a	BLE_HS_EENCRYPT_KEY_SZ	Insufficient key size.
0x1b	BLE_HS_ESTORE_CAP	Storage at capacity.
0x1c	BLE_HS_ESTORE_FAIL	Storage IO error.

Return codes - ATT

Table 2: Return codes - ATT

NimBLE Value	Formal Value	Name	Condition
0x0101	0x01	BLE_ATT_ERR_INVALID_HANDLE	The attribute handle given was not valid on this server.
0x0102	0x02	BLE_ATT_ERR_READ_NOT_PERMITTED	The attribute cannot be read.
0x0103	0x03	BLE_ATT_ERR_WRITE_NOT_PERMITTED	The attribute cannot be written.
0x0104	0x04	BLE_ATT_ERR_INVALID_PDU	The attribute PDU was invalid.
0x0105	0x05	BLE_ATT_ERR_INSUFFICIENT_AUTHEN	The attribute requires authentication before it can be read or written.
0x0106	0x06	BLE_ATT_ERR_REQ_NOT_SUPPORTED	Attribute server does not support the request received from the client.
0x0107	0x07	BLE_ATT_ERR_INVALID_OFFSET	Offset specified was past the end of the attribute.
0x0108	0x08	BLE_ATT_ERR_INSUFFICIENT_AUTHOR	The attribute requires authorization before it can be read or written.
0x0109	0x09	BLE_ATT_ERR_PREPARE_QUEUE_FULL	Too many prepare writes have been queued.
0x010a	0x0a	BLE_ATT_ERR_ATTR_NOT_FOUND	No attribute found within the given attribute handle range.
0x010b	0x0b	BLE_ATT_ERR_ATTR_NOT_LONG	The attribute cannot be read or written using the Read Blob Request.
0x010c	0x0c	BLE_ATT_ERR_INSUFFICIENT_KEY_SZ	The Encryption Key Size used for encrypting this link is insufficient.
0x010d	0x0d	BLE_ATT_ERR_INVALID_ATTR_VALUE_LEN	The attribute value length is invalid for the operation.
0x010e	0x0e	BLE_ATT_ERR_UNLIKELY	The attribute request that was requested has encountered an error that was unlikely, and therefore could not be completed as requested.
0x010f	0x0f	BLE_ATT_ERR_INSUFFICIENT_ENC	The attribute requires encryption before it can be read or written.
0x0110	0x10	BLE_ATT_ERR_UNSUPPORTED_GROUP	The attribute type is not a supported grouping attribute as defined by a higher layer specification.
0x0111	0x11	BLE_ATT_ERR_INSUFFICIENT_RES	Insufficient Resources to complete the request.

Return codes - HCI

Table 3: Return codes - HCI

NimBLE Value	Formal Value	Name	Condition
0x0201	0x01	BLE_ERR_UNKNOWN_HCI_CMD	Unknown HCI Command
0x0202	0x02	BLE_ERR_UNK_CONN_ID	Unknown Connection Identifier
0x0203	0x03	BLE_ERR_HW_FAIL	Hardware Failure

continues on next page

Table 3 – continued from previous page

NimBLE Value	Formal Value	Name	Condition
0x0204	0x04	BLE_ERR_PAGE_TMO	Page Timeout
0x0205	0x05	BLE_ERR_AUTH_FAIL	Authentication Failure
0x0206	0x06	BLE_ERR_PINKEY_MISSING	PIN or Key Missing
0x0207	0x07	BLE_ERR_MEM_CAPACITY	Memory Capacity Exceeded
0x0208	0x08	BLE_ERR_CONN_SPVN_TMO	Connection Timeout
0x0209	0x09	BLE_ERR_CONN_LIMIT	Connection Limit Exceeded
0x020a	0xa	BLE_ERR_SYNCH_CONN_LIMIT	Synchronous Connection Limit To A Device Exceeded
0x020b	0xb	BLE_ERR_ACL_CONN_EXISTS	ACL Connection Already Exists
0x020c	0xc	BLE_ERR_CMD_DISALLOWED	Command Disallowed
0x020d	0xd	BLE_ERR_CONN_REJ_RESOURCES	Connection Rejected due to Limited Resources
0x020e	0xe	BLE_ERR_CONN_REJ_SECURITY	Connection Rejected Due To Security Reasons
0x020f	0xf	BLE_ERR_CONN_REJ_BD_ADDR	Connection Rejected due to Unacceptable BD_ADDR
0x0210	0x10	BLE_ERR_CONN_ACCEPT_TMO	Connection Accept Timeout Exceeded
0x0211	0x11	BLE_ERR_UNSUPPORTED	Unsupported Feature or Parameter Value
0x0212	0x12	BLE_ERR_INV_HCI_CMD_PARMS	Invalid HCI Command Parameters
0x0213	0x13	BLE_ERR_Rem_USER_CONN_TERM	Remote User Terminated Connection
0x0214	0x14	BLE_ERR_RD_CONN_TERM_RESRCS	Remote Device Terminated Connection due to Low Resources
0x0215	0x15	BLE_ERR_RD_CONN_TERM_PWROFF	Remote Device Terminated Connection due to Power Off
0x0216	0x16	BLE_ERR_CONN_TERM_LOCAL	Connection Terminated By Local Host
0x0217	0x17	BLE_ERR_REPEAT_ATTEMPTS	Repeated Attempts
0x0218	0x18	BLE_ERR_NO_PAIRING	Pairing Not Allowed
0x0219	0x19	BLE_ERR_UNK_LMP	Unknown LMP PDU
0x021a	0xa	BLE_ERR_UNSUPP_Rem_FEATURE	Unsupported Remote Feature / Unsupported LMP Feature
0x021b	0xb	BLE_ERR_SCO_OFFSET	SCO Offset Rejected
0x021c	0xc	BLE_ERR_SCO_ITVL	SCO Interval Rejected
0x021d	0xd	BLE_ERR_SCO_AIR_MODE	SCO Air Mode Rejected
0x021e	0xe	BLE_ERR_INV_LMP_LL_PARM	Invalid LMP Parameters / Invalid LL Parameters
0x021f	0xf	BLE_ERR_UNSPECIFIED	Unspecified Error
0x0220	0x20	BLE_ERR_UNSUPP_LMP_LL_PARM	Unsupported LMP Parameter Value / Unsupported LL Parameter Value
0x0221	0x21	BLE_ERR_NO_ROLE_CHANGE	Role Change Not Allowed
0x0222	0x22	BLE_ERR_LMP_LL_RSP_TMO	LMP Response Timeout / LL Response Timeout
0x0223	0x23	BLE_ERR_LMP_COLLISION	LMP Error Transaction Collision
0x0224	0x24	BLE_ERR_LMP_PDU	LMP PDU Not Allowed
0x0225	0x25	BLE_ERR_ENCRYPTION_MODE	Encryption Mode Not Acceptable
0x0226	0x26	BLE_ERR_LINK_KEY_CHANGE	Link Key cannot be Changed
0x0227	0x27	BLE_ERR_UNSUPP_QOS	Requested QoS Not Supported
0x0228	0x28	BLE_ERR_INSTANT_PASSED	Instant Passed
0x0229	0x29	BLE_ERR_UNIT_KEY_PAIRING	Pairing With Unit Key Not Supported
0x022a	0x2a	BLE_ERR_DIFF_TRANS_COLL	Different Transaction Collision

continues on next page

Table 3 – continued from previous page

NimBLE Value	Formal Value	Name	Condition
0x022c	0x2c	BLE_ERR_QOS_PARM	QoS Unacceptable Parameter
0x022d	0x2d	BLE_ERR_QOS_REJECTED	QoS Rejected
0x022e	0x2e	BLE_ERR_CHAN_CLASS	Channel Classification Not Supported
0x022f	0x2f	BLE_ERR_INSUFFICIENT_SEC	Insufficient Security
0x0230	0x30	BLE_ERR_PARM_OUT_OF_RANGE	Parameter Out Of Mandatory Range
0x0232	0x32	BLE_ERR_PENDING_ROLE_SW	Role Switch Pending
0x0234	0x34	BLE_ERR_RESERVED_SLOT	Reserved Slot Violation
0x0235	0x35	BLE_ERR_ROLE_SW_FAIL	Role Switch Failed
0x0236	0x36	BLE_ERR_INQ_RSP_TOO_BIG	Extended Inquiry Response Too Large
0x0237	0x37	BLE_ERR_SEC_SIMPLE_PAIR	Secure Simple Pairing Not Supported By Host
0x0238	0x38	BLE_ERR_HOST_BUSY_PAIR	Host Busy - Pairing
0x0239	0x39	BLE_ERR_CONN rej CHANNEL	Connection Rejected due to No Suitable Channel Found
0x023a	0x3a	BLE_ERR_CTLR_BUSY	Controller Busy
0x023b	0x3b	BLE_ERR_CONN_PARMS	Unacceptable Connection Parameters
0x023c	0x3c	BLE_ERR_DIR_ADV_TMO	Directed Advertising Timeout
0x023d	0x3d	BLE_ERR_CONN_TERM_MIC	Connection Terminated due to MIC Failure
0x023e	0x3e	BLE_ERR_CONN_ESTABLISHMENT	Connection Failed to be Established
0x023f	0x3f	BLE_ERR_MAC_CONN_FAIL	MAC Connection Failed
0x0240	0x40	BLE_ERR_COARSE_CLK_ADJ	Coarse Clock Adjustment Rejected but Will Try to Adjust Using Clock Dragging

Return codes - L2CAP

Table 4: Return codes - L2CAP

NimBLE Value	Formal Value	Name	Condition
0x0300	0x00	BLE_L2CAP_SIG_ERR_CMD_NOT_UNDERSTOOD	Invalid or unsupported incoming L2CAP sig command.
0x0301	0x01	BLE_L2CAP_SIG_ERR_MTU_EXCEEDED	Incoming packet too large.
0x0302	0x02	BLE_L2CAP_SIG_ERR_INVALID_CID	No channel with specified ID.

Return codes - Security manager (us)

Table 5: Return codes - Security manager (us)

NimBLE Value	Formal Value	Name	Condition
0x0401	0x01	BLE_SM_ERR_PASSKEY	The user input of passkey failed, for example, the user cancelled the operation.
0x0402	0x02	BLE_SM_ERR_OOB	The OOB data is not available.
0x0403	0x03	BLE_SM_ERR_AUTHREQ	The pairing procedure cannot be performed as authentication requirements cannot be met due to IO capabilities of one or both devices.
0x0404	0x04	BLE_SM_ERR_CONFIRM_MISMATCH	The confirm value does not match the calculated compare value.
0x0405	0x05	BLE_SM_ERR_PAIR_NOT_SUPP	Pairing is not supported by the device.
0x0406	0x06	BLE_SM_ERR_ENC_KEY_SZ	The resultant encryption key size is insufficient for the security requirements of this device.
0x0407	0x07	BLE_SM_ERR_CMD_NOT_SUPP	The SMP command received is not supported on this device.
0x0408	0x08	BLE_SM_ERR_UNSPECIFIED	Pairing failed due to an unspecified reason.
0x0409	0x09	BLE_SM_ERR_REPEAT	Pairing or authentication procedure is disallowed because too little time has elapsed since last pairing request or security request.
0x040a	0xa	BLE_SM_ERR_INVAL	The Invalid Parameters error code indicates that the command length is invalid or that a parameter is outside of the specified range.
0x040b	0xb	BLE_SM_ERR_DHKEY	Indicates to the remote device that the DHKey Check value received doesn't match the one calculated by the local device.
0x040c	0xc	BLE_SM_ERR_NUMCMP	Indicates that the confirm values in the numeric comparison protocol do not match.
0x040d	0xd	BLE_SM_ERR_ALREADY	Indicates that the pairing over the LE transport failed due to a Pairing Request sent over the BR/EDR transport in process.
0x040e	0xe	BLE_SM_ERR_CROSS_TRANS	Indicates that the BR/EDR Link Key generated on the BR/EDR transport cannot be used to derive and distribute keys for the LE transport.

Return codes - Security manager (peer)

Table 6: Return codes - Security manager (peer)

NimBLE Value	Formal Value	Name	Condition
0x0501	0x01	BLE_SM_ERR_PASSKEY	The user input of passkey failed, for example, the user cancelled the operation.
0x0502	0x02	BLE_SM_ERR_OOB	The OOB data is not available.
0x0503	0x03	BLE_SM_ERR_AUTHREQ	The pairing procedure cannot be performed as authentication requirements cannot be met due to IO capabilities of one or both devices.
0x0504	0x04	BLE_SM_ERR_CONFIRM_MISMATCH	The confirm value does not match the calculated compare value.
0x0505	0x05	BLE_SM_ERR_PAIR_NOT_SUPP	Pairing is not supported by the device.
0x0506	0x06	BLE_SM_ERR_ENC_KEY_SZ	The resultant encryption key size is insufficient for the security requirements of this device.
0x0507	0x07	BLE_SM_ERR_CMD_NOT_SUPP	The SMP command received is not supported on this device.
0x0508	0x08	BLE_SM_ERR_UNSPECIFIED	Pairing failed due to an unspecified reason.
0x0509	0x09	BLE_SM_ERR_REPEAT	Pairing or authentication procedure is disallowed because too little time has elapsed since last pairing request or security request.
0x050a	0xa	BLE_SM_ERR_INVAL	The Invalid Parameters error code indicates that the command length is invalid or that a parameter is outside of the specified range.
0x050b	0xb	BLE_SM_ERR_DHKEY	Indicates to the remote device that the DHKey Check value received doesn't match the one calculated by the local device.
0x050c	0xc	BLE_SM_ERR_NUMCMP	Indicates that the confirm values in the numeric comparison protocol do not match.
0x050d	0xd	BLE_SM_ERR_ALREADY	Indicates that the pairing over the LE transport failed due to a Pairing Request sent over the BR/EDR transport in process.
0x050e	0xe	BLE_SM_ERR_CROSS_TRANS	Indicates that the BR/EDR Link Key generated on the BR/EDR transport cannot be used to derive and distribute keys for the LE transport.

7.7.3 NimBLE Host GAP Reference

Introduction

The Generic Access Profile (GAP) is responsible for all connecting, advertising, scanning, and connection updating operations.

API

`typedef int ble_gap_event_fn(struct ble_gap_event *event, void *arg)`

Callback function type for handling BLE GAP events.

`typedef int ble_gap_conn_foreach_handle_fn(uint16_t conn_handle, void *arg)`

Callback function type for iterating through BLE connection handles.

`int ble_gap_conn_find(uint16_t handle, struct ble_gap_conn_desc *out_desc)`

Searches for a connection with the specified handle.

If a matching connection is found, the supplied connection descriptor is filled correspondingly.

Parameters

- **handle** – The connection handle to search for.
- **out_desc** – On success, this is populated with information relating to the matching connection. Pass NULL if you don't need this information.

Returns

0 on success, BLE_HS_ENOTCONN if no matching connection was found.

`int ble_gap_conn_find_by_addr(const ble_addr_t *addr, struct ble_gap_conn_desc *out_desc)`

Searches for a connection with a peer with the specified address.

If a matching connection is found, the supplied connection descriptor is filled correspondingly.

Parameters

- **addr** – The ble address of a connected peer device to search for.
- **out_desc** – On success, this is populated with information relating to the matching connection. Pass NULL if you don't need this information.

Returns

0 on success, BLE_HS_ENOTCONN if no matching connection was found.

`int ble_gap_set_event_cb(uint16_t conn_handle, ble_gap_event_fn *cb, void *cb_arg)`

Configures a connection to use the specified GAP event callback.

A connection's GAP event callback is first specified when the connection is created, either via advertising or initiation. This function replaces the callback that was last configured.

Parameters

- **conn_handle** – The handle of the connection to configure.
- **cb** – The callback to associate with the connection.
- **cb_arg** – An optional argument that the callback receives.

Returns

0 on success, BLE_HS_ENOTCONN if there is no connection with the specified handle.

`int ble_gap_adv_start(uint8_t own_addr_type, const ble_addr_t *direct_addr, int32_t duration_ms, const struct ble_gap_adv_params *adv_params, ble_gap_event_fn *cb, void *cb_arg)`

Start advertising.

This function configures and start advertising procedure.

Parameters

- **own_addr_type** – The type of address the stack should use for itself. Valid values are:

- BLE_OWN_ADDR_PUBLIC
 - BLE_OWN_ADDR_RANDOM
 - BLE_OWN_ADDR_RPA_PUBLIC_DEFAULT
 - BLE_OWN_ADDR_RPA_RANDOM_DEFAULT
- **direct_addr** – The peer’s address for directed advertising. This parameter shall be non-NULL if directed advertising is being used.
 - **duration_ms** – The duration of the advertisement procedure. On expiration, the procedure ends and a BLE_GAP_EVENT_ADV_COMPLETE event is reported. Units are milliseconds. Specify BLE_HS_FOREVER for no expiration.
 - **adv_params** – Additional arguments specifying the particulars of the advertising procedure.
 - **cb** – The callback to associate with this advertising procedure. If advertising ends, the event is reported through this callback. If advertising results in a connection, the connection inherits this callback as its event-reporting mechanism.
 - **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success, error code on failure.

`int ble_gap_adv_stop(void)`

Stops the currently-active advertising procedure.

A success return code indicates that advertising has been fully aborted and a new advertising procedure can be initiated immediately.

NOTE: If the caller is running in the same task as the NimBLE host, or if it is running in a higher priority task than that of the host, care must be taken when restarting advertising. Under these conditions, the following is *not* a reliable method to restart advertising: `ble_gap_adv_stop()` `ble_gap_adv_start()`

Instead, the call to `ble_gap_adv_start()` must be made in a separate event context. That is, `ble_gap_adv_start()` must be called asynchronously by enqueueing an event on the current task’s event queue. See <https://github.com/apache/mynewt-nimble/pull/211> for more information.

Returns

0 on success, BLE_HS_EALREADY if there is no active advertising procedure, other error code on failure.

`int ble_gap_adv_active(void)`

Indicates whether an advertisement procedure is currently in progress.

Returns

0 if no advertisement procedure in progress, 1 otherwise.

`int ble_gap_adv_set_data(const uint8_t *data, int data_len)`

Configures the data to include in subsequent advertisements.

Parameters

- **data** – Buffer containing the advertising data.
- **data_len** – The size of the advertising data, in bytes.

Returns

0 on success, BLE_HS_EBUSY if advertising is in progress, other error code on failure.

```
int ble_gap_adv_rsp_set_data(const uint8_t *data, int data_len)
```

Configures the data to include in subsequent scan responses.

Parameters

- **data** – Buffer containing the scan response data.
- **data_len** – The size of the response data, in bytes.

Returns

0 on success, BLE_HS_EBUSY if advertising is in progress, other error code on failure.

```
int ble_gap_adv_set_fields(const struct ble_hs_adv_fields *adv_fields)
```

Configures the fields to include in subsequent advertisements.

This is a convenience wrapper for [*ble_gap_adv_set_data\(\)*](#).

Parameters

- **adv_fields** – Specifies the advertisement data.

Returns

0 on success, BLE_HS_EBUSY if advertising is in progress, BLE_HS_EMSGSIZE if the specified data is too large to fit in an advertisement, other error code on failure.

```
int ble_gap_adv_rsp_set_fields(const struct ble_hs_adv_fields *rsp_fields)
```

Configures the fields to include in subsequent scan responses.

This is a convenience wrapper for [*ble_gap_adv_rsp_set_data\(\)*](#).

Parameters

- **rsp_fields** – Specifies the scan response data.

Returns

0 on success, BLE_HS_EBUSY if advertising is in progress, BLE_HS_EMSGSIZE if the specified data is too large to fit in a scan response, other error code on failure.

```
int ble_gap_ext_adv_configure(uint8_t instance, const struct ble_gap_ext_adv_params *params, int8_t
                             *selected_tx_power, ble_gap_event_fn *cb, void *cb_arg)
```

Configure extended advertising instance.

Parameters

- **instance** – Instance ID
- **params** – Additional arguments specifying the particulars of the advertising.
- **selected_tx_power** – Selected advertising transmit power will be stored in that param if non-NULL.
- **cb** – The callback to associate with this advertising procedure. Advertising complete event is reported through this callback
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gap_ext_adv_set_addr(uint8_t instance, const ble_addr_t *addr)
```

Set random address for configured advertising instance.

Parameters

- **instance** – Instance ID

- **addr** – Random address to be set

Returns

0 on success; nonzero on failure.

int ble_gap_ext_adv_start(uint8_t instance, int duration, int max_events)

Start advertising instance.

Parameters

- **instance** – Instance ID
- **duration** – The duration of the advertisement procedure. On expiration, the procedure ends and a BLE_GAP_EVENT_ADV_COMPLETE event is reported. Units are 10 milliseconds. Specify 0 for no expiration. @params max_events Number of advertising events that should be sent before advertising ends and a BLE_GAP_EVENT_ADV_COMPLETE event is reported. Specify 0 for no limit.

Returns

0 on success, error code on failure.

int ble_gap_ext_adv_stop(uint8_t instance)

Stops advertising procedure for specified instance.

Parameters

- **instance** – Instance ID

Returns

0 on success, BLE_HS_EALREADY if there is no active advertising procedure for instance, other error code on failure.

int ble_gap_ext_adv_set_data(uint8_t instance, struct *os_mbuf* *data)

Configures the data to include in advertisements packets for specified advertising instance.

Parameters

- **instance** – Instance ID
- **data** – Chain containing the advertising data.

Returns

0 on success or error code on failure.

int ble_gap_ext_adv_rsp_set_data(uint8_t instance, struct *os_mbuf* *data)

Configures the data to include in subsequent scan responses for specified advertising instance.

Parameters

- **instance** – Instance ID
- **data** – Chain containing the scan response data.

Returns

0 on success or error code on failure.

int ble_gap_ext_adv_remove(uint8_t instance)

Remove existing advertising instance.

Parameters

- **instance** – Instance ID

Returns

0 on success, BLE_HS_EBUSY if advertising is in progress, other error code on failure.

```
int ble_gap_ext_adv_clear(void)
```

Clear all existing advertising instances.

Returns

0 on success, BLE_HS_EBUSY if advertising is in progress, other error code on failure.

```
int ble_gap_ext_adv_active(uint8_t instance)
```

Indicates whether an advertisement procedure is currently in progress on the specified Instance.

Parameters

- **instance** – Instance Id

Returns

0 if there is no active advertising procedure for the instance, 1 otherwise

```
int ble_gap_adv_get_free_instance(uint8_t *out_adv_instance)
```

Finds first not configured advertising instance.

Parameters

- **out_adv_instance** – [out] Pointer to be filled with found advertising instance

Returns

0 if free advertising instance was found, error code otherwise

```
int ble_gap_periodic_adv_configure(uint8_t instance, const struct ble_gap_periodic_adv_params *params)
```

Configure periodic advertising for specified advertising instance.

This is allowed only for instances configured as non-announymous, non-connectable and non-scannable.

Parameters

- **instance** – Instance ID
- **params** – Additional arguments specifying the particulars of periodic advertising.

Returns

0 on success; nonzero on failure.

```
int ble_gap_periodic_adv_start(uint8_t instance, const struct ble_gap_periodic_adv_start_params *params)
```

Start periodic advertising for specified advertising instance.

Parameters

- **instance** – Instance ID
- **params** – Additional arguments specifying the particulars of periodic advertising.

Returns

0 on success, error code on failure.

```
int ble_gap_periodic_adv_stop(uint8_t instance)
```

Stop periodic advertising for specified advertising instance.

Parameters

- **instance** – Instance ID

Returns

0 on success, error code on failure.

```
int ble_gap_periodic_adv_set_data(uint8_t instance, struct os_mbuf *data, const struct  
                                ble_gap_periodic_adv_set_data_params *params)
```

Configures the data to include in periodic advertisements for specified advertising instance.

Parameters

- **instance** – Instance ID
- **data** – Chain containing the periodic advertising data.
- **params** – Additional arguments specifying the particulars of periodic advertising data.

Returns

0 on success or error code on failure.

```
int ble_gap_periodic_adv_sync_create(const ble_addr_t *addr, uint8_t adv_sid, const struct  
                                    ble_gap_periodic_sync_params *params, ble_gap_event_fn *cb, void  
                                    *cb_arg)
```

Schedule the Synchronization procedure with periodic advertiser.

Procedure is performed as soon as Extended Discovery procedure is started. If Extended Discovery is already active when issuing this procedure, it will be performed immediately. It is up to application to start Extended Discovery.

Parameters

- **addr** – Peer address to synchronize with. If NULL than peers from periodic list are used.
- **adv_sid** – Advertiser Set ID
- **params** – Additional arguments specifying the particulars of the synchronization procedure.
- **cb** – The callback to associate with this synchronization procedure. BLE_GAP_EVENT_PERIODIC_REPORT events are reported only by this callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gap_periodic_adv_sync_create_cancel(void)
```

Cancel pending synchronization procedure.

Returns

0 on success; nonzero on failure.

```
int ble_gap_periodic_adv_sync_terminate(uint16_t sync_handle)
```

Terminate synchronization procedure.

Parameters

- **sync_handle** – Handle identifying synchronization to terminate.

Returns

0 on success; nonzero on failure.

```
int ble_gap_periodic_adv_sync_reporting(uint16_t sync_handle, bool enable, const struct  
                                         ble_gap_periodic_adv_sync_reporting_params *params)
```

Disable or enable periodic reports for specified sync.

Parameters

- **sync_handle** – Handle identifying synchronization.
- **enable** – If reports should be enabled.

- **params** – Additional arguments specifying the particulars of periodic reports.

Returns

0 on success; nonzero on failure.

```
int ble_gap_periodic_adv_sync_transfer(uint16_t sync_handle, uint16_t conn_handle, uint16_t
                                         service_data)
```

Initialize sync transfer procedure for specified handles.

This allows to transfer periodic sync to which host is synchronized.

Parameters

- **sync_handle** – Handle identifying synchronization.
- **conn_handle** – Handle identifying connection.
- **service_data** – Sync transfer service data

Returns

0 on success; nonzero on failure.

```
int ble_gap_periodic_adv_sync_set_info(uint8_t instance, uint16_t conn_handle, uint16_t service_data)
```

Initialize set info transfer procedure for specified handles.

This allows to transfer periodic sync which is being advertised by host.

Parameters

- **instance** – Advertising instance with periodic adv enabled.
- **conn_handle** – Handle identifying connection.
- **service_data** – Sync transfer service data

Returns

0 on success; nonzero on failure.

```
int periodic_adv_set_default_sync_params(const struct ble_gap_periodic_sync_params *params)
```

Set the default periodic sync transfer params.

Parameters

- **conn_handle** – Handle identifying connection.
- **params** – Default Parameters for periodic sync transfer.

Returns

0 on success; nonzero on failure.

```
int ble_gap_periodic_adv_sync_receive(uint16_t conn_handle, const struct ble_gap_periodic_sync_params
                                         *params, ble_gap_event_fn *cb, void *cb_arg)
```

Enables or disables sync transfer reception on specified connection.

When sync transfer arrives, BLE_GAP_EVENT_PERIODIC_TRANSFER is sent to the user. After that, sync transfer reception on that connection is terminated and user needs to call this API again when expect to receive next sync transfers.

Note: If ACL connection gets disconnected before sync transfer arrived, user will not receive BLE_GAP_EVENT_PERIODIC_TRANSFER. Instead, sync transfer reception is terminated by the host automatically.

Parameters

- **conn_handle** – Handle identifying connection.

- **params** – Parameters for enabled sync transfer reception. Specify NULL to disable reception.
- **cb** – The callback to associate with this synchronization procedure. BLE_GAP_EVENT_PERIODIC_REPORT events are reported only by this callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gap_add_dev_to_periodic_adv_list(const ble_addr_t *peer_addr, uint8_t adv_sid)
```

Add peer device to periodic synchronization list.

Parameters

- **addr** – Peer address to add to list.
- **adv_sid** – Advertiser Set ID

Returns

0 on success; nonzero on failure.

```
int ble_gap_rem_dev_from_periodic_adv_list(const ble_addr_t *peer_addr, uint8_t adv_sid)
```

Remove peer device from periodic synchronization list.

Parameters

- **addr** – Peer address to remove from list.
- **adv_sid** – Advertiser Set ID

Returns

0 on success; nonzero on failure.

```
int ble_gap_clear_periodic_adv_list(void)
```

Clear periodic synchronization list.

Returns

0 on success; nonzero on failure.

```
int ble_gap_read_periodic_adv_list_size(uint8_t *per_adv_list_size)
```

Get periodic synchronization list size.

Parameters

- **per_adv_list_size** – On success list size is stored here.

Returns

0 on success; nonzero on failure.

```
int ble_gap_disc(uint8_t own_addr_type, int32_t duration_ms, const struct ble_gap_disc_params *disc_params,  
                 ble_gap_event_fn *cb, void *cb_arg)
```

Performs the Limited or General Discovery Procedures.

Parameters

- **own_addr_type** – The type of address the stack should use for itself when sending scan requests. Valid values are:
 - BLE_ADDR_TYPE_PUBLIC
 - BLE_ADDR_TYPE_RANDOM
 - BLE_ADDR_TYPE_RPA_PUB_DEFAULT

- BLE_ADDR_TYPE_RPA_RND_DEFAULT This parameter is ignored unless active scanning is being used.
- **duration_ms** – The duration of the discovery procedure. On expiration, the procedure ends and a BLE_GAP_EVENT_DISC_COMPLETE event is reported. Units are milliseconds. Specify BLE_HS_FOREVER for no expiration. Specify 0 to use stack defaults.
- **disc_params** – Additional arguments specifying the particulars of the discovery procedure.
- **cb** – The callback to associate with this discovery procedure. Advertising reports and discovery termination events are reported through this callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gap_ext_disc(uint8_t own_addr_type, uint16_t duration, uint16_t period, uint8_t filter_duplicates, uint8_t
                      filter_policy, uint8_t limited, const struct ble_gap_ext_disc_params *uncoded_params,
                      const struct ble_gap_ext_disc_params *coded_params, ble_gap_event_fn *cb, void
                      *cb_arg)
```

Performs the Limited or General Extended Discovery Procedures.

Parameters

- **own_addr_type** – The type of address the stack should use for itself when sending scan requests. Valid values are:
 - BLE_ADDR_TYPE_PUBLIC
 - BLE_ADDR_TYPE_RANDOM
 - BLE_ADDR_TYPE_RPA_PUB_DEFAULT
 - BLE_ADDR_TYPE_RPA_RND_DEFAULT This parameter is ignored unless active scanning is being used.
- **duration** – The duration of the discovery procedure. On expiration, if period is set to 0, the procedure ends and a BLE_GAP_EVENT_DISC_COMPLETE event is reported. Units are 10 milliseconds. Specify 0 for no expiration.
- **period** – Time interval from when the Controller started its last Scan Duration until it begins the subsequent Scan Duration. Specify 0 to scan continuously. Units are 1.28 second.
- **filter_duplicates** – Set to enable packet filtering in the controller
- **filter_policy** – Set the used filter policy. Valid values are:
 - BLE_HCI_SCAN_FILT_NO_WL
 - BLE_HCI_SCAN_FILT_USE_WL
 - BLE_HCI_SCAN_FILT_NO_WL_INITA
 - BLE_HCI_SCAN_FILT_USE_WL_INITA
 - BLE_HCI_SCAN_FILT_MAX This parameter is ignored unless **filter_duplicates** is set.
- **limited** – If limited discovery procedure should be used.
- **uncoded_params** – Additional arguments specifying the particulars of the discovery procedure for uncoded PHY. If NULL is provided no scan is performed for this PHY.
- **coded_params** – Additional arguments specifying the particulars of the discovery procedure for coded PHY. If NULL is provided no scan is performed for this PHY.

- **cb** – The callback to associate with this discovery procedure. Advertising reports and discovery termination events are reported through this callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gap_disc_cancel(void)
```

Cancels the discovery procedure currently in progress.

A success return code indicates that scanning has been fully aborted; a new discovery or connect procedure can be initiated immediately.

Returns

0 on success; BLE_HS_EALREADY if there is no discovery procedure to cancel; Other nonzero on unexpected error.

```
int ble_gap_disc_active(void)
```

Indicates whether a discovery procedure is currently in progress.

Returns

0: No discovery procedure in progress; 1: Discovery procedure in progress.

```
int ble_gap_connect(uint8_t own_addr_type, const ble_addr_t *peer_addr, int32_t duration_ms, const struct  
                      ble_gap_conn_params *params, ble_gap_event_fn *cb, void *cb_arg)
```

Initiates a connect procedure.

Parameters

- **own_addr_type** – The type of address the stack should use for itself during connection establishment.
 - BLE_OWN_ADDR_PUBLIC
 - BLE_OWN_ADDR_RANDOM
 - BLE_OWN_ADDR_RPA_PUBLIC_DEFAULT
 - BLE_OWN_ADDR_RPA_RANDOM_DEFAULT
- **peer_addr** – The address of the peer to connect to. If this parameter is NULL, the white list is used.
- **duration_ms** – The duration of the discovery procedure. On expiration, the procedure ends and a BLE_GAP_EVENT_DISC_COMPLETE event is reported. Units are milliseconds.
- **params** – Additional arguments specifying the particulars of the connect procedure. Specify null for default values.
- **cb** – The callback to associate with this connect procedure. When the connect procedure completes, the result is reported through this callback. If the connect procedure succeeds, the connection inherits this callback as its event-reporting mechanism.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; BLE_HS_EALREADY if a connection attempt is already in progress; BLE_HS_EBUSY if initiating a connection is not possible because scanning is in progress; BLE_HS_EDONE if the specified peer is already connected; Other nonzero on error.

```
int ble_gap_ext_connect(uint8_t own_addr_type, const ble_addr_t *peer_addr, int32_t duration_ms, uint8_t
    phy_mask, const struct ble_gap_conn_params *phy_1m_conn_params, const struct
    ble_gap_conn_params *phy_2m_conn_params, const struct ble_gap_conn_params
    *phy_coded_conn_params, ble_gap_event_fn *cb, void *cb_arg)
```

Initiates an extended connect procedure.

Parameters

- **own_addr_type** – The type of address the stack should use for itself during connection establishment.
 - BLE_OWN_ADDR_PUBLIC
 - BLE_OWN_ADDR_RANDOM
 - BLE_OWN_ADDR_RPA_PUBLIC_DEFAULT
 - BLE_OWN_ADDR_RPA_RANDOM_DEFAULT
- **peer_addr** – The address of the peer to connect to. If this parameter is NULL, the white list is used.
- **duration_ms** – The duration of the discovery procedure. On expiration, the procedure ends and a BLE_GAP_EVENT_DISC_COMPLETE event is reported. Units are milliseconds.
- **phy_mask** – Define on which PHYs connection attempt should be done
- **phy_1m_conn_params** – Additional arguments specifying the particulars of the connect procedure. When BLE_GAP_LE_PHY_1M_MASK is set in phy_mask this parameter can be specify to null for default values.
- **phy_2m_conn_params** – Additional arguments specifying the particulars of the connect procedure. When BLE_GAP_LE_PHY_2M_MASK is set in phy_mask this parameter can be specify to null for default values.
- **phy_coded_conn_params** – Additional arguments specifying the particulars of the connect procedure. When BLE_GAP_LE_PHY_CODED_MASK is set in phy_mask this parameter can be specify to null for default values.
- **cb** – The callback to associate with this connect procedure. When the connect procedure completes, the result is reported through this callback. If the connect procedure succeeds, the connection inherits this callback as its event-reporting mechanism.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; BLE_HS_EALREADY if a connection attempt is already in progress; BLE_HS_EBUSY if initiating a connection is not possible because scanning is in progress; BLE_HS_EDONE if the specified peer is already connected; Other nonzero on error.

```
int ble_gap_conn_cancel(void)
```

Aborts a connect procedure in progress.

Returns

0 on success; BLE_HS_EALREADY if there is no active connect procedure. Other nonzero on error.

```
int ble_gap_conn_active(void)
```

Indicates whether a connect procedure is currently in progress.

Returns

0: No connect procedure in progress; 1: Connect procedure in progress.

```
int ble_gap_terminate(uint16_t conn_handle, uint8_t hci_reason)
```

Terminates an established connection.

Parameters

- **conn_handle** – The handle corresponding to the connection to terminate.
- **hci_reason** – The HCI error code to indicate as the reason for termination.

Returns

0 on success; BLE_HS_ENOTCONN if there is no connection with the specified handle; Other nonzero on failure.

```
int ble_gap_wl_set(const ble_addr_t *addrs, uint8_t white_list_count)
```

Overwrites the controller's white list with the specified contents.

Parameters

- **addrs** – The entries to write to the white list.
- **white_list_count** – The number of entries in the white list.

Returns

0 on success; nonzero on failure.

```
int ble_gap_wl_read_size(uint8_t *size)
```

Retrieves the size of whitelist supported by controller.

Parameters

- **size** – On success, this holds the size of controller accept list

Returns

0 on success; nonzero on failure

```
int ble_gap_update_params(uint16_t conn_handle, const struct ble_gap_upd_params *params)
```

Initiates a connection parameter update procedure.

Parameters

- **conn_handle** – The handle corresponding to the connection to update.
- **params** – The connection parameters to attempt to update to.

Returns

0 on success; BLE_HS_ENOTCONN if the there is no connection with the specified handle; BLE_HS_EALREADY if a connection update procedure for this connection is already in progress; BLE_HS_EINVAL if requested parameters are invalid; Other nonzero on error.

```
int ble_gap_set_data_len(uint16_t conn_handle, uint16_t tx_octets, uint16_t tx_time)
```

Configure LE Data Length in controller (OGF = 0x08, OCF = 0x0022).

Parameters

- **conn_handle** – Connection handle.
- **tx_octets** – The preferred value of payload octets that the Controller should use for a new connection (Range 0x001B-0x00FB).
- **tx_time** – The preferred maximum number of microseconds that the local Controller should use to transmit a single link layer packet (Range 0x0148-0x4290).

Returns

0 on success, other error code on failure.

```
int ble_gap_read_sugg_def_data_len(uint16_t *out_sugg_max_tx_octets, uint16_t *out_sugg_max_tx_time)
```

Read LE Suggested Default Data Length in controller (OGF = 0x08, OCF = 0x0024).

Parameters

- **out_sugg_max_tx_octets** – The Host’s suggested value for the Controller’s maximum transmitted number of payload octets in LL Data PDUs to be used for new connections. (Range 0x001B-0x00FB).
- **out_sugg_max_tx_time** – The Host’s suggested value for the Controller’s maximum packet transmission time for packets containing LL Data PDUs to be used for new connections. (Range 0x0148-0x4290).

Returns

0 on success, other error code on failure.

```
int ble_gap_write_sugg_def_data_len(uint16_t sugg_max_tx_octets, uint16_t sugg_max_tx_time)
```

Configure LE Suggested Default Data Length in controller (OGF = 0x08, OCF = 0x0024).

Parameters

- **sugg_max_tx_octets** – The Host’s suggested value for the Controller’s maximum transmitted number of payload octets in LL Data PDUs to be used for new connections. (Range 0x001B-0x00FB).
- **sugg_max_tx_time** – The Host’s suggested value for the Controller’s maximum packet transmission time for packets containing LL Data PDUs to be used for new connections. (Range 0x0148-0x4290).

Returns

0 on success, other error code on failure.

```
int ble_gap_security_initiate(uint16_t conn_handle)
```

Initiates the GAP security procedure.

Depending on connection role and stored security information this function will start appropriate security procedure (pairing or encryption).

Parameters

- **conn_handle** – The handle corresponding to the connection to secure.

Returns

0 on success; BLE_HS_ENOTCONN if the there is no connection with the specified handle; BLE_HS_EALREADY if an security procedure for this connection is already in progress; Other nonzero on error.

```
int ble_gap_pair_initiate(uint16_t conn_handle)
```

Initiates the GAP pairing procedure as a master.

This is for testing only and should not be used by application. Use [*ble_gap_security_initiate\(\)*](#) instead.

Parameters

- **conn_handle** – The handle corresponding to the connection to start pairing on.

Returns

0 on success; BLE_HS_ENOTCONN if the there is no connection with the specified handle; BLE_HS_EALREADY if an pairing procedure for this connection is already in progress; Other nonzero on error.

```
int ble_gap_encryption_initiate(uint16_t conn_handle, uint8_t key_size, const uint8_t *ltk, uint16_t ediv,
                                uint64_t rand_val, int auth)
```

Initiates the GAP encryption procedure as a master.

This is for testing only and should not be used by application. Use [*ble_gap_security_initiate\(\)*](#) instead.

Parameters

- **conn_handle** – The handle corresponding to the connection to start encryption.
- **key_size** – Encryption key size
- **ltk** – Long Term Key to be used for encryption.
- **ediv** – Encryption Diversifier for LTK
- **rand_val** – Random Value for EDIV and LTK
- **auth** – If LTK provided is authenticated.

Returns

0 on success; BLE_HS_ENOTCONN if the there is no connection with the specified handle;
BLE_HS_EALREADY if an encryption procedure for this connection is already in progress;
Other nonzero on error.

```
int ble_gap_conn_rssi(uint16_t conn_handle, int8_t *out_rssi)
```

Retrieves the most-recently measured RSSI for the specified connection.

A connection's RSSI is updated whenever a data channel PDU is received.

Parameters

- **conn_handle** – Specifies the connection to query.
- **out_rssi** – On success, the retrieved RSSI is written here.

Returns

0 on success; A BLE host HCI return code if the controller rejected the request; A BLE host core return code on unexpected error.

```
int ble_gap_unpair(const ble_addr_t *peer_addr)
```

Unpairs a device with the specified address.

The keys related to that peer device are removed from storage and peer address is removed from the resolve list from the controller. If a peer is connected, the connection is terminated.

Parameters

- **peer_addr** – Address of the device to be unpaired

Returns

0 on success; A BLE host HCI return code if the controller rejected the request; A BLE host core return code on unexpected error.

```
int ble_gap_unpair_oldest_peer(void)
```

Unpairs the oldest bonded peer device.

The keys related to that peer device are removed from storage and peer address is removed from the resolve list from the controller. If a peer is connected, the connection is terminated.

Returns

0 on success; A BLE host HCI return code if the controller rejected the request; A BLE host core return code on unexpected error.

```
int ble_gap_unpair_oldest_except(const ble_addr_t *peer_addr)
```

Similar to [ble_gap_unpair_oldest_peer\(\)](#), except it makes sure that the peer received in input parameters is not deleted.

Parameters

- **peer_addr** – Address of the peer (not to be deleted)

Returns

0 on success; A BLE host HCI return code if the controller rejected the request; A BLE host core return code on unexpected error.

```
int ble_gap_set_priv_mode(const ble_addr_t *peer_addr, uint8_t priv_mode)
```

Set privacy mode for specified peer device.

Parameters

- **peer_addr** – Peer device address
- **priv_mode** – Privacy mode to be used. Can be one of following constants:
 - BLE_GAP_PRIVATE_MODE_NETWORK
 - BLE_GAP_PRIVATE_MODE_DEVICE

Returns

0 on success; nonzero on failure.

```
int ble_gap_read_le_phy(uint16_t conn_handle, uint8_t *tx_phy, uint8_t *rx_phy)
```

Read PHYs used for specified connection.

On success output parameters are filled with information about used PHY type.

Parameters

- **conn_handle** – Connection handle
- **tx_phy** – TX PHY used. Can be one of following constants:
 - BLE_GAP_LE_PHY_1M
 - BLE_GAP_LE_PHY_2M
 - BLE_GAP_LE_PHY_CODED
- **rx_phy** – RX PHY used. Can be one of following constants:
 - BLE_GAP_LE_PHY_1M
 - BLE_GAP_LE_PHY_2M
 - BLE_GAP_LE_PHY_CODED

Returns

0 on success; nonzero on failure.

```
int ble_gap_set_preferred_default_le_phy(uint8_t tx_phys_mask, uint8_t rx_phys_mask)
```

Set preferred default PHYs to be used for connections.

Parameters

- **tx_phys_mask** – Preferred TX PHY. Can be mask of following constants:
 - BLE_GAP_LE_PHY_1M_MASK
 - BLE_GAP_LE_PHY_2M_MASK

- BLE_GAP_LE_PHY_CODED_MASK
- BLE_GAP_LE_PHY_ANY_MASK
- **rx_phys_mask** – Preferred RX PHY. Can be mask of following constants:
 - BLE_GAP_LE_PHY_1M_MASK
 - BLE_GAP_LE_PHY_2M_MASK
 - BLE_GAP_LE_PHY_CODED_MASK
 - BLE_GAP_LE_PHY_ANY_MASK

Returns

0 on success; nonzero on failure.

```
int ble_gap_set_preferred_le_phy(uint16_t conn_handle, uint8_t tx_phys_mask, uint8_t rx_phys_mask,  
                                uint16_t phy_opts)
```

Set preferred PHYs to be used for connection.

Parameters

- **conn_handle** – Connection handle
- **tx_phys_mask** – Preferred TX PHY. Can be mask of following constants:
 - BLE_GAP_LE_PHY_1M_MASK
 - BLE_GAP_LE_PHY_2M_MASK
 - BLE_GAP_LE_PHY_CODED_MASK
 - BLE_GAP_LE_PHY_ANY_MASK
- **rx_phys_mask** – Preferred RX PHY. Can be mask of following constants:
 - BLE_GAP_LE_PHY_1M_MASK
 - BLE_GAP_LE_PHY_2M_MASK
 - BLE_GAP_LE_PHY_CODED_MASK
 - BLE_GAP_LE_PHY_ANY_MASK
- **phy_opts** – Additional PHY options. Valid values are:
 - BLE_GAP_LE_PHY_CODED_ANY
 - BLE_GAP_LE_PHY_CODED_S2
 - BLE_GAP_LE_PHY_CODED_S8

Returns

0 on success; nonzero on failure.

```
int ble_gap_set_default_subrate(uint16_t subrate_min, uint16_t subrate_max, uint16_t max_latency, uint16_t  
                                cont_num, uint16_t supervision_timeout)
```

Set default subrate.

Parameters

- **subrate_min** – Min subrate factor allowed in request by a peripheral
- **subrate_max** – Max subrate factor allowed in request by a peripheral
- **max_latency** – Max peripheral latency allowed in units of subrate conn interval.

- **cont_num** – Min number of underlying conn event to remain active after a packet containing PDU with non-zero length field is sent or received in request by a peripheral.
- **supervision_timeout** – Max supervision timeout allowed in request by a peripheral

```
int ble_gap_subrate_req(uint16_t conn_handle, uint16_t subrate_min, uint16_t subrate_max, uint16_t
max_latency, uint16_t cont_num, uint16_t supervision_timeout)
```

Subrate Request.

Parameters

- **conn_handle** – Connection Handle of the ACL.
- **subrate_min** – Min subrate factor to be applied
- **subrate_max** – Max subrate factor to be applied
- **max_latency** – Max peripheral latency allowed in units of subrated conn interval.
- **cont_num** – Min number of underlying conn event to remain active after a packet containing PDU with non-zero length field is sent or received in request by a peripheral.
- **supervision_timeout** – Max supervision timeout allowed for this connection

```
int ble_gap_event_listener_register(struct ble_gap_event_listener *listener, ble_gap_event_fn *fn, void
*arg)
```

Registers listener for GAP events.

On success listener structure will be initialized automatically and does not need to be initialized prior to calling this function. To change callback and/or argument unregister listener first and register it again.

Parameters

- **listener** – Listener structure
- **fn** – Callback function
- **arg** – Callback argument

Returns

0 on success BLE_HS_EINVAL if no callback is specified BLE_HS_EALREADY if listener is already registered

```
int ble_gap_event_listener_unregister(struct ble_gap_event_listener *listener)
```

Unregisters listener for GAP events.

Parameters

- **listener** – Listener structure

Returns

0 on success BLE_HS_ENOENT if listener was not registered

```
void ble_gap_conn_FOREACH_HANDLE(ble_gap_conn_FOREACH_HANDLE_fn *cb, void *arg)
```

Calls function defined by the user for every connection that is currently established.

Parameters

- **cb** – Callback function
- **arg** – Callback argument

```
int ble_gap_conn_find_handle_by_addr(const ble_addr_t *addr, uint16_t *out_conn_handle)
```

Looks for the connection with specified address.

Parameters

- **addr** – Address to look for
- **out_conn_handle** – Pointer to the variable in which conn_handle will be saved if found

Returns

0 on success BLE_HS_ENOTCONN if connection with specified address was not found

```
int ble_gap_set_path_loss_reporting_enable(uint16_t conn_handle, uint8_t enable)
```

Enable Set Path Loss Reporting.

Parameters

- **conn_handle** – Connection handle @params enable 1: Enable 0: Disable

Returns

0 on success; nonzero on failure.

```
int ble_gap_set_transmit_power_reporting_enable(uint16_t conn_handle, uint8_t local_enable, uint8_t  
                                              remote_enable)
```

Enable Reporting of Transmit Power.

@params remote_enable 1: Enable remote transmit power reports 0: Disable remote transmit power reports

Parameters

- **conn_handle** – Connection handle @params local_enable 1: Enable local transmit power reports 0: Disable local transmit power reports

Returns

0 on success; nonzero on failure.

```
int ble_gap_enh_read_transmit_power_level(uint16_t conn_handle, uint8_t phy, uint8_t *out_status, uint8_t  
                                         *out_phy, uint8_t *out_curr_tx_power_level, uint8_t  
                                         *out_max_tx_power_level)
```

LE Enhanced Read Transmit Power Level.

@params status 0 on success; nonzero on failure. @params conn_handle Connection handle @params phy Advertising Phy

@params curr_tx_power_level Current trasnmit Power Level

@params mx_tx_power_level Maximum transmit power level

Parameters

- **conn_handle** – Connection handle @params phy Advertising Phy

Returns

0 on success; nonzero on failure.

```
int ble_gap_read_remote_transmit_power_level(uint16_t conn_handle, uint8_t phy)
```

Read Remote Transmit Power Level.

Parameters

- **conn_handle** – Connection handle @params phy Advertising Phy

Returns

0 on success; nonzero on failure.

```
int ble_gap_set_path_loss_reporting_param(uint16_t conn_handle, uint8_t high_threshold, uint8_t
                                         high_hysteresis, uint8_t low_threshold, uint8_t low_hysteresis,
                                         uint16_t min_time_spent)
```

Set Path Loss Reprtng Param.

Parameters

- **conn_handle** – Connection handle @params high_threshold High Threshold value for path loss @params high_hysteresis Hysteresis value for high threshold @params low_threshold Low Threshold value for path loss @params low_hysteresis Hysteresis value for low threshold @params min_time_spent Minimum time controller observes the path loss

Returns

0 on success; nonzero on failure.

BLE_GAP_ADV_DFLT_CHANNEL_MAP

Default channels mask: all three channels are used.

BLE_GAP_EXT_ADV_DATA_STATUS_COMPLETE**BLE_GAP_EXT_ADV_DATA_STATUS_INCOMPLETE****BLE_GAP_EXT_ADV_DATA_STATUS_TRUNCATED****struct ble_gap_sec_state**

#include <ble_gap.h> Connection security state.

Public Members**unsigned encrypted**

If connection is encrypted.

unsigned authenticated

If connection is authenticated.

unsigned bonded

If connection is bonded (security information is stored)

unsigned key_size

Size of a key used for encryption.

struct ble_gap_adv_params

#include <ble_gap.h> Advertising parameters.

Public Members

`uint8_t conn_mode`

Advertising mode.

Can be one of following constants:

- `BLE_GAP_CONN_MODE_NON` (non-connectable; 3.C.9.3.2).
- `BLE_GAP_CONN_MODE_DIR` (directed-connectable; 3.C.9.3.3).
- `BLE_GAP_CONN_MODE_UND` (undirected-connectable; 3.C.9.3.4).

`uint8_t disc_mode`

Discoverable mode.

Can be one of following constants:

- `BLE_GAP_DISC_MODE_NON` (non-discoverable; 3.C.9.2.2).
- `BLE_GAP_DISC_MODE_LTD` (limited-discoverable; 3.C.9.2.3).
- `BLE_GAP_DISC_MODE_GEN` (general-discoverable; 3.C.9.2.4).

`uint16_t itvl_min`

Minimum advertising interval, if 0 stack use sane defaults.

`uint16_t itvl_max`

Maximum advertising interval, if 0 stack use sane defaults.

`uint8_t channel_map`

Advertising channel map , if 0 stack use sane defaults.

`uint8_t filter_policy`

Advertising Filter policy.

`uint8_t high_duty_cycle`

If do High Duty cycle for Directed Advertising.

`struct ble_gap_conn_desc`

`#include <ble_gap.h>` Connection descriptor.

Public Members

`struct ble_gap_sec_state sec_state`

Connection security state.

`ble_addr_t our_id_addr`

Local identity address.

```
ble_addr_t peer_id_addr
```

Peer identity address.

```
ble_addr_t our_ota_addr
```

Local over-the-air address.

```
ble_addr_t peer_ota_addr
```

Peer over-the-air address.

```
uint16_t conn_handle
```

Connection handle.

```
uint16_t conn_itvl
```

Connection interval.

```
uint16_t conn_latency
```

Connection latency.

```
uint16_t supervision_timeout
```

Connection supervision timeout.

```
uint8_t role
```

Connection Role Possible values BLE_GAP_ROLE_SLAVE or BLE_GAP_ROLE_MASTER.

```
uint8_t master_clock_accuracy
```

Master clock accuracy.

```
struct ble_gap_conn_params
```

```
#include <ble_gap.h> Connection parameters
```

Public Members

```
uint16_t scan_itvl
```

Scan interval in 0.625ms units.

```
uint16_t scan_window
```

Scan window in 0.625ms units.

```
uint16_t itvl_min
```

Minimum value for connection interval in 1.25ms units.

```
uint16_t itvl_max
```

Maximum value for connection interval in 1.25ms units.

`uint16_t latency`

Connection latency.

`uint16_t supervision_timeout`

Supervision timeout in 10ms units.

`uint16_t min_ce_len`

Minimum length of connection event in 0.625ms units.

`uint16_t max_ce_len`

Maximum length of connection event in 0.625ms units.

`struct ble_gap_ext_disc_params`

`#include <ble_gap.h>` Extended discovery parameters.

Public Members

`uint16_t itvl`

Scan interval in 0.625ms units.

`uint16_t window`

Scan window in 0.625ms units.

`uint8_t passive`

If passive scan should be used.

`struct ble_gap_disc_params`

`#include <ble_gap.h>` Discovery parameters.

Public Members

`uint16_t itvl`

Scan interval in 0.625ms units.

`uint16_t window`

Scan window in 0.625ms units.

`uint8_t filter_policy`

Scan filter policy.

`uint8_t limited`

If limited discovery procedure should be used.

```
uint8_t passive
```

If passive scan should be used.

```
uint8_t filter_duplicates
```

If enable duplicates filtering.

```
struct ble_gap_upd_params
```

#include <ble_gap.h> Connection parameters update parameters.

Public Members

```
uint16_t itvl_min
```

Minimum value for connection interval in 1.25ms units.

```
uint16_t itvl_max
```

Maximum value for connection interval in 1.25ms units.

```
uint16_t latency
```

Connection latency.

```
uint16_t supervision_timeout
```

Supervision timeout in 10ms units.

```
uint16_t min_ce_len
```

Minimum length of connection event in 0.625ms units.

```
uint16_t max_ce_len
```

Maximum length of connection event in 0.625ms units.

```
struct ble_gap_passkey_params
```

#include <ble_gap.h> Passkey query.

Public Members

```
uint8_t action
```

Passkey action, can be one of following constants:

- BLE_SM_IOACT_NONE
- BLE_SM_IOACT_OOB
- BLE_SM_IOACT_INPUT
- BLE_SM_IOACT_DISP
- BLE_SM_IOACT_NUMCMP

```
uint32_t numcmp  
Passkey to compare, valid for BLE_SM_IOACT_NUMCMP action.
```

```
struct ble_gap_ext_disc_desc  
#include <ble_gap.h> Extended advertising report.
```

Public Members

```
uint8_t props  
Report properties bitmask.
```

- BLE_HCI_ADV_CONN_MASK
- BLE_HCI_ADV_SCAN_MASK
- BLE_HCI_ADV_DIRECT_MASK
- BLE_HCI_ADV_SCAN_RSP_MASK
- BLE_HCI_ADV_LEGACY_MASK

```
uint8_t data_status  
Advertising data status, can be one of following constants:
```

- BLE_GAP_EXT_ADV_DATA_STATUS_COMPLETE
- BLE_GAP_EXT_ADV_DATA_STATUS_INCOMPLETE
- BLE_GAP_EXT_ADV_DATA_STATUS_TRUNCATED

```
uint8_t legacy_event_type
```

Legacy advertising PDU type.

Valid if BLE_HCI_ADV_LEGACY_MASK props is set. Can be one of following constants:

- BLE_HCI_ADV_RPT_EVTYPE_ADV_IND
- BLE_HCI_ADV_RPT_EVTYPE_DIR_IND
- BLE_HCI_ADV_RPT_EVTYPE_SCAN_IND
- BLE_HCI_ADV_RPT_EVTYPE_NONCONN_IND
- BLE_HCI_ADV_RPT_EVTYPE_SCAN_RSP

```
ble_addr_t addr
```

Advertiser address.

```
int8_t rssi  
Received signal strength indication in dBm (127 if unavailable)
```

int8_t tx_power

Advertiser transmit power in dBm (127 if unavailable)

uint8_t sid

Advertising Set ID.

uint8_t prim_phy

Primary advertising PHY, can be one of following constants:

- BLE_HCI_LE_PHY_1M
- BLE_HCI_LE_PHY_CODED

uint8_t sec_phy

Secondary advertising PHY, can be one of following constants:

- BLE_HCI_LE_PHY_1M
- LE_HCI_LE_PHY_2M
- BLE_HCI_LE_PHY_CODED

uint16_t periodic_adv_itvl

Periodic advertising interval.

0 if no periodic advertising.

uint8_t length_data

Advertising Data length.

const uint8_t *data

Advertising data.

ble_addr_t direct_addr

Directed advertising address.

Valid if BLE_HCI_ADV_DIRECT_MASK props is set (BLE_ADDR_ANY otherwise).

struct ble_gap_disc_desc

#include <ble_gap.h> Advertising report.

Public Members

uint8_t event_type

Advertising PDU type.

Can be one of following constants:

- BLE_HCI_ADV_RPT_EVTTYPE_ADV_IND
- BLE_HCI_ADV_RPT_EVTTYPE_DIR_IND
- BLE_HCI_ADV_RPT_EVTTYPE_SCAN_IND
- BLE_HCI_ADV_RPT_EVTTYPE_NONCONN_IND
- BLE_HCI_ADV_RPT_EVTTYPE_SCAN_RSP

uint8_t length_data

Advertising Data length.

ble_addr_t addr

Advertiser address.

int8_t rssi

Received signal strength indication in dBm (127 if unavailable)

const uint8_t *data

Advertising data.

ble_addr_t direct_addr

Directed advertising address.

Valid for BLE_HCI_ADV_RPT_EVTTYPE_DIR_IND event type (BLE_ADDR_ANY otherwise).

struct ble_gap_repeat_pairing

#include <ble_gap.h> Represents a repeat pairing operation between two devices.

This structure contains information about a repeat pairing operation between two devices. The host can use this information to determine whether it needs to initiate a pairing procedure with a remote device again.

Public Members

uint16_t conn_handle

The handle of the relevant connection.

uint8_t cur_key_size

Properties of the existing bond.

The size of the current encryption key in octets.

uint8_t cur_authenticated

A flag indicating whether the current connection is authenticated.

uint8_t cur_sc

A flag indicating whether the current connection is using secure connections.

uint8_t new_key_size

Properties of the imminent secure link if the pairing procedure is allowed to continue.

The size of the imminent encryption key in octets.

uint8_t new_authenticated

A flag indicating whether the imminent connection will be authenticated.

uint8_t new_sc

A flag indicating whether the imminent connection will use secure connections.

uint8_t new_bonding

A flag indicating whether the pairing procedure will result in a new bonding.,.

struct ble_gap_event

#include <ble_gap.h> Represents a GAP-related event.

When such an event occurs, the host notifies the application by passing an instance of this structure to an application-specified callback.

Public Members**uint8_t type**

Indicates the type of GAP event that occurred.

This is one of the BLE_GAP_EVENT codes.

union ble_gap_event

A discriminated union containing additional details concerning the GAP event.

The ‘type’ field indicates which member of the union is valid.

struct ble_gap_ext_adv_params

#include <ble_gap.h> Extended advertising parameters

Public Members

unsigned int **connectable**

If perform connectable advertising.

unsigned int **scannable**

If perform scannable advertising.

unsigned int **directed**

If perform directed advertising.

unsigned int **high_duty_directed**

If perform high-duty directed advertising.

unsigned int **legacy_pdu**

If use legacy PDUs for advertising.

Valid combinations of the connectable, scannable, directed, high_duty_directed options with the legacy_pdu flag are:

- IND -> legacy_pdu + connectable + scannable
- LD_DIR -> legacy_pdu + connectable + directed
- HD_DIR -> legacy_pdu + connectable + directed + high_duty_directed
- SCAN -> legacy_pdu + scannable
- NONCONN -> legacy_pdu

Any other combination of these options combined with the legacy_pdu flag are invalid.

unsigned int **anonymous**

If perform anonymous advertising.

unsigned int **include_tx_power**

If include TX power in advertising PDU.

unsigned int **scan_req_notif**

If enable scan request notification

uint32_t **itvl_min**

Minimum advertising interval in 0.625ms units, if 0 stack use sane defaults.

uint32_t **itvl_max**

Maximum advertising interval in 0.625ms units, if 0 stack use sane defaults.

uint8_t **channel_map**

Advertising channel map , if 0 stack use sane defaults.

```
uint8_t own_addr_type
```

Own address type to be used by advertising instance.

```
ble_addr_t peer
```

Peer address for directed advertising, valid only if directed is set.

```
uint8_t filter_policy
```

Advertising Filter policy.

```
uint8_t primary_phy
```

Primary advertising PHY to use , can be one of following constants:

- BLE_HCI_LE_PHY_1M
- BLE_HCI_LE_PHY_CODED

```
uint8_t secondary_phy
```

Secondary advertising PHY to use, can be one of following constants:

- BLE_HCI_LE_PHY_1M
- LE_HCI_LE_PHY_2M
- BLE_HCI_LE_PHY_CODED

```
int8_t tx_power
```

Preferred advertiser transmit power.

```
uint8_t sid
```

Advertising Set ID.

```
uint8_t primary_phy_opt
```

Primary PHY options.

```
uint8_t secondary_phy_opt
```

Secondary PHY options.

```
struct ble_gap_periodic_adv_params
```

```
#include <ble_gap.h> Periodic advertising parameters
```

Public Members

unsigned int **include_tx_power**

If include TX power in advertising PDU.

uint16_t **itvl_min**

Minimum advertising interval in 1.25ms units, if 0 stack use sane defaults.

uint16_t **itvl_max**

Maximum advertising interval in 1.25ms units, if 0 stack use sane defaults.

struct **ble_gap_periodic_adv_start_params**

#include <ble_gap.h> Periodic advertising enable parameters

struct **ble_gap_periodic_adv_sync_reporting_params**

#include <ble_gap.h> Periodic advertising sync reporting parameters

struct **ble_gap_periodic_adv_set_data_params**

#include <ble_gap.h> Periodic adv set data parameters

struct **ble_gap_periodic_sync_params**

#include <ble_gap.h> Periodic sync parameters

Public Members

uint16_t **skip**

The maximum number of periodic advertising events that controller can skip after a successful receive.

uint16_t **sync_timeout**

Synchronization timeout for the periodic advertising train in 10ms units.

unsigned int **reports_disabled**

If reports should be initially disabled when sync is created.

struct **ble_gap_event_listener**

#include <ble_gap.h> Event listener structure.

This should be used as an opaque structure and not modified manually.

Public Functions

`SLIST_ENTRY (ble_gap_event_listener) link`

Singly-linked list entry.

Public Members

`ble_gap_event_fn *fn`

The function to call when a GAP event occurs.

`void *arg`

An optional argument to pass to the event handler function.

7.7.4 NimBLE Host GATT Client Reference

Introduction

The Generic Attribute Profile (GATT) manages all activities involving services, characteristics, and descriptors. The client half of the GATT API initiates GATT procedures.

API

`typedef int ble_gatt_mtu_fn(uint16_t conn_handle, const struct ble_gatt_error *error, uint16_t mtu, void *arg)`

Function prototype for the GATT MTU exchange callback.

`typedef int ble_gatt_disc_svc_fn(uint16_t conn_handle, const struct ble_gatt_error *error, const struct ble_gatt_svc *service, void *arg)`

Function prototype for the GATT service discovery callback.

`typedef int ble_gatt_attr_fn(uint16_t conn_handle, const struct ble_gatt_error *error, struct ble_gatt_attr *attr, void *arg)`

The host will free the attribute mbuf automatically after the callback is executed.

The application can take ownership of the mbuf and prevent it from being freed by assigning NULL to attr->om.

`typedef int ble_gatt_attr_mult_fn(uint16_t conn_handle, const struct ble_gatt_error *error, struct ble_gatt_attr *attrs, uint8_t numAttrs, void *arg)`

The host will free the attribute mbuf automatically after the callback is executed.

The application can take ownership of the mbuf and prevent it from being freed by assigning NULL to attr->om.

`typedef int ble_gatt_reliable_attr_fn(uint16_t conn_handle, const struct ble_gatt_error *error, struct ble_gatt_attr *attrs, uint8_t numAttrs, void *arg)`

The host will free the attribute mbufs automatically after the callback is executed.

The application can take ownership of the mbufs and prevent them from being freed by assigning NULL to each attribute's om field.

`typedef int ble_gatt_chr_fn(uint16_t conn_handle, const struct ble_gatt_error *error, const struct ble_gatt_chr *chr, void *arg)`

Function prototype for the GATT characteristic callback.

```
typedef int ble_gatt_dsc_fn(uint16_t conn_handle, const struct ble_gatt_error *error, uint16_t chr_val_handle,  
                           const struct ble_gatt_dsc *dsc, void *arg)
```

Function prototype for the GATT descriptor callback.

```
typedef int ble_gatt_access_fn(uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt *ctxt,  
                               void *arg)
```

Type definition for GATT access callback function.

```
typedef uint16_t ble_gatt_chr_flags
```

Type definition for GATT characteristic flags.

```
typedef void ble_gatt_register_fn(struct ble_gatt_register_ctxt *ctxt, void *arg)
```

Type definition for GATT registration callback function.

```
typedef void (*ble_gatt_svc_foreach_fn)(const struct ble_gatt_svc_def *svc, uint16_t handle, uint16_t  
end_group_handle, void *arg)
```

Type definition for GATT service iteration callback function.

```
int ble_gattc_exchange_mtu(uint16_t conn_handle, ble_gatt_mtu_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Exchange MTU.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_all_svcs(uint16_t conn_handle, ble_gatt_disc_svc_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Discover All Primary Services.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_svc_by_uuid(uint16_t conn_handle, const ble_uuid_t *uuid, ble_gatt_disc_svc_fn *cb, void  
*cb_arg)
```

Initiates GATT procedure: Discover Primary Service by Service UUID.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **uuid** – The 128-bit UUID of the service to discover.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_find_inc_svcs(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle,
                            ble_gatt_disc_svc_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Find Included Services.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **start_handle** – The handle to begin the search at (generally the service definition handle).
- **end_handle** – The handle to end the search at (generally the last handle in the service).
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_all_chrs(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle,
                            ble_gatt_chr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Discover All Characteristics of a Service.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **start_handle** – The handle to begin the search at (generally the service definition handle).
- **end_handle** – The handle to end the search at (generally the last handle in the service).
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_chrs_by_uuid(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle, const
                                ble_uuid_t *uuid, ble_gatt_chr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Discover Characteristics by UUID.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **start_handle** – The handle to begin the search at (generally the service definition handle).
- **end_handle** – The handle to end the search at (generally the last handle in the service).
- **uuid** – The 128-bit UUID of the characteristic to discover.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_all_dscs(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle,
                            ble_gatt_dsc_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Discover All Characteristic Descriptors.

Parameters

- **conn_handle** – The connection over which to execute the procedure.

- **start_handle** – The handle of the characteristic value attribute.
- **end_handle** – The last handle in the characteristic definition.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read(uint16_t conn_handle, uint16_t attr_handle, ble_gatt_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Read Characteristic Value.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to read.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read_by_uuid(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle, const ble_uuid_t  
*uuid, ble_gatt_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Read Using Characteristic UUID.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **start_handle** – The first handle to search (generally the handle of the service definition).
- **end_handle** – The last handle to search (generally the last handle in the service definition).
- **uuid** – The 128-bit UUID of the characteristic to read.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read_long(uint16_t conn_handle, uint16_t handle, uint16_t offset, ble_gatt_attr_fn *cb, void  
*cb_arg)
```

Initiates GATT procedure: Read Long Characteristic Values.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **handle** – The handle of the characteristic value to read.
- **offset** – The offset within the characteristic value to start reading.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read_mult(uint16_t conn_handle, const uint16_t *handles, uint8_t num_handles, ble_gatt_attr_fn  
                        *cb, void *cb_arg)
```

Initiates GATT procedure: Read Multiple Characteristic Values.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **handles** – An array of 16-bit attribute handles to read.
- **num_handles** – The number of entries in the “handles” array.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read_mult_var(uint16_t conn_handle, const uint16_t *handles, uint8_t num_handles,  
                            ble_gatt_attr_mult_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Read Multiple Variable Length Characteristic Values.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **handles** – An array of 16-bit attribute handles to read.
- **num_handles** – The number of entries in the “handles” array.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_no_rsp(uint16_t conn_handle, uint16_t attr_handle, struct os_mbuf *om)
```

Initiates GATT procedure: Write Without Response.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to write to.
- **om** – The value to write to the characteristic.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_no_rsp_flat(uint16_t conn_handle, uint16_t attr_handle, const void *data, uint16_t  
                                data_len)
```

Initiates GATT procedure: Write Without Response.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to write to.
- **data** – The value to write to the characteristic.

- **data_len** – The number of bytes to write.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write(uint16_t conn_handle, uint16_t attr_handle, struct os_mbuf *om, ble_gatt_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Write Characteristic Value.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to write to.
- **om** – The value to write to the characteristic.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_flat(uint16_t conn_handle, uint16_t attr_handle, const void *data, uint16_t data_len, ble_gatt_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Write Characteristic Value (flat buffer version).

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to write to.
- **data** – The value to write to the characteristic.
- **data_len** – The number of bytes to write.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_long(uint16_t conn_handle, uint16_t attr_handle, uint16_t offset, struct os_mbuf *om, ble_gatt_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Write Long Characteristic Values.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to write to.
- **offset** – The offset at which to begin writing the value.
- **om** – The value to write to the characteristic.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_reliable(uint16_t conn_handle, struct ble_gatt_attr *attrs, int num_attrs,
                             ble_gatt_reliable_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Reliable Writes.

This function consumes the supplied mbus regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attrs** – An array of attribute descriptors; specifies which characteristics to write to and what data to write to them. The mbuf pointer in each attribute is set to NULL by this function.
- **numAttrs** – The number of characteristics to write; equal to the number of elements in the ‘attrs’ array.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_notify_custom(uint16_t conn_handle, uint16_t att_handle, struct os_mbuf *om)
```

Sends a “free-form” characteristic notification.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **att_handle** – The attribute handle to indicate in the outgoing notification.
- **om** – The value to write to the characteristic.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_notify_multiple_custom(uint16_t conn_handle, size_t chr_count, struct ble_gatt_notif *tuples)
```

Sends a “free-form” multiple handle variable length characteristic notification.

This function consumes supplied mbus regardless of the outcome. Notifications are sent in order of supplied entries. Function tries to send minimum amount of PDUs. If PDU can’t contain all of the characteristic values, multiple notifications are sent. If only one handle-value pair fits into PDU, or only one characteristic remains in the list, regular characteristic notification is sent.

If GATT client doesn’t support receiving multiple handle notifications, this will use GATT notification for each characteristic, separately.

If value of characteristic is not specified it will be read from local GATT database.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_count** – Number of characteristics to notify about.
- **tuples** – Handle-value pairs in form of *ble_gatt_notif* structures.

Returns

0 on success; nonzero on failure.

int **ble_gattc_notify_custom**(uint16_t conn_handle, uint16_t att_handle, struct *os_mbuf* *om)

Deprecated:

Should not be used. Use ble_gatts_notify_custom instead.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **att_handle** – The attribute handle to indicate in the outgoing notification.
- **om** – The value to write to the characteristic.

Returns

0 on success; nonzero on failure.

int **ble_gatts_notify**(uint16_t conn_handle, uint16_t chr_val_handle)

Sends a characteristic notification.

The content of the message is read from the specified characteristic.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing notification.

Returns

0 on success; nonzero on failure.

int **ble_gattc_notify**(uint16_t conn_handle, uint16_t chr_val_handle)

Deprecated:

Should not be used. Use ble_gatts_notify instead.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing notification.

Returns

0 on success; nonzero on failure.

int **ble_gatts_notify_multiple**(uint16_t conn_handle, size_t num_handles, const uint16_t *chr_val_handles)

Sends a multiple handle variable length characteristic notification.

The content of the message is read from the specified characteristics. Notifications are sent in order of supplied handles. Function tries to send minimum amount of PDUs. If PDU can't contain all of the characteristic values, multiple notifications are sent. If only one handle-value pair fits into PDU, or only one characteristic remains in the list, regular characteristic notification is sent.

If GATT client doesn't support receiving multiple handle notifications, this will use GATT notification for each characteristic, separately.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **num_handles** – The number of entries in the “chr_val_handles” array.

- **chr_val_handles** – Array of attribute handles of the characteristics to include in the outgoing notification.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_indicate_custom(uint16_t conn_handle, uint16_t chr_val_handle, struct os_mbuf *txom)
```

Sends a “free-form” characteristic indication.

The provided mbuf contains the indication payload. This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing indication.
- **txom** – The data to include in the indication.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_indicate_custom(uint16_t conn_handle, uint16_t chr_val_handle, struct os_mbuf *txom)
```

Deprecated:

Should not be used. Use ble_gatts_indicate_custom instead.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing indication.
- **txom** – The data to include in the indication.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_indicate(uint16_t conn_handle, uint16_t chr_val_handle)
```

Sends a characteristic indication.

The content of the message is read from the specified characteristic.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing indication.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_indicate(uint16_t conn_handle, uint16_t chr_val_handle)
```

Deprecated:

Should not be used. Use ble_gatts_indicate instead.

Sends a characteristic indication.

The content of the message is read from the specified characteristic.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing indication.

Returns

0 on success; nonzero on failure.

int ble_gattc_init(void)

Initialize the BLE GATT client.

Returns

0 on success; nonzero on failure.

int ble_gatts_add_svcs(const struct *ble_gatt_svc_def* *svcs)

Queues a set of service definitions for registration.

All services queued in this manner get registered when *ble_gatts_start()* is called.

Parameters

- **svcs** – An array of service definitions to queue for registration. This array must be terminated with an entry whose ‘type’ equals 0.

Returns

0 on success; BLE_HS_ENOMEM on heap exhaustion.

int ble_gatts_svc_set_visibility(uint16_t handle, int visible)

Set visibility of local GATT service.

Invisible services are not removed from database but are not discoverable by peer devices. Service Changed should be handled by application when needed by calling *ble_svc_gatt_changed()*.

Parameters

- **handle** – Handle of service
- **visible** – non-zero if service should be visible

Returns

0 on success; BLE_HS_ENOENT if service wasn’t found.

int ble_gatts_count_cfg(const struct *ble_gatt_svc_def* *defs)

Adjusts a host configuration object’s settings to accommodate the specified service definition array.

This function adds the counts to the appropriate fields in the supplied configuration object without clearing them first, so it can be called repeatedly with different inputs to calculate totals. Be sure to zero the GATT server settings prior to the first call to this function.

Parameters

- **defs** – The service array containing the resource definitions to be counted.

Returns

0 on success; BLE_HS_EINVAL if the svcs array contains an invalid resource definition.

void ble_gatts_chr_updated(uint16_t chr_val_handle)

Send notification (or indication) to any connected devices that have subscribed for notification (or indication) for specified characteristic.

Parameters

- **chr_val_handle** – Characteristic value handle

```
int ble_gatts_find_svc(const ble_uuid_t *uuid, uint16_t *out_handle)
```

Retrieves the attribute handle associated with a local GATT service.

Parameters

- **uuid** – The UUID of the service to look up.
- **out_handle** – On success, populated with the handle of the service attribute. Pass null if you don't need this value.

Returns

0 on success; BLE_HS_ENOENT if the specified service could not be found.

```
int ble_gatts_find_chr(const ble_uuid_t *svc_uuid, const ble_uuid_t *chr_uuid, uint16_t *out_def_handle,
                      uint16_t *out_val_handle)
```

Retrieves the pair of attribute handles associated with a local GATT characteristic.

Parameters

- **svc_uuid** – The UUID of the parent service.
- **chr_uuid** – The UUID of the characteristic to look up.
- **out_def_handle** – On success, populated with the handle of the characteristic definition attribute. Pass null if you don't need this value.
- **out_val_handle** – On success, populated with the handle of the characteristic value attribute. Pass null if you don't need this value.

Returns

0 on success; BLE_HS_ENOENT if the specified service or characteristic could not be found.

```
int ble_gatts_find_dsc(const ble_uuid_t *svc_uuid, const ble_uuid_t *chr_uuid, const ble_uuid_t *dsc_uuid,
                      uint16_t *out_dsc_handle)
```

Retrieves the attribute handle associated with a local GATT descriptor.

Parameters

- **svc_uuid** – The UUID of the grandparent service.
- **chr_uuid** – The UUID of the parent characteristic.
- **dsc_uuid** – The UUID of the descriptor to look up.
- **out_dsc_handle** – On success, populated with the handle of the descriptor attribute. Pass null if you don't need this value.

Returns

0 on success; BLE_HS_ENOENT if the specified service, characteristic, or descriptor could not be found.

```
void ble_gatts_show_local(void)
```

Prints dump of local GATT database.

This is useful to log local state of database in human readable form.

```
int ble_gatts_reset(void)
```

Resets the GATT server to its initial state.

On success, this function removes all supported services, characteristics, and descriptors. This function requires that:

- o No peers are connected, and
- o No GAP operations are active (advertise, discover, or connect).

Returns

0 on success; BLE_HS_EBUSY if the GATT server could not be reset due to existing connections or active GAP procedures.

int ble_gatts_start(void)

Makes all registered services available to peers.

This function gets called automatically by the NimBLE host on startup; manual calls are only necessary for replacing the set of supported services with a new one. This function requires that:

- o No peers are connected,
- and o No GAP operations are active (advertise, discover, or connect).

Returns

0 on success; A BLE host core return code on unexpected error.

int ble_gatts_peer_cl_sup_feat_get(uint16_t conn_handle, uint8_t *out_supported_feat, uint8_t len)

Gets Client Supported Features for specified connection.

Parameters

- **conn_handle** – Connection handle identifying the connection for which Client Supported Features should be saved
- **out_supported_feat** – Client supported features to be returned.
- **len** – The size of the Client Supported Features characteristic in octets.

Returns

0 on success; BLE_HS_ENOTCONN if no matching connection was found BLE_HS_EINVAL if supplied buffer is empty or if any Client Supported Feature was attempted to be disabled. A BLE host core return code on unexpected error.

struct ble_gatt_error

#include <ble_gatt.h> Represents a GATT error.

Public Members

uint16_t status

The GATT status code indicating the type of error.

uint16_t att_handle

The attribute handle associated with the error.

struct ble_gatt_svc

#include <ble_gatt.h> Represents a GATT Service.

Public Members

uint16_t start_handle

The start handle of the GATT service.

uint16_t end_handle

The end handle of the GATT service.

ble_uuid_any_t uuid

The UUID of the GATT service.

struct ble_gatt_attr

#include <ble_gatt.h> Represents a GATT attribute.

Public Members

uint16_t handle

The handle of the GATT attribute.

uint16_t offset

The offset of the data within the attribute.

struct *os_mbuf* *om

Pointer to the data buffer represented by an *os_mbuf*.

struct ble_gatt_chr

#include <ble_gatt.h> Represents a GATT characteristic.

Public Members

uint16_t def_handle

The handle of the GATT characteristic definition.

uint16_t val_handle

The handle of the GATT characteristic value.

uint8_t properties

The properties of the GATT characteristic.

ble_uuid_any_t uuid

The UUID of the GATT characteristic.

struct ble_gatt_dsc

#include <ble_gatt.h> Represents a GATT descriptor.

Public Members

`uint16_t handle`

The handle of the GATT descriptor.

`ble_uuid_any_t uuid`

The UUID of the GATT descriptor.

`struct ble_gatt_notif`

`#include <ble_gatt.h>` Represents a handle-value tuple for multiple handle notifications.

Public Members

`uint16_t handle`

The handle of the GATT characteristic.

`struct os_mbuf *value`

The buffer with GATT characteristic value.

`struct ble_gatt_chr_def`

`#include <ble_gatt.h>` Represents the definition of a GATT characteristic.

Public Members

`const ble_uuid_t *uuid`

Pointer to characteristic UUID; use BLE_UUIDxx_DECLARE macros to declare proper UUID; NULL if there are no more characteristics in the service.

`ble_gatt_access_fn *access_cb`

Callback that gets executed when this characteristic is read or written.

`void *arg`

Optional argument for callback.

`struct ble_gatt_dsc_def *descriptors`

Array of this characteristic's descriptors.

NULL if no descriptors. Do not include CCCD; it gets added automatically if this characteristic's notify or indicate flag is set.

`ble_gatt_chr_flags flags`

Specifies the set of permitted operations for this characteristic.

`uint8_t min_key_size`

Specifies minimum required key size to access this characteristic.

```
uint16_t *val_handle
```

At registration time, this is filled in with the characteristic's value attribute handle.

```
struct ble_gatt_svc_def
```

#include <ble_gatt.h> Represents the definition of a GATT service.

Public Members

```
uint8_t type
```

One of the following:
o BLE_GATT_SVC_TYPE_PRIMARY - primary service
o BLE_GATT_SVC_TYPE_SECONDARY - secondary service
o 0 - No more services in this array.

```
const ble_uuid_t *uuid
```

Pointer to service UUID; use BLE_UUIDxx_DECLARE macros to declare proper UUID; NULL if there are no more characteristics in the service.

```
const struct ble_gatt_svc_def **includes
```

Array of pointers to other service definitions.

These services are reported as “included services” during service discovery. Terminate the array with NULL.

```
const struct ble_gatt_chr_def *characteristics
```

Array of characteristic definitions corresponding to characteristics belonging to this service.

```
struct ble_gatt_dsc_def
```

#include <ble_gatt.h> Represents the definition of a GATT descriptor.

Public Members

```
const ble_uuid_t *uuid
```

Pointer to descriptor UUID; use BLE_UUIDxx_DECLARE macros to declare proper UUID; NULL if there are no more characteristics in the service.

```
uint8_t att_flags
```

Specifies the set of permitted operations for this descriptor.

```
uint8_t min_key_size
```

Specifies minimum required key size to access this descriptor.

```
ble_gatt_access_fn *access_cb
```

Callback that gets executed when the descriptor is read or written.

```
void *arg
```

Optional argument for callback.

struct ble_gatt_access_ctxt

#include <ble_gatt.h> Context for an access to a GATT characteristic or descriptor.

When a client reads or writes a locally registered characteristic or descriptor, an instance of this struct gets passed to the application callback.

Public Members

uint8_t op

Indicates the gatt operation being performed.

This is equal to one of the following values:

- o BLE_GATT_ACCESS_OP_READ_CHR
- o BLE_GATT_ACCESS_OP_WRITE_CHR
- o BLE_GATT_ACCESS_OP_READ_DSC
- o BLE_GATT_ACCESS_OP_WRITE_DSC

struct os_mbuf *om

A container for the GATT access data.

o For reads: The application populates this with the value of the characteristic or descriptor being read.
o For writes: This is already populated with the value being written by the peer. If the application wishes to retain this mbuf for later use, the access callback must set this pointer to NULL to prevent the stack from freeing it.

union ble_gatt_access_ctxt

The GATT operation being performed dictates which field in this union is valid.

If a characteristic is being accessed, the chr field is valid. Otherwise a descriptor is being accessed, in which case the dsc field is valid.

struct ble_gatt_register_ctxt

#include <ble_gatt.h> Context passed to the registration callback; represents the GATT service, characteristic, or descriptor being registered.

Public Members

uint8_t op

Indicates the gatt registration operation just performed.

This is equal to one of the following values:

- o BLE_GATT_REGISTER_OP_SVC
- o BLE_GATT_REGISTER_OP_CHR
- o BLE_GATT_REGISTER_OP_DSC

union ble_gatt_register_ctxt

The value of the op field determines which field in this union is valid.

7.7.5 NimBLE Host GATT Server Reference

Introduction

The Generic Attribute Profile (GATT) manages all activities involving services, characteristics, and descriptors. The server half of the GATT API handles registration and responding to GATT clients.

API

```
typedef int ble_gatt_mtu_fn(uint16_t conn_handle, const struct ble_gatt_error *error, uint16_t mtu, void *arg)
```

Function prototype for the GATT MTU exchange callback.

```
typedef int ble_gatt_disc_svc_fn(uint16_t conn_handle, const struct ble_gatt_error *error, const struct ble_gatt_svc *service, void *arg)
```

Function prototype for the GATT service discovery callback.

```
typedef int ble_gatt_attr_fn(uint16_t conn_handle, const struct ble_gatt_error *error, struct ble_gatt_attr *attr, void *arg)
```

The host will free the attribute mbuf automatically after the callback is executed.

The application can take ownership of the mbuf and prevent it from being freed by assigning NULL to attr->om.

```
typedef int ble_gatt_attr_mult_fn(uint16_t conn_handle, const struct ble_gatt_error *error, struct ble_gatt_attr *attrs, uint8_t numAttrs, void *arg)
```

The host will free the attribute mbuf automatically after the callback is executed.

The application can take ownership of the mbuf and prevent it from being freed by assigning NULL to attr->om.

```
typedef int ble_gatt_reliable_attr_fn(uint16_t conn_handle, const struct ble_gatt_error *error, struct ble_gatt_attr *attrs, uint8_t numAttrs, void *arg)
```

The host will free the attribute mbufs automatically after the callback is executed.

The application can take ownership of the mbufs and prevent them from being freed by assigning NULL to each attribute's om field.

```
typedef int ble_gatt_chr_fn(uint16_t conn_handle, const struct ble_gatt_error *error, const struct ble_gatt_chr *chr, void *arg)
```

Function prototype for the GATT characteristic callback.

```
typedef int ble_gatt_dsc_fn(uint16_t conn_handle, const struct ble_gatt_error *error, uint16_t chr_val_handle, const struct ble_gatt_dsc *dsc, void *arg)
```

Function prototype for the GATT descriptor callback.

```
typedef int ble_gatt_access_fn(uint16_t conn_handle, uint16_t attr_handle, struct ble_gatt_access_ctxt *ctxt, void *arg)
```

Type definition for GATT access callback function.

```
typedef uint16_t ble_gatt_chr_flags
```

Type definition for GATT characteristic flags.

```
typedef void ble_gatt_register_fn(struct ble_gatt_register_ctxt *ctxt, void *arg)
```

Type definition for GATT registration callback function.

```
typedef void (*ble_gatt_svc_foreach_fn)(const struct ble_gatt_svc_def *svc, uint16_t handle, uint16_t end_group_handle, void *arg)
```

Type definition for GATT service iteration callback function.

```
int ble_gattc_exchange_mtu(uint16_t conn_handle, ble_gatt_mtu_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Exchange MTU.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_all_svcs(uint16_t conn_handle, ble_gatt_disc_svc_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Discover All Primary Services.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_svc_by_uuid(uint16_t conn_handle, const ble_uuid_t *uuid, ble_gatt_disc_svc_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Discover Primary Service by Service UUID.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **uuid** – The 128-bit UUID of the service to discover.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_find_inc_svcs(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle, ble_gatt_disc_svc_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Find Included Services.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **start_handle** – The handle to begin the search at (generally the service definition handle).
- **end_handle** – The handle to end the search at (generally the last handle in the service).
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_all_chrs(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle,
                            ble_gatt_chr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Discover All Characteristics of a Service.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **start_handle** – The handle to begin the search at (generally the service definition handle).
- **end_handle** – The handle to end the search at (generally the last handle in the service).
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_chrs_by_uuid(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle, const
                                ble_uuid_t *uuid, ble_gatt_chr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Discover Characteristics by UUID.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **start_handle** – The handle to begin the search at (generally the service definition handle).
- **end_handle** – The handle to end the search at (generally the last handle in the service).
- **uuid** – The 128-bit UUID of the characteristic to discover.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_disc_all_dscs(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle,
                            ble_gatt_dsc_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Discover All Characteristic Descriptors.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **start_handle** – The handle of the characteristic value attribute.
- **end_handle** – The last handle in the characteristic definition.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read(uint16_t conn_handle, uint16_t attr_handle, ble_gatt_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Read Characteristic Value.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to read.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read_by_uuid(uint16_t conn_handle, uint16_t start_handle, uint16_t end_handle, const ble_uuid_t
                           *uuid, ble_gatt_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Read Using Characteristic UUID.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **start_handle** – The first handle to search (generally the handle of the service definition).
- **end_handle** – The last handle to search (generally the last handle in the service definition).
- **uuid** – The 128-bit UUID of the characteristic to read.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read_long(uint16_t conn_handle, uint16_t handle, uint16_t offset, ble_gatt_attr_fn *cb, void
                        *cb_arg)
```

Initiates GATT procedure: Read Long Characteristic Values.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **handle** – The handle of the characteristic value to read.
- **offset** – The offset within the characteristic value to start reading.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read_mult(uint16_t conn_handle, const uint16_t *handles, uint8_t num_handles, ble_gatt_attr_fn
                        *cb, void *cb_arg)
```

Initiates GATT procedure: Read Multiple Characteristic Values.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **handles** – An array of 16-bit attribute handles to read.
- **num_handles** – The number of entries in the “handles” array.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_read_mult_var(uint16_t conn_handle, const uint16_t *handles, uint8_t num_handles,
                            ble_gatt_attr_mult_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Read Multiple Variable Length Characteristic Values.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **handles** – An array of 16-bit attribute handles to read.
- **num_handles** – The number of entries in the “handles” array.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_no_rsp(uint16_t conn_handle, uint16_t attr_handle, struct os_mbuf *om)
```

Initiates GATT procedure: Write Without Response.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to write to.
- **om** – The value to write to the characteristic.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_no_rsp_flat(uint16_t conn_handle, uint16_t attr_handle, const void *data, uint16_t
                                 data_len)
```

Initiates GATT procedure: Write Without Response.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to write to.
- **data** – The value to write to the characteristic.
- **data_len** – The number of bytes to write.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write(uint16_t conn_handle, uint16_t attr_handle, struct os_mbuf *om, ble_gatt_attr_fn *cb, void
                     *cb_arg)
```

Initiates GATT procedure: Write Characteristic Value.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.

- **attr_handle** – The handle of the characteristic value to write to.
- **om** – The value to write to the characteristic.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_flat(uint16_t conn_handle, uint16_t attr_handle, const void *data, uint16_t data_len,  
                         ble_gatt_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Write Characteristic Value (flat buffer version).

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to write to.
- **data** – The value to write to the characteristic.
- **data_len** – The number of bytes to write.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_long(uint16_t conn_handle, uint16_t attr_handle, uint16_t offset, struct os_mbuf *om,  
                         ble_gatt_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Write Long Characteristic Values.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attr_handle** – The handle of the characteristic value to write to.
- **offset** – The offset at which to begin writing the value.
- **om** – The value to write to the characteristic.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_write_reliable(uint16_t conn_handle, struct ble_gatt_attr *attrs, int num_attrs,  
                            ble_gatt_reliable_attr_fn *cb, void *cb_arg)
```

Initiates GATT procedure: Reliable Writes.

This function consumes the supplied mbus regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **attrs** – An array of attribute descriptors; specifies which characteristics to write to and what data to write to them. The mbuf pointer in each attribute is set to NULL by this function.

- **num_attrs** – The number of characteristics to write; equal to the number of elements in the ‘attrs’ array.
- **cb** – The function to call to report procedure status updates; null for no callback.
- **cb_arg** – The optional argument to pass to the callback function.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_notify_custom(uint16_t conn_handle, uint16_t att_handle, struct os_mbuf *om)
```

Sends a “free-form” characteristic notification.

This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **att_handle** – The attribute handle to indicate in the outgoing notification.
- **om** – The value to write to the characteristic.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_notify_multiple_custom(uint16_t conn_handle, size_t chr_count, struct ble_gatt_notif *tuples)
```

Sends a “free-form” multiple handle variable length characteristic notification.

This function consumes supplied mbus regardless of the outcome. Notifications are sent in order of supplied entries. Function tries to send minimum amount of PDUs. If PDU can’t contain all of the characteristic values, multiple notifications are sent. If only one handle-value pair fits into PDU, or only one characteristic remains in the list, regular characteristic notification is sent.

If GATT client doesn’t support receiving multiple handle notifications, this will use GATT notification for each characteristic, separately.

If value of characteristic is not specified it will be read from local GATT database.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_count** – Number of characteristics to notify about.
- **tuples** – Handle-value pairs in form of *ble_gatt_notif* structures.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_notify_custom(uint16_t conn_handle, uint16_t att_handle, struct os_mbuf *om)
```

Deprecated:

Should not be used. Use *ble_gatts_notify_custom* instead.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **att_handle** – The attribute handle to indicate in the outgoing notification.
- **om** – The value to write to the characteristic.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_notify(uint16_t conn_handle, uint16_t chr_val_handle)
```

Sends a characteristic notification.

The content of the message is read from the specified characteristic.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing notification.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_notify(uint16_t conn_handle, uint16_t chr_val_handle)
```

Deprecated:

Should not be used. Use ble_gatts_notify instead.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing notification.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_notify_multiple(uint16_t conn_handle, size_t num_handles, const uint16_t *chr_val_handles)
```

Sends a multiple handle variable length characteristic notification.

The content of the message is read from the specified characteristics. Notifications are sent in order of supplied handles. Function tries to send minimum amount of PDUs. If PDU can't contain all of the characteristic values, multiple notifications are sent. If only one handle-value pair fits into PDU, or only one characteristic remains in the list, regular characteristic notification is sent.

If GATT client doesn't support receiving multiple handle notifications, this will use GATT notification for each characteristic, separately.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **num_handles** – The number of entries in the “chr_val_handles” array.
- **chr_val_handles** – Array of attribute handles of the characteristics to include in the outgoing notification.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_indicate_custom(uint16_t conn_handle, uint16_t chr_val_handle, struct os_mbuf *txom)
```

Sends a “free-form” characteristic indication.

The provided mbuf contains the indication payload. This function consumes the supplied mbuf regardless of the outcome.

Parameters

- **conn_handle** – The connection over which to execute the procedure.

- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing indication.
- **txom** – The data to include in the indication.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_indicate_custom(uint16_t conn_handle, uint16_t chr_val_handle, struct os_mbuf *txom)
```

Deprecated:

Should not be used. Use ble_gatts_indicate instead.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing indication.
- **txom** – The data to include in the indication.

Returns

0 on success; nonzero on failure.

```
int ble_gatts_indicate(uint16_t conn_handle, uint16_t chr_val_handle)
```

Sends a characteristic indication.

The content of the message is read from the specified characteristic.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing indication.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_indicate(uint16_t conn_handle, uint16_t chr_val_handle)
```

Deprecated:

Should not be used. Use ble_gatts_indicate instead.

Sends a characteristic indication.

The content of the message is read from the specified characteristic.

Parameters

- **conn_handle** – The connection over which to execute the procedure.
- **chr_val_handle** – The value attribute handle of the characteristic to include in the outgoing indication.

Returns

0 on success; nonzero on failure.

```
int ble_gattc_init(void)
```

Initialize the BLE GATT client.

Returns

0 on success; nonzero on failure.

int **ble_gatts_add_svcs**(const struct *ble_gatt_svc_def* *svcs)

Queues a set of service definitions for registration.

All services queued in this manner get registered when *ble_gatts_start()* is called.

Parameters

- **svcs** – An array of service definitions to queue for registration. This array must be terminated with an entry whose ‘type’ equals 0.

Returns

0 on success; BLE_HS_ENOMEM on heap exhaustion.

int **ble_gatts_svc_set_visibility**(uint16_t handle, int visible)

Set visibility of local GATT service.

Invisible services are not removed from database but are not discoverable by peer devices. Service Changed should be handled by application when needed by calling *ble_svc_gatt_changed()*.

Parameters

- **handle** – Handle of service
- **visible** – non-zero if service should be visible

Returns

0 on success; BLE_HS_ENOENT if service wasn’t found.

int **ble_gatts_count_cfg**(const struct *ble_gatt_svc_def* *defs)

Adjusts a host configuration object’s settings to accommodate the specified service definition array.

This function adds the counts to the appropriate fields in the supplied configuration object without clearing them first, so it can be called repeatedly with different inputs to calculate totals. Be sure to zero the GATT server settings prior to the first call to this function.

Parameters

- **defs** – The service array containing the resource definitions to be counted.

Returns

0 on success; BLE_HS_EINVAL if the svcs array contains an invalid resource definition.

void **ble_gatts_chr_updated**(uint16_t chr_val_handle)

Send notification (or indication) to any connected devices that have subscribed for notification (or indication) for specified characteristic.

Parameters

- **chr_val_handle** – Characteristic value handle

int **ble_gatts_find_svc**(const ble_uuid_t *uuid, uint16_t *out_handle)

Retrieves the attribute handle associated with a local GATT service.

Parameters

- **uuid** – The UUID of the service to look up.
- **out_handle** – On success, populated with the handle of the service attribute. Pass null if you don’t need this value.

Returns

0 on success; BLE_HS_ENOENT if the specified service could not be found.

```
int ble_gatts_find_chr(const ble_uuid_t *svc_uuid, const ble_uuid_t *chr_uuid, uint16_t *out_def_handle,
                      uint16_t *out_val_handle)
```

Retrieves the pair of attribute handles associated with a local GATT characteristic.

Parameters

- **svc_uuid** – The UUID of the parent service.
- **chr_uuid** – The UUID of the characteristic to look up.
- **out_def_handle** – On success, populated with the handle of the characteristic definition attribute. Pass null if you don't need this value.
- **out_val_handle** – On success, populated with the handle of the characteristic value attribute. Pass null if you don't need this value.

Returns

0 on success; BLE_HS_ENOENT if the specified service or characteristic could not be found.

```
int ble_gatts_find_dsc(const ble_uuid_t *svc_uuid, const ble_uuid_t *chr_uuid, const ble_uuid_t *dsc_uuid,
                      uint16_t *out_dsc_handle)
```

Retrieves the attribute handle associated with a local GATT descriptor.

Parameters

- **svc_uuid** – The UUID of the grandparent service.
- **chr_uuid** – The UUID of the parent characteristic.
- **dsc_uuid** – The UUID of the descriptor to look up.
- **out_dsc_handle** – On success, populated with the handle of the descriptor attribute. Pass null if you don't need this value.

Returns

0 on success; BLE_HS_ENOENT if the specified service, characteristic, or descriptor could not be found.

```
void ble_gatts_show_local(void)
```

Prints dump of local GATT database.

This is useful to log local state of database in human readable form.

```
int ble_gatts_reset(void)
```

Resets the GATT server to its initial state.

On success, this function removes all supported services, characteristics, and descriptors. This function requires that:

- o No peers are connected, and
- o No GAP operations are active (advertise, discover, or connect).

Returns

0 on success; BLE_HS_EBUSY if the GATT server could not be reset due to existing connections or active GAP procedures.

```
int ble_gatts_start(void)
```

Makes all registered services available to peers.

This function gets called automatically by the NimBLE host on startup; manual calls are only necessary for replacing the set of supported services with a new one. This function requires that:

- o No peers are connected, and
- o No GAP operations are active (advertise, discover, or connect).

Returns

0 on success; A BLE host core return code on unexpected error.

```
int ble_gatts_peer_cl_sup_feat_get(uint16_t conn_handle, uint8_t *out_supported_feat, uint8_t len)
```

Gets Client Supported Features for specified connection.

Parameters

- **conn_handle** – Connection handle identifying the connection for which Client Supported Features should be saved
- **out_supported_feat** – Client supported features to be returned.
- **len** – The size of the Client Supported Features characteristic in octets.

Returns

0 on success; BLE_HS_ENOTCONN if no matching connection was found BLE_HS_EINVAL if supplied buffer is empty or if any Client Supported Feature was attempted to be disabled. A BLE host core return code on unexpected error.

```
struct ble_gatt_error
```

#include <ble_gatt.h> Represents a GATT error.

Public Members

uint16_t **status**

The GATT status code indicating the type of error.

uint16_t **att_handle**

The attribute handle associated with the error.

```
struct ble_gatt_svc
```

#include <ble_gatt.h> Represents a GATT Service.

Public Members

uint16_t **start_handle**

The start handle of the GATT service.

uint16_t **end_handle**

The end handle of the GATT service.

ble_uuid_any_t **uuid**

The UUID of the GATT service.

```
struct ble_gatt_attr
```

#include <ble_gatt.h> Represents a GATT attribute.

Public Members**uint16_t handle**

The handle of the GATT attribute.

uint16_t offset

The offset of the data within the attribute.

struct *os_mbuf* *om

Pointer to the data buffer represented by an *os_mbuf*.

struct ble_gatt_chr

#include <ble_gatt.h> Represents a GATT characteristic.

Public Members**uint16_t def_handle**

The handle of the GATT characteristic definition.

uint16_t val_handle

The handle of the GATT characteristic value.

uint8_t properties

The properties of the GATT characteristic.

ble_uuid_any_t uuid

The UUID of the GATT characteristic.

struct ble_gatt_dsc

#include <ble_gatt.h> Represents a GATT descriptor.

Public Members**uint16_t handle**

The handle of the GATT descriptor.

ble_uuid_any_t uuid

The UUID of the GATT descriptor.

struct ble_gatt_notif

#include <ble_gatt.h> Represents a handle-value tuple for multiple handle notifications.

Public Members

`uint16_t handle`

The handle of the GATT characteristic.

`struct os_mbuf *value`

The buffer with GATT characteristic value.

`struct ble_gatt_chr_def`

`#include <ble_gatt.h>` Represents the definition of a GATT characteristic.

Public Members

`const ble_uuid_t *uuid`

Pointer to characteristic UUID; use `BLE_UUIDxx_DECLARE` macros to declare proper UUID; NULL if there are no more characteristics in the service.

`ble_gatt_access_fn *access_cb`

Callback that gets executed when this characteristic is read or written.

`void *arg`

Optional argument for callback.

`struct ble_gatt_dsc_def *descriptors`

Array of this characteristic's descriptors.

NULL if no descriptors. Do not include CCCD; it gets added automatically if this characteristic's notify or indicate flag is set.

`ble_gatt_chr_flags flags`

Specifies the set of permitted operations for this characteristic.

`uint8_t min_key_size`

Specifies minimum required key size to access this characteristic.

`uint16_t *val_handle`

At registration time, this is filled in with the characteristic's value attribute handle.

`struct ble_gatt_svc_def`

`#include <ble_gatt.h>` Represents the definition of a GATT service.

Public Members

uint8_t type

One of the following:
 o BLE_GATT_SVC_TYPE_PRIMARY - primary service
 o BLE_GATT_SVC_TYPE_SECONDARY - secondary service
 o 0 - No more services in this array.

const ble_uuid_t *uuid

Pointer to service UUID; use BLE_UUIDxx_DECLARE macros to declare proper UUID; NULL if there are no more characteristics in the service.

const struct *ble_gatt_svc_def* **includes

Array of pointers to other service definitions.

These services are reported as “included services” during service discovery. Terminate the array with NULL.

const struct *ble_gatt_chr_def* *characteristics

Array of characteristic definitions corresponding to characteristics belonging to this service.

struct *ble_gatt_dsc_def*

#include <*ble_gatt.h*> Represents the definition of a GATT descriptor.

Public Members

const ble_uuid_t *uuid

Pointer to descriptor UUID; use BLE_UUIDxx_DECLARE macros to declare proper UUID; NULL if there are no more characteristics in the service.

uint8_t att_flags

Specifies the set of permitted operations for this descriptor.

uint8_t min_key_size

Specifies minimum required key size to access this descriptor.

***ble_gatt_access_fn* *access_cb**

Callback that gets executed when the descriptor is read or written.

void *arg

Optional argument for callback.

struct *ble_gatt_access_ctxt*

#include <*ble_gatt.h*> Context for an access to a GATT characteristic or descriptor.

When a client reads or writes a locally registered characteristic or descriptor, an instance of this struct gets passed to the application callback.

Public Members

`uint8_t op`

Indicates the gatt operation being performed.

This is equal to one of the following values:

- o `BLE_GATT_ACCESS_OP_READ_CHR`
- o `BLE_GATT_ACCESS_OP_WRITE_CHR`
- o `BLE_GATT_ACCESS_OP_READ_DSC`
- o `BLE_GATT_ACCESS_OP_WRITE_DSC`

`struct os_mbuf *om`

A container for the GATT access data.

o For reads: The application populates this with the value of the characteristic or descriptor being read.
o For writes: This is already populated with the value being written by the peer. If the application wishes to retain this mbuf for later use, the access callback must set this pointer to NULL to prevent the stack from freeing it.

`union ble_gatt_access_ctxt`

The GATT operation being performed dictates which field in this union is valid.

If a characteristic is being accessed, the chr field is valid. Otherwise a descriptor is being accessed, in which case the dsc field is valid.

`struct ble_gatt_register_ctxt`

`#include <ble_gatt.h>` Context passed to the registration callback; represents the GATT service, characteristic, or descriptor being registered.

Public Members

`uint8_t op`

Indicates the gatt registration operation just performed.

This is equal to one of the following values:

- o `BLE_GATT_REGISTER_OP_SVC`
- o `BLE_GATT_REGISTER_OP_CHR`
- o `BLE_GATT_REGISTER_OP_DSC`

`union ble_gatt_register_ctxt`

The value of the op field determines which field in this union is valid.

7.7.6 NimBLE Host Identity Reference

Introduction

The identity API provides facilities for querying and configuring your device's addresses. BLE's addressing scheme is quite involved; the summary that follows is only a brief introduction.

BLE defines four address types:

Type	Description	Identity?	Configured with
Public	Address assigned by manufacturer; the three most significant bytes form the manufacturer's OUI.	Yes	N/A; read from controller at startup.
Static random	Randomly generated address.	Yes	<code>ble_hs_id_set_rnd()</code>
Resolvable private (RPA)	Address randomly generated from an identity address and an identity resolving key (IRK).	No	N/A; generated by controller periodically.
Non-resolvable private (NRPA)	Randomly generated address.	No	<code>ble_hs_id_set_rnd()</code>

Identity Addresses

The third column in the above table indicates the *identity* property of each address type. An identity address never changes, and a device can be identified by one of its unique identity addresses.

Non-identity addresses are used by devices supporting BLE privacy. A device using the privacy feature frequently changes its own address to a newly-generated non-identity address. By cycling its address, the device makes it impossible for eavesdroppers to track its location.

A device can have up to two identity addresses at once: one public and one static random. As indicated in the above table, the public identity address cannot be configured; the static random identity address can be set by calling `ble_hs_id_set_rnd()`.

The address type is selected on a per-GAP-procedure basis. Each time you initiate a GAP procedure, you indicate which address type the device should use for the duration of the procedure.

Header

```
#include "host/ble_hs.h"
```

API

`int ble_hs_id_gen_rnd(int nrpa, ble_addr_t *out_addr)`

Generates a new random address.

This function does not configure the device with the new address; the caller can use the address in subsequent operations.

Parameters

- **nrpa** – The type of random address to generate: 0: static 1: non-resolvable private
- **out_addr** – On success, the generated address gets written here.

Returns

0 on success; nonzero on failure.

`int ble_hs_id_set_rnd(const uint8_t *rnd_addr)`

Sets the device's random address.

The address type (static vs. non-resolvable private) is inferred from the most-significant byte of the address. The address is specified in host byte order (little-endian!).

Parameters

- **rnd_addr** – The random address to set.

Returns

0 on success; BLE_HS_EINVAL if the specified address is not a valid static random or non-resolvable private address. Other nonzero on error.

```
int ble_hs_id_copy_addr(uint8_t id_addr_type, uint8_t *out_id_addr, int *out_is_nrpa)
```

Retrieves one of the device's identity addresses.

The device can have two identity addresses: one public and one random. The id_addr_type argument specifies which of these two addresses to retrieve.

Parameters

- **id_addr_type** – The type of identity address to retrieve. Valid values are:
 - o BLE_ADDR_PUBLIC
 - o BLE_ADDR_RANDOM
- **out_id_addr** – On success, the requested identity address is copied into this buffer. The buffer must be at least six bytes in size. Pass NULL if you do not require this information.
- **out_is_nrpa** – On success, the pointed-to value indicates whether the retrieved address is a non-resolvable private address. Pass NULL if you do not require this information.

Returns

0 on success; BLE_HS_EINVAL if an invalid address type was specified; BLE_HS_ENOADDR if the device does not have an identity address of the requested type; Other BLE host core code on error.

```
int ble_hs_id_infer_auto(int privacy, uint8_t *out_addr_type)
```

Determines the best address type to use for automatic address type resolution.

Calculation of the best address type is done as follows:

if privacy requested: if we have a random static address: → RPA with static random ID else → RPA with public ID end else if we have a random static address: → random static address else → public address end end

Parameters

- **privacy** – (0/1) Whether to use a private address.
- **out_addr_type** – On success, the “own addr type” code gets written here.

Returns

0 if an address type was successfully inferred. BLE_HS_ENOADDR if the device does not have a suitable address. Other BLE host core code on error.

7.7.7 NimBLE Host ATT Client Reference

Introduction

The Attribute Protocol (ATT) is a mid-level protocol that all BLE devices use to exchange data. Data is exchanged when an ATT client reads or writes an attribute belonging to an ATT server. Any device that needs to send or receive data must support both the client and server functionality of the ATT protocol. The only devices which do not support ATT are the most basic ones: broadcasters and observers (i.e., beacons and listening devices).

Most ATT functionality is not interesting to an application. Rather than use ATT directly, an application uses the higher level GATT profile, which sits directly above ATT in the host. NimBLE exposes the few bits of ATT functionality which are not encompassed by higher level GATT functions. This section documents the ATT functionality that the NimBLE host exposes to the application.

API

int `ble_hs_is_enabled`(void)

Indicates whether the host is enabled.

The host is enabled if it is starting or fully started. It is disabled if it is stopping or stopped.

Returns

1 if the host is enabled; 0 if the host is disabled.

int `ble_hs_synced`(void)

Indicates whether the host has synchronized with the controller.

Synchronization must occur before any host procedures can be performed.

Returns

1 if the host and controller are in sync; 0 if the host and controller are out of sync.

int `ble_hs_start`(void)

Synchronizes the host with the controller by sending a sequence of HCI commands.

This function must be called before any other host functionality is used, but it must be called after both the host and controller are initialized. Typically, the host-parent-task calls this function at the top of its task routine. This function must only be called in the host parent task. A safe alternative for starting the stack from any task is to call [`ble_hs_sched_start\(\)`](#).

If the host fails to synchronize with the controller (if the controller is not fully booted, for example), the host will attempt to resynchronize every 100 ms. For this reason, an error return code is not necessarily fatal.

Returns

0 on success; nonzero on error.

void `ble_hs_sched_start`(void)

Enqueues a host start event to the default event queue.

The actual host startup is performed in the host parent task, but using the default queue here ensures the event won't run until the end of `main()` when this is called during system initialization. This allows the application to configure the host package in the meantime.

If auto-start is disabled, the application should use this function to start the BLE stack. This function can be called at any time as long as the host is stopped. When the host successfully starts, the application is notified via the `ble_hs_cfg.sync_cb` callback.

void `ble_hs_sched_reset`(int reason)

Causes the host to reset the NimBLE stack as soon as possible.

The application is notified when the reset occurs via the host reset callback.

Parameters

- **reason** – The host error code that gets passed to the reset callback.

void `ble_hs_evq_set`(struct ble_npl_eventq *evq)

Designates the specified event queue for NimBLE host work.

By default, the host uses the default event queue and runs in the main task. This function is useful if you want the host to run in a different task.

Parameters

- **evq** – The event queue to use for host work.

```
void ble_hs_init(void)
```

Initializes the NimBLE host.

This function must be called before the OS is started. The NimBLE stack requires an application task to function. One application task in particular is designated as the “host parent task”. In addition to application-specific work, the host parent task does work for NimBLE by processing events generated by the host.

```
int ble_hs_shutdown(int reason)
```

Called when the system is shutting down.

Stops the BLE host.

Parameters

- **reason** – The reason for the shutdown. One of the HAL_RESET_[...] codes or an implementation-defined value.

Returns

SYSDOWN_IN_PROGRESS.

BLE_HS_FOREVER

Represents an infinite value for timeouts or durations.

BLE_HS_CONN_HANDLE_NONE

Connection handle not present.

7.8 API for btshell app

7.8.1 Usage

“btshell” is one of the sample applications that come with Mynewt. It is a shell application which provides a basic interface to the host-side of the BLE stack. “btshell” includes all the possible roles (Central/Peripheral) and they may be run simultaneously. You can run btshell on a board and issue commands that make it behave as a central or a peripheral with different peers.

btshell is a new application that uses shell subsystem introduced in Mynewt 1.1 and has updated commands and parameters names. Thanks to support for tab completion commands names are more descriptive and self-explanatory without requiring extensive typing.

Highlighted below are some of the ways you can use the API to establish connections and discover services and characteristics from peer devices. For descriptions of the full API, go to the next sections on [GAP API for btshell](#) and [GATT feature API for btshell](#).

- *Set device address.*
- *Initiate a direct connection to a device*
- *Configure advertisements to include device name*
- *Begin sending undirected general advertisements*
- *Show established connections.*
- *Discover and display peer’s services, characteristics, and descriptors.*

- *Read an attribute belonging to the peer*
- *Write to an attribute belonging to the peer*
- *Perform a passive scan*

Set device address.

On startup, btshell has the following identity address configuration:

- Public address: None
- Random address: None

The below `set` commands can be used to change the address configuration:

```
set addr_type=public addr=<device-address>
set addr_type=random addr=<device-address>
```

For example:

```
set addr_type=public addr=01:02:03:04:05:06
set addr_type=random addr=c1:aa:bb:cc:dd:ee
```

The address configuration can be viewed with the `show-addr` command, as follows:

```
show-addr
public_id_addr=01:02:03:04:05:06 random_id_addr=c1:aa:bb:cc:dd:ee
```

Initiate a direct connection to a device

In this case, your board is acting as a central and initiating a connection with another BLE device. The example assumes you know the address of the peer, either by scanning for available peers or because you have set up the peer yourself.

```
connect peer_addr=d4:f5:13:53:d2:43
connection established; handle=1 our_ota_addr_type=0 our_ota_addr=0a:0b:0c:0d:0e:0f out_
id_addr_type=0 our_id_addr=0a:0b:0c:0d:0e:0f peer_addr_type=0 peer_
addr=43:d2:53:13:f5:d4 conn_itvl=40 conn_latency=0 supervision_timeout=256 encrypted=0
authenticated=0 bonded=0
```

The `handle=1` in the output indicates that it is connection-1.

Configure advertisements to include device name

In this case, your board is acting as a peripheral.

With Extended Advertising enabled (should be executed after `advertise-configure`):

```
advertise-set-adv-data name=<your-device-name>
```

With Extended Advertising disabled:

```
set-adv-data name=<your-device-name>
```

Begin sending undirected general advertisements

In this case, your board is acting as a peripheral.

With Extended Advertising enabled:

```
advertise-configure connectable=1 legacy=1 scannable=1  
advertise-start
```

With Extended Advertising disabled:

```
advertise conn=und discov=gen
```

Show established connections.

```
gatt-show-conn
```

Discover and display peer's services, characteristics, and descriptors.

This is how you discover and then display the services of the peer you established earlier across connection-1.

```
gatt-discover-full conn=1  
gatt-show  
[ts=132425ssb, mod=64 level=2] CONNECTION: handle=1 addr=d4:f5:13:53:d2:43  
[ts=132428ssb, mod=64 level=2] start=1 end=5 uuid=0x1800  
[ts=132433ssb, mod=64 level=2] start=6 end=16 uuid=0x1808  
[ts=132437ssb, mod=64 level=2] start=17 end=31 uuid=0x180a  
[ts=132441ssb, mod=64 level=2] start=32 end=65535 uuid=00000000-0000-1000-  
↪10000000000000000000
```

Read an attribute belonging to the peer

```
gatt-read conn=1 attr=21
```

Write to an attribute belonging to the peer

```
gatt-write conn=1 attr=3 value=0x01:0x02:0x03
```

Perform a passive scan

This is how you tell your board to listen to all advertisements around it. The duration is specified in ms.

```
scan duration=1000 passive=1 filter=no_wl
```

7.8.2 GAP API for btshell

Generic Access Profile (GAP) defines the generic procedures related to discovery of Bluetooth devices (idle mode procedures) and link management aspects of connecting to Bluetooth devices (connecting mode procedures). It also defines procedures related to use of different security levels.

Several different modes and procedures may be performed simultaneously over an LE physical transport. The following modes and procedures are defined for use over an LE physical transport:

1. Broadcast mode and observation procedure

- These allow two devices to communicate in a unidirectional connectionless manner using the advertising events.

2. Discovery modes and procedures

- All devices shall be in either non-discoverable mode or one of the discoverable modes.
- A device in the discoverable mode shall be in either the general discoverable mode or the limited discoverable mode.
- A device in non-discoverable mode will not be discovered by any device that is performing either the general discovery procedure or the limited discovery procedure.

3. Connection modes and procedures

- allow a device to establish a connection to another device.
- allow updating of parameters of the connection
- allow termination of the connection

4. Bonding modes and procedures

- Bonding allows two connected devices to exchange and store security and identity information to create a trusted relationship.
- Bonding can occur only between two devices in bondable mode.

Available commands

Configuration

Table 7: Configuration

Command	Parameters	Possible values	Description
set			Set configuration options
	addr	XX:XX:XX:XX:XX:XX	Local device address
	addr_type	public random	Local device address type Use random address for scan requests
	mtu	[23-UINT16_MAX]	GATT Maximum Transmission Unit (MTU)
	irk	XX:XX:XX...	Local Identity Resolving Key (16 byte)
set-priv-mode			Set privacy mode for device
	addr	XX:XX:XX:XX:XX:XX	Remote device address
	addr_type	public random	Remote device public address type Remote device random address type
	mode	[0-1]	0 - use network privacy, 1 - use device privacy
white-list			Add devices to white list (this command accepts multiple instances of addr and addr_type parameters)
	addr	XX:XX:XX:XX:XX:XX	Remote device address
	addr_type	public random	Remote device public address type Remote device random address type

Device discovery and connection

Table 8: Device discovery and connection

Command	Parameters	Possible values	Description
scan			Discover remote devices
	cancel		cancel ongoing scan procedure
	extended	none 1M coded both	Start legacy scan Start extended scan on 1M PHY Start extended scan on Coded PHY Start extended scan on both PHYs
	duration	[1-INT32_MAX],	Duration of scan in milliseconds
	limited	[0-1]	Use limited discovery procedure
	passive	[0-1]	Use passive scan
	interval	[0-UINT16_MAX]	Scan interval, if 0 use stack's default
	window	[0-UINT16_MAX]	Scan window, if 0 use stack's default
	filter	no_wl use_wl no_wl_inita use_wl_inita	Scan filter policy - Accept all advertising packets Accept only advertising packets from devices on White List Accept all advertising packets (including directed RPA) Accept only advertising packets from devices on White List (including directed RPA)
	nodups	[0-1]	Disable duplicates filtering
	own_addr_type	public	Use public address for scan requests

continues on next page

Table 8 – continued from previous page

Command	Parameters	Possible values	Description
		random rpa_pub rpa_rnd	Use random address for scan requests Use RPA address for scan requests (fallback to public if no IRK) Use RPA address for scan requests (fallback to random if no IRK)
	extended_duration extended_period	[0-UINT16_MAX]	Duration of extended scan in 10 milliseconds Periodic scan interval in 1.28 seconds (0 disabled)
	longrange_interval	[0-UINT16_MAX]	Scan interval for Coded Scan , if 0 use stack's default
	longrange_window	[0-UINT16_MAX]	Scan window for Coded Scan , if 0 use stack's default
	longrange_passive	[0-1]	Use passive scan for Coded Scan
connect			Initiate connection to remote device
	cancel		Cancel ongoing connection procedure
	extended	none 1M coded both all	Use legacy connection procedure Extended connect using 1M PHY scan parameters Extended connect using Coded PHY scan parameters Extended connect using 1M and Coded PHYs scan parameters Extended connect using 1M and Coded PHYs scan parameters (Provide also connection parameters for 2M PHY)
	peer_addr_type	public random public_id random_id	Remote device public address type Remote device random address type Remote device public address type (Identity) Remote device random address type (Identity)
	peer_addr	XX:XX:XX:XX:XX: public random rpa_pub rpa_rnd	Remote device address Use public address for scan requests Use random address for scan requests Use RPA address for scan requests (fallback to public if no IRK) Use RPA address for scan requests (fallback to random if no IRK)
	own_addr_type	[0-INT32_MAX]	Connection attempt duration, if 0 use stack's default
	scan_interval	[0-UINT16_MAX]	Scan interval, default: 0x0010
	scan_window	[0-UINT16_MAX]	Scan window, default: 0x0010
	interval_min	[0-UINT16_MAX]	Minimum connection interval, default: 30
	interval_max	[0-UINT16_MAX]	Maximum connection interval, default: 50
	latency	[UINT16]	Connection latency, default: 0
	timeout	[UINT16]	Connection timeout, default: 0x0100
	min_conn_event_len	[UINT16]	Minimum length of connection event, default: 0x0010
	max_conn_event_len	[UINT16]	Maximum length of connection event, default: 0x0300
	coded_scan_interval	[0-UINT16_MAX]	Coded PHY Scan interval, default: 0x0010

continues on next page

Table 8 – continued from previous page

Command	Parameters	Possible values	Description
	coded_scan_window coded_interval_min	[0-UINT16_MAX] [0-UINT16_MAX]	Coded PHY Scan window, default: 0x0010 Coded PHY Minimum connection interval, default: 30
	coded_interval_max	[0-UINT16_MAX]	Coded PHY Maximum connection interval, default: 50
	coded_latency coded_timeout	[UINT16] [UINT16]	Coded PHY Connection latency, default: 0 Coded PHY Connection timeout, default: 0x0100
	coded_min_conn_event_len	[UINT16]	Coded PHY Minimum length of connection event, default: 0x0010
	coded_max_conn_event_len	[UINT16]	Coded PHY Maximum length of connection event, default: 0x0300
	2M_scan_interval 2M_scan_window	[0-UINT16_MAX] [0-UINT16_MAX]	2M PHY Scan interval, default: 0x0010 2M PHY Scan window, default: 0x0010
	2M_interval_min	[0-UINT16_MAX]	2M PHY Minimum connection interval, default: 30
	2M_interval_max	[0-UINT16_MAX]	2M PHY Maximum connection interval, default: 50
	2M_latency 2M_timeout	[UINT16] [UINT16]	2M PHY Connection latency, default: 0 2M PHY Connection timeout, default: 0x0100
	2M_min_conn_event_len	[UINT16]	2M PHY Minimum length of connection event, default: 0x0010
	2M_max_conn_event_len	[UINT16]	2M PHY Maximum length of connection event, default: 0x0300
disconnect			Disconnect existing connection
	conn	[UINT16]	Connection handle
	reason	[UINT8]	Disconnect reason
show-addr			Show local public and random identity addresses
show-conn			Show current connections
conn-rssi			Obtain RSSI of specified connection
	conn	[UINT16]	Connection handle
conn-update-params			Update parameters of specified connection
	conn	[UINT16]	Connection handle
	interval_min	[0-UINT16_MAX]	Minimum connection interval, default: 30
	interval_max	[0-UINT16_MAX]	Maximum connection interval, default: 50
	latency	[UINT16]	Connection latency, default: 0
	timeout	[UINT16]	Connection timeout, default: 0x0100
	min_conn_event_len	[UINT16]	Minimum length of connection event, default: 0x0010
	max_conn_event_len	[UINT16]	Maximum length of connection event, default: 0x0300
conn-datalen			Set DLE parameters for connection
	conn	[UINT16]	Connection handle
	octets	[UINT16]	Maximum transmission packet size
	time	[UINT16]	Maximum transmission packet time
phy-set			Set preferred PHYs used for connection
	conn	[UINT16]	Connection handle

continues on next page

Table 8 – continued from previous page

Command	Parameters	Possible values	Description
phy-set-default	tx_phys_mask	[UINT8]	Prefered PHYs on TX is mask of following bits0x00 - no preference0x01 - 1M, 0x02 - 2M, 0x04 - Coded
	rx_phys_mask	[UINT8]	Prefered PHYs on RX is mask of following bits0x00 - no preference0x01 - 1M, 0x02 - 2M, 0x04 - Coded
	phy_opts	[UINT16]	Options for Coded PHY 0 - any coding, 1 - prefer S2, 2 - prefer S8
phy-set-default			Set default preferred PHYs used for new connection
phy-read	tx_phys_mask	[UINT8]	Prefered PHYs on TX is mask of following bits0x00 - no preference0x01 - 1M, 0x02 - 2M, 0x04 - Coded
	rx_phys_mask	[UINT8]	Prefered PHYs on RX is mask of following bits0x00 - no preference0x01 - 1M, 0x02 - 2M, 0x04 - Coded
phy-read			Read connection current PHY
l2cap-update	conn	[UINT16]	Connection handle
			Update connection parameters
	interval_min	[0-UINT16_MAX]	Minimum connection interval, default: 30
	interval_max	[0-UINT16_MAX]	Maximum connection interval, default: 50
l2cap-update			Connection latency, default: 0
l2cap-update			Connection timeout, default: 0x0100

Security

Table 9: Security

Command	Parameters	Possible values	Description
security-set-data	Set security configuration		
oob-flag	[0-1]		Set Out-Of-Band (OOB) flag in Security Manager
mitm-flag	[0-1]		Set Man-In-The-Middle (MITM) flag in Security Manager
io_capabilities	0 1 2 3 4		Set Input-Output Capabilities to “DisplayOnly” Set Input-Output Capabilities to “DisplayYesNo” Set Input-Output Capabilities to “KeyboardOnly” Set Input-Output Capabilities to “NoInputNoOutput” Set Input-Output Capabilities to “KeyboardDisplay”
our_key_dist	[UINT8]		Set Local Keys Distribution, this is a bit field of possible values: LTK (0x01), IRK (0x02), CSRK (0x04), LTK_SC(0x08)
their_key_dist	[UINT8]		Set Remote Keys Distribution, this is a bit field of possible values: LTK (0x01), IRK (0x02), CSRK (0x04), LTK_SC(0x08)
bonding-flag	[0-1]		Set Bonding flag in Security Manager
sc-flag	[0-1]		Set Secure Connections flag in Security Manager
security-pair	Start pairing procedure		
conn	[UINT16]		Connection handle
security-encryption	Start encryption procedure		
conn	[UINT16]		Connection handle
ediv	[UINT16]		EDIV for LTK to use (use storage if not provided)
rand	[UINT64]		Rand for LTK
ltk	XX:XX:XX:...		LTK (16 bytes)
security-start	Start security procedure (This starts either pairing or encryption depending if keys are stored)		
conn	[UINT16]		Connection handle
auth-passkey	Reply to Passkey request		
conn	[UINT16]		Connection handle
action	[UINT16]		Action to reply (as received in event)
key	[0-999999]		Passkey to reply (Input or Display action)
oob	XX:XX:XX:...		Out-Of-Band secret (16 bytes) (OOB action)
yesno	Yy-Ny		Confirm passkey (for Passkey Confirm action)

Advertising with Extended Advertising enabled

Table 10: Advertising with Extended Advertising enabled

Command	Parameters	Possible values	Description
advertise-configure	Configure new advertising instance		
instance	[0-UINT8_MAX]		Advertising instance
connectable	[0-1]		Use connectable advertising
scannable	[0-1]		Use scannable advertising
peer_addr_type	public		Remote device public address type

continues on next page

Table 10 – continued from previous page

Command	Parameters	Possible values	Description
	random		Remote device random address type
	public_id		Remote device public address type (Identity)
	random_id		Remote device random address type (Identity)
peer_addr	XX:XX:XX:XX:XX:		Remote device address - if provided perform directed advertising
own_addr_type	public		Use public address for scan requests
	random		Use random address for scan requests
	rpa_pub		Use RPA address for scan requests (fallback to public if no IRK)
	rpa_rnd		Use RPA address for scan requests (fallback to random if no IRK)
channel_map	[0-UINT8_MAX]		Primary advertising channels map. If 0 use all channels.
filter	none		Advertising filter policy - no filtering, no whitelist used
	scan		process all connection requests but only scans from white list
	conn		process all scan request but only connection requests from white list
	both		ignore all scan and connection requests unless in white list
interval_min	[0-UINT32_MAX]		Minimum advertising interval in 0.625 miliseconds If 0 use stack default.
interval_max	[0-UINT32_MAX]		Maximum advertising interval in 0.625 miliseconds If 0 use stack default.
rx_power	[-127 - 127]		Advertising TX power in dBm
primary_phy	1M		Use 1M PHY on primary advertising channels
	coded		Use Coded PHY on primary advertising channels
secondary_phy	1M		Use 1M PHY on secondary advertising channels
	coded		Use coded PHY on primary advertising channels
	2M		Use 2M PHY on primary advertising channels
sid	[0-16]		Advertise instance SID
high_duty	[0-1]		Use high_duty advertising
anonymous	[0-1]		Use anonymous advertising
legacy	[0-1]		Use legacy PDUs for advertising
include_tx_power	[0-1]		Include TX power information in advertising PDUs
scan_req_notif	[0-1]		Enable SCAN_REQ notifications
advertise-set-addr			Configure <i>random</i> address for instance
	instance	[0-UINT8_MAX]	Advertising instance
	addr	XX:XX:XX:XX:XX:	Random address
advertise-set-adv-data			Configure advertising instance ADV_DATA. This allow to configure following TLVs:
advertise-set-scan-rsp			Configure advertising instance SCAN_RSP. This allow to configure following TLVs:
	instance	[0-UINT8_MAX]	Advertising instance
	flags	[0-UINT8_MAX]	Flags value
	uuid16	[UINT16]	16-bit UUID value (can be passed multiple times)
	uuid16_is_complete	[0-1]	I 16-bit UUID list is complete

continues on next page

Table 10 – continued from previous page

Command	Parameters	Possible values	Description
	uuid32	[UINT32]	32-bit UUID value (can be passed multiple times)
	uuid32_is_complete	[0-1]	I 32-bit UUID list is complete
	uuid128	XX:XX:XX:...	128-bit UUID value (16 bytes) (can be passed multiple times)
	uuid128_is_complete	[0-1]	I 128-bit UUID list is complete
	tx_power_level	[-127 - 127]	TX Power level to include
	appearance	[UINT16]	Appearance
	name	string	Name
	advertising_interval	[UINT16]	Advertising interval
	ser- vice_data_uuid32	XX:XX:XX:...	32-bit UUID service data
	ser- vice_data_uuid128	XX:XX:XX:...	128-bit UUID service data
	uri	XX:XX:XX:...	URI
	msg_data	XX:XX:XX:...	Manufacturer data
	eddystone_url	string	Eddystone with specified URL
advertise- start			Start advertising with configured instance
	instance	[0-UINT8_MAX]	Advertising instance
	duration	[0-UINT16_MAX]	Advertising duration in 10ms units. 0 - forever
	max_events	[0-UINT8_MAX]	Maximum number of advertising events. 0 - no limit
advertise- stop			Stop advertising
advertise- remove	instance	[0-UINT8_MAX]	Advertising instance
			Remove configured advertising instance
	instance	[0-UINT8_MAX]	Advertising instance

Legacy Advertising with Extended Advertising disabled

Table 11: Legacy Advertising with Extended Advertising disabled

Command	Parameters	Possible values	Description
advertise			Enable advertising
	stop		Stop enabled advertising
	conn	und non dir	Connectable mode: undirected non-connectable directed
	discov	gen ltd non	Discoverable mode: general discoverable limited discoverable non-discoverable
	scannable	[0-1]	Use scannable advertising
	peer_addr_type	public random public_id random_id	Remote device public address type Remote device random address type Remote device public address type (Identity) Remote device random address type (Identity)

continues on next page

Table 11 – continued from previous page

L2CAP Connection Oriented Channels

Table 12: L2CAP Connection Oriented Channels

Command	Parameters	Possible values	Description
l2cap-create-server	psm	[UINT16]	Create L2CAP server PSM
l2cap-connect	conn	[UINT16]	Connect to remote L2CAP server Connection handle
	psm	[UINT16]	PSM
l2cap-disconnect	conn	[UINT16]	Disconnect from L2CAP server Connection handle
	idx	[UINT16]	L2CAP connection oriented channel identifier
l2cap-send	conn	[UINT16]	Send data over connected L2CAP channel Connection handle
	idx	[UINT16]	L2CAP connection oriented channel identifier
	bytes	[UINT16]	Number of bytes to send (hardcoded data pattern)
l2cap-show-coc			Show connected L2CAP channels

Keys storage

Table 13: Keys storage

Command	Parameters	Possible values	Description
keystore-add			Add keys to storage
	type	msec	Master Key
		ssec	Slave Key
		cccd	Client Characteristic Configuration Descriptor
	addr	XX:XX:XX:XX:XX:XX	Device address
	addr_type	public	Device address type
		random	Use random address for scan requests
	ediv	[UINT16]	EDIV for LTK to add
	rand	[UINT64]	Rand for LTK
	ltk	XX:XX:XX...	LTK (16 bytes)
	irk	XX:XX:XX...	Identity Resolving Key (16 bytes)
	csrk	XX:XX:XX...	Connection Signature Resolving Key (16 bytes)
keystore-del			Delete keys from storage
	type	msec	Master Key
		ssec	Slave Key
		cccd	Client Characteristic Configuration Descriptor
	addr	XX:XX:XX:XX:XX:XX	Device address
	addr_type	public	Device address type
		random	Use random address for scan requests
	ediv	[UINT16]	EDIV for LTK to remove
	rand	[UINT64]	Rand for LTK
keystore-show			Show stored keys
	type	msec	Master Keys
		ssec	Slave Keys
		cccd	Client Characteristic Configuration Descriptors

7.8.3 GATT feature API for btshell

Generic Attribute Profile (GATT) describes a service framework using the Attribute Protocol for discovering services, and for reading and writing characteristic values on a peer device. There are 11 features defined in the GATT Profile, and each of the features is mapped to procedures and sub-procedures:

Available commands

Configuration

Table 14: Configuration

Command	Parameters	Possible values	Description
gatt-discover-characteristic			Discover GATT characteristics
	conn	[UINT16]	Connection handle
	uuid	[UINT16]	Characteristic UUID
	start	[UINT16]	Discovery start handle
	end	[UINT16]	Discovery end handle
gatt-discover-descriptor			Discover GATT descriptors
	conn	[UINT16]	Connection handle
	start	[UINT16]	Discovery start handle
	end	[UINT16]	Discovery end handle
gatt-discover-service			Discover services
	conn	[UINT16]	Connection handle
	uuid16	[UINT16]	Service UUID
gatt-discover-full			Discover services, characteristic and descriptors
	conn	[UINT16]	Connection handle
gatt-find-included-services			Find included services
	conn	[UINT16]	Connection handle
	start	[UINT16]	Discovery start handle
	end	[UINT16]	Discovery end handle
gatt-exchange-mtu			Initiate ATT MTU exchange procedure
	conn	[UINT16]	Connection handle
gatt-read			Read attribute
	conn	[UINT16]	Connection handle
	long	[0-1]	Long read
	attr	[UINT16]	Attribute handle
	offset	[UINT16]	Long read offset value
	uuid	[UINT16]	Characteristic UUID
	start	[UINT16]	Discovery start handle
	end	[UINT16]	Discovery end handle
gatt-notify			Send notification or indication to all subscribed peers
	attr	[UINT16]	Attribute handle
gatt-service-changed			Send Services Changed notification
	start	[UINT16]	Start handle

continues on next page

Table 14 – continued from previous page

Command	Parameters	Possible values	Description
gatt-service-visibility	end	[UINT16]	End handle Set service visibility
	handle	[UINT16]	Service handle
gatt-show	visibility	[0-1]	Service visibility
gatt-show-local			Show remote devices discovered databases structure
gatt-write			Show local database structure
			Write attribute
gatt-write	conn	[UINT16]	Connection handle
	no_rsp	[0-1]	Use Write Without Response
	long	[0-1]	Use Long Write procedure
	attr	[UINT16]	Attribute handle
	offset	[UINT16]	Long write offset value
	value	XX:XX:XX...	Data to write

7.8.4 Advertisement Data Fields

This part defines the advertisement data fields used in the `btshell` app. For a complete list of all data types and formats used for Extended Inquiry Response (EIR), Advertising Data (AD), and OOB data blocks, refer to the Supplement to the Bluetooth Core Specification, CSSv6, available for download [here](#).

Table 15: Advertisement Data Fields

Name	Definition	Details	btshell Notes
flags	Indicates basic information about the advertiser.	Flags used over the LE physical channel are: * Limited Discoverable Mode * General Discoverable Mode * BR/EDR Not Supported * Simultaneous LE and BR/EDR to Same Device Capable (Controller) * Simultaneous LE and BR/EDR to Same Device Capable (Host)	NimBLE will auto-calculate if set to 0.
uuid16	16-bit Bluetooth Service UUIDs	Indicates the Service UUID list is incomplete i.e. more 16-bit Service UUIDs available. 16 bit UUIDs shall only be used if they are assigned by the Bluetooth SIG.	Set repeatedly for multiple service UUIDs.
uuid16_is_complete	16-bit Bluetooth Service UUIDs	Indicates the Service UUID list is complete. 16 bit UUIDs shall only be used if they are assigned by the Bluetooth SIG.	
uuid32	32-bit Bluetooth Service UUIDs	Indicates the Service UUID list is incomplete i.e. more 32-bit Service UUIDs available. 32 bit UUIDs shall only be used if they are assigned by the Bluetooth SIG.	Set repeatedly for multiple service UUIDs.
uuid32_is_complete	32-bit Bluetooth Service UUIDs	Indicates the Service UUID list is complete. 32 bit UUIDs shall only be used if they are assigned by the Bluetooth SIG.	

continues on next page

Table 15 – continued from previous page

Name	Definition	Details	btshell Notes
uuid128	Global 128-bit Service UUIDs	More 128-bit Service UUIDs available.	Set repeatedly for multiple service UUIDs.
uuid128_is_complete	Global 128-bit Service UUIDs	Complete list of 128-bit Service UUIDs	
tx_power_level	TX Power Level	Indicates the transmitted power level of the packet containing the data type. The TX Power Level data type may be used to calculate path loss on a received packet using the following equation: $\text{pathloss} = \text{Tx Power Level} - \text{RSSI}$ where ‘RSSI’ is the received signal strength, in dBm, of the packet received.	NimBLE will auto-calculate if set to -128.
slave_interval_range	Slave Connection Interval Range	Contains the Peripheral’s preferred connection interval range, for all logical connections. Size: 4 Octets . The first 2 octets defines the minimum value for the connection interval in the following manner: $\text{connIntervalMin} = \text{Conn_Interval_Min} * 1.25 \text{ ms}$ Conn_Interval_Min range: 0x0006 to 0x0C80 Value of 0xFFFF indicates no specific minimum. The other 2 octets defines the maximum value for the connection interval in the following manner: $\text{connIntervalMax} = \text{Conn_Interval_Max} * 1.25 \text{ ms}$ Conn_Interval_Max range: 0x0006 to 0x0C80 Conn_Interval_Max shall be equal to or greater than the Conn_Interval_Min. Value of 0xFFFF indicates no specific maximum.	
service_data_uuid16	Service Data - 16 bit UUID	Size: 2 or more octets The first 2 octets contain the 16 bit Service UUID followed by additional service data	
public_target_address	Public Target Address	Defines the address of one or more intended recipients of an advertisement when one or more devices were bonded using a public address. This data type shall exist only once. It may be sent in either the Advertising or Scan Response data, but not both.	
appearance	Appearance	Defines the external appearance of the device. The Appearance data type shall exist only once. It may be sent in either the Advertising or Scan Response data, but not both.	
advertising_interval	Advertising Interval	Contains the advInterval value as defined in the Core specification, Volume 6, Part B, Section 4.4.2.2.	
service_data_uuid32	Service Data - 32 bit UUID	Size: 4 or more octets The first 4 octets contain the 32 bit Service UUID followed by additional service data	
service_data_uuid128	Service Data - 128 bit UUID	Size: 16 or more octets The first 16 octets contain the 128 bit Service UUID followed by additional service data	

continues on next page

Table 15 – continued from previous page

Name	Definition	Details	btshell Notes
uri	Uniform Resource Identifier (URI)	Scheme name string and URI as a UTF-8 string	
mfg_data	Manufacturer Specific data	Size: 2 or more octets The first 2 octets contain the Company Identifier Code followed by additional manufacturer specific data	
eddystone_url			

7.9 Bluetooth Mesh

- *Introduction to Mesh*
- *Topology*
- *Bearers*
- *Provisioning*
- *Models*
- *Mesh Node features supported by Apache Mynewt*

7.9.1 Introduction to Mesh

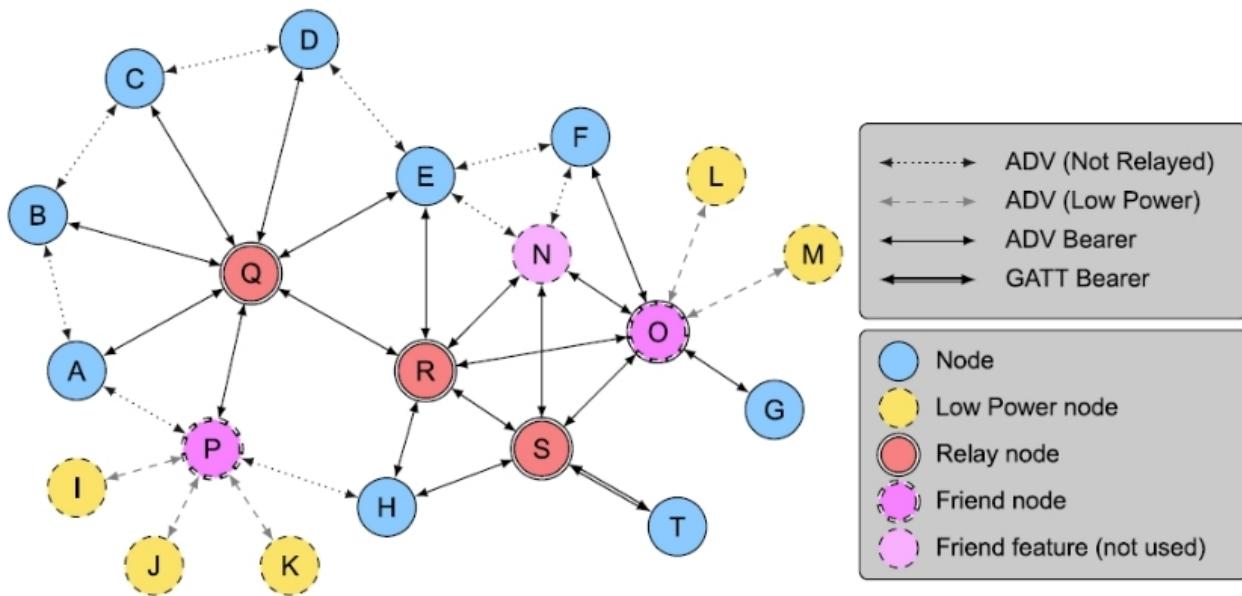
Bluetooth Mesh is a new standard from Bluetooth SIG that was released in 2017. It enables many-to-many device communication (as opposed to point-to-point approach in BLE) and is optimised for large-scale networks like building automation or sensors network. It utilizes managed flood based approach where only mains-powered nodes relay messages making it very power efficient (battery powered low-power nodes that don't relay messages can operate in mesh network for years).

Bluetooth Mesh is complementary to Bluetooth specification and requires features from 4.0 release only. This allows deployment of networks using hardware already available on the market.

7.9.2 Topology

Bluetooth Mesh defines few features (roles) for devices in network. Those are:

- Relay - receive and retransmit mesh messages over the advertising bearer to enable larger networks
- Proxy - receive and retransmit mesh messages between GATT and advertising bearers.
- Low Power - operate within a mesh network at significantly reduced receiver duty cycles only in conjunction with a node supporting the Friend feature
- Friend - the ability to help a node supporting the Low Power feature to operate by storing messages destined for those nodes



7.9.3 Bearers

Mesh Profile specification allows two kinds of bearers for transmitting data:

- Advertising Bearer
 - Uses LE advertising to broadcast messages to all nodes that are listening at this time
 - Uses non-connectable advertising only
 - 29 octets of network message
- GATT Bearer
 - Uses LE Connections to send messages
 - Uses standard GATT service (one for Provisioning and one for Proxy)

7.9.4 Provisioning

Provisioning is a process of adding an unprovisioned device to a mesh network managed by a Provisioner. A Provisioner provides the unprovisioned device with provisioning data that allows it to become a mesh node (network key, current IV index and unicast address). A Provisioner is typically a smart phone or other mobile computing device.

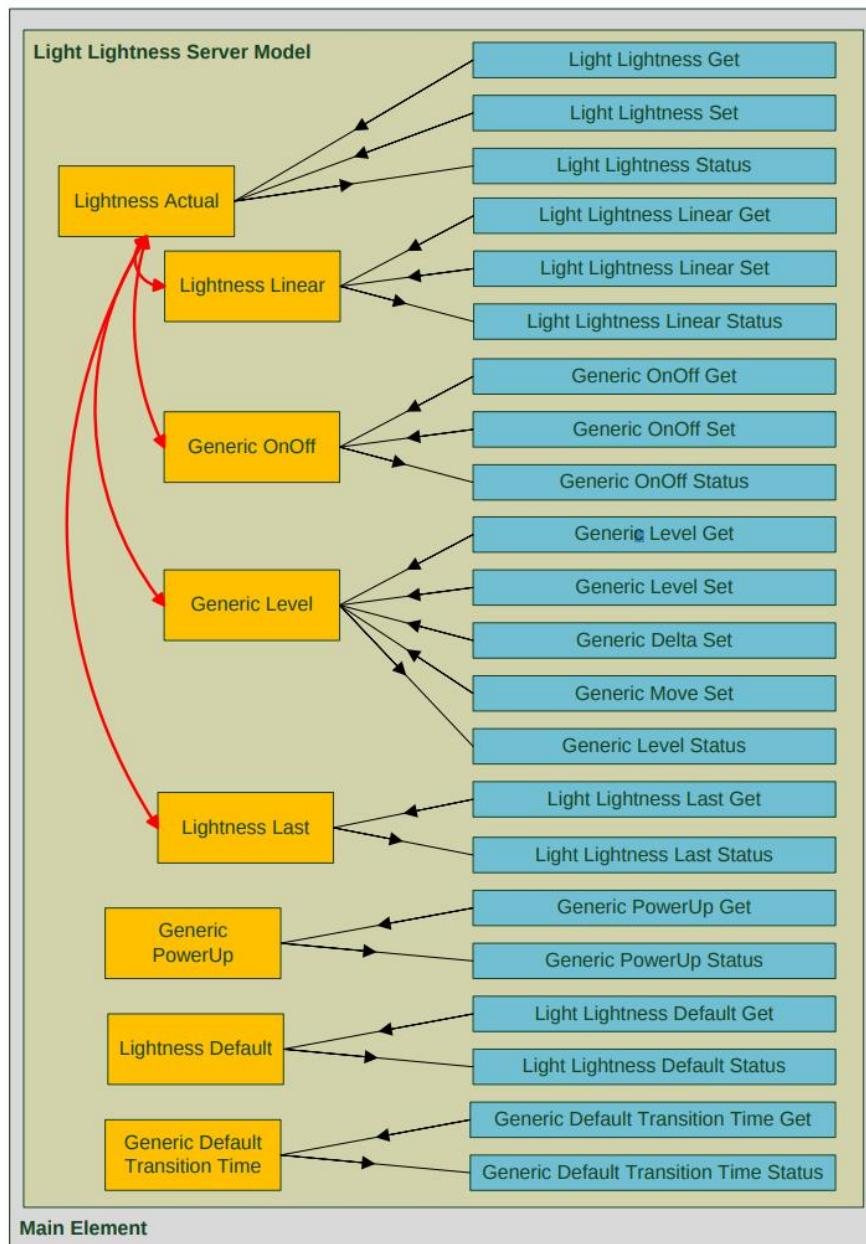
7.9.5 Models

Models define basic functionality of nodes on a mesh network. Mesh Profile Specification defines foundation models used to configure and manage network. Mesh Model Specification includes models defining functionality that is standard across device types. Those consists of:

- Generics - root models
 - On/Off
 - Level
 - Battery Server

- Location
- Client Property
- and others
- Sensors - defines a standard way of interfacing with sensors
- Time and Scenes - defines a set of functionalities related to time and saved states on devices
- Lighting - defines a set functionalities related to lighting control

Complex models e.g. Lighting may contain other models eg Generic On/Off. The following image shows an example of Light Lightness Server Model.



7.9.6 Mesh Node features supported by Apache Mynewt

- Advertising and GATT bearers
- PB-GATT and PB-ADV provisioning
- Foundation Models (server role)
- Relay support
- GATT Proxy

7.9.7 Sample application

blemesh sample application implements Bluetooth Mesh node that supports On/Off and Level models.

To build application use following target. Note that since this application uses Non-resolvable Private Address there is no need for configuring public address.

```
newt target create blemesh
newt target set blemesh app=@apache-mynewt-nimble/apps/blemesh
newt target set blemesh bsp=@apache-mynewt-core/hw/bsp/nordic_pca10056
newt target set blemesh build_profile=optimized
newt target set blemesh syscfg=BLE_MESH_PB_GATT=1:BLE_MESH_DEV_UUID='(uint8_t[16]){\0x22, \0x20, \0}'
```

Every device should have unique Device UUID so config amend and rebuild is needed for each of the devices that will be added to a network.

```
newt target set blemesh syscfg=BLE_MESH_PB_GATT=1:BLE_MESH_DEV_UUID='(uint8_t[16]){\0x22, \0x21, \0}'  
...
newt target set blemesh syscfg=BLE_MESH_PB_GATT=1:BLE_MESH_DEV_UUID='(uint8_t[16]){\0x22, \0x22, \0}'  
...
newt target set blemesh syscfg=BLE_MESH_PB_GATT=1:BLE_MESH_DEV_UUID='(uint8_t[16]){\0x22, \0x23, \0}'
```

GATT bearer is enabled so that it is possible to provision those with Bluetooth Mesh application from Silicon Labs (available [here](#)) which doesn't support advertising bearer.

NEWT TOOL GUIDE

8.1 Introduction

Newt is a smart build and package management system for embedded contexts. It is a single tool that accomplishes both the following goals:

- source package management
- build, debug and install.

8.1.1 Rationale

In order for the Mynewt operating system to work well for constrained environments across the many different types of micro-controller applications (from doorbells to medical devices to power grids), a system is needed that lets you select which packages to install and which packages to build.

The build systems for embedded devices are often fairly complicated and not well served for this purpose. For example, autoconf is designed for detecting system compatibility issues but not well suited when it comes to tasks like:

- Building for multiple targets
- Deciding what to build in and what not to build in
- Managing dependencies between components

Fortunately, solutions addressing these very issues can be found in source package management systems in higher level languages such as Javascript (Node), Go, PHP and Ruby. We decided to fuse their source management systems with a make system built for embedded systems and create Newt.

8.1.2 Build System

A good build system must allow the user to take a few common steps while developing embedded applications:

- Generate full flash images
- Download debug images to a target board using a debugger
- Conditionally compile libraries & code based upon build settings

Newt can read a directory tree, build a dependency tree, and emit the right build artifacts. An example newt source tree is in mynewt-blinky/develop:

```
$ tree -L 3 .
├── DISCLAIMER
├── LICENSE
├── NOTICE
└── README.md
├── apps
│   └── blinky
│       ├── pkg.yml
│       └── src
└── project.yml
└── targets
    ├── my_blinky_sim
    │   ├── pkg.yml
    │   └── target.yml
    └── unittest
        └── pkg.yml
            └── target.yml
```

6 directories, 10 files

When Newt sees a directory tree that contains a “project.yml” file, it is smart enough to recognize it as the base directory of a project, and automatically builds a package tree. It also recognizes two important package directories in the package tree - “apps” and “targets”. More on these directories in [Theory of Operations](#).

When Newt builds a target, it recursively resolves all package dependencies, and generates artifacts that are placed in the bin/targets/<target-name>/app/apps/<app-name> directory, where the bin directory is under the project base directory, target-name is the name of the target, and app-name is the name of the application. For our example `my_blinky_sim` is the name of the target and `blinky` is the name of the application. The `blinky.elf` executable is stored in the bin/targets/my_blinky_sim/app/apps/blinky directory as shown in the source tree:

```
$ tree -L 6 bin/
bin/
└── targets
    └── my_blinky_sim
        ├── app
        │   └── apps
        │       └── blinky
        │           ├── apps
        │           ├── apps_blinky.a
        │           ├── apps_blinky.a.cmd
        │           ├── blinky.elf
        │           ├── blinky.elf.cmd
        │           ├── blinky.elf.dSYM
        │           ├── blinky.elf.lst
        │           └── manifest.json
        └── hw
            ├── bsp
            │   └── native
            ├── drivers
            │   └── uart
            └── hal
                ├── hw_hal.a
                └── hw_hal.a.cmd
```

(continues on next page)

(continued from previous page)

```

|   |   |   |
      └ repos
<snip>

```

8.1.3 More operations using Newt

Once a target has been built, Newt allows additional operations on the target.

- **load**: Download built target to board
- **debug**: Open debugger session to target
- **size**: Get size of target components
- **create-image**: Add image header to the binary image
- **run**: Build, create image, load, and finally open a debug session with the target
- **target**: Create, delete, configure, and query a target

For more details on how Newt works, go to [Theory of Operations](#).

8.1.4 Source Management and Repositories

The other major element of the Newt tool is the ability to create reusable source distributions from a collection of code. **A project can be a reusable container of source code**. In other words, projects can be versioned and redistributed, not packages. A project bundles together packages that are typically needed to work together in a product e.g. RTOS core, filesystem APIs, and networking stack.

A project that has been made redistributable is known as a **repository**. Repositories can be added to your local project by adding them into your project.yml file. Here is an example of the blinky project's yml file which relies on apache-mynewt-core:

```
$ more project.yml
<snip>
project.repositories:
    - apache-mynewt-core

repository.apache-mynewt-core:
    type: github
    vers: 1-latest
    user: apache
    repo: incubator-mynewt-core
```

When you specify this repository in the blinky's project file, you can then use the Newt tool to install dependencies:

```
$ newt install
Downloading repository description for apache-mynewt-core... success!
Downloading repository incubator-mynewt-core (branch: develop) at https://github.com/
→apache/incubator-mynewt-core.git
Cloning into '/var/folders/71/7b3w9m4n2mg3sqmgw2q1b9p80000gn/T/newt-repo814721459'...
remote: Counting objects: 17601, done.
remote: Compressing objects: 100% (300/300), done.
remote: Total 17601 (delta 142), reused 0 (delta 0), pack-reused 17284
```

(continues on next page)

(continued from previous page)

```
Receiving objects: 100% (17601/17601), 6.09 MiB \| 3.17 MiB/s, done.  
Resolving deltas: 100% (10347/10347), done.  
Checking connectivity... done.  
Repos successfully installed
```

Newt will install this repository in the <project>/repos directory. In the case of blinky, the directory structure ends up looking like:

```
$ tree -L 2  
. .  
├── DISCLAIMER  
├── LICENSE  
├── NOTICE  
├── README.md  
├── apps  
│   └── blinky  
├── project.state  
├── project.yml  
└── repos  
    └── apache-mynewt-core  
        ├── targets  
        │   └── my_blinky_sim  
        └── unittest
```

In order to reference the installed repositories in packages, the “@” notation should be specified in the repository specifier. As an example, the apps/blinky application has the following dependencies in its pkg.yml file. This tells the build system to look in the base directory of repos/apache-mynewt-core for the kernel/os, hw/hal, and sys/console/full packages.

```
$ more apps/blinky/pkg.yml  
pkg.deps:  
- "@apache-mynewt-core/kernel/os"  
- "@apache-mynewt-core/hw/hal"  
- "@apache-mynewt-core/sys/console/full"
```

Newt has the ability to autocomplete within bash. The following instructions allow MAC users to enable autocomplete within bash.

1. Install the autocomplete tools for bash via `brew install bash-completion`
2. Tell your shell to use newt for autocomplete of newt via `complete -C "newt complete" newt`. You can add this to your `.bashrc` or other init file to have it automatically set for all bash shells.

8.1.5 Notes:

1. Autocomplete will give you flag hints, but only if you type a ‘-’.
2. Autocomplete will not give you completion hints for the flag arguments (those optional things after the flag like `-l DEBUG`)
3. Autocomplete uses newt to parse the project to find targets and libs.

8.2 Theory of Operations

Newt has a fairly smart package manager that can read a directory tree, build a dependency tree, and emit the right build artifacts.

8.2.1 Building dependencies

Newt can read a directory tree, build a dependency tree, and emit the right build artifacts. An example newt source tree is in mynewt-blinky/develop:

```
$ tree -L 3
.
├── LICENSE
├── NOTICE
├── README.md
└── apps
    └── blinky
        ├── pkg.yml
        └── src
└── project.yml
└── targets
    └── my_blinky_sim
        ├── pkg.yml
        └── target.yml
└── unittest
    ├── pkg.yml
    └── target.yml

6 directories, 9 files
```

When newt sees a directory tree that contains a “project.yml” file it knows that it is in the base directory of a project, and automatically builds a package tree. You can see that there are two essential package directories, “apps” and “targets.”

“apps” Package Directory

apps is where applications are stored, and applications are where the main() function is contained. The base project directory comes with one simple app called `blinky` in the `apps` directory. The core repository `@apache-mynewt-core` comes with many additional sample apps in its `apps` directory. At the time of this writing, there are several example BLE apps, the boot app, slinky app for using newt manager protocol, and more in that directory.

```
$ ls repos/apache-mynewt-core/apps
blecent blehr      blemesh_shell  blesplit  boot      btshell      lora_app_shell  ↵
↳ ocf_sample   slinky     splitty      trng_test
blecsc  blemesh      bleprph      bletest    bsncent   ffs2native  loraping    ↵
↳ pwm_test     slinky_oic testbench
blehci  blemesh_light bleprph_oic  bleuart   bsnprph  iptest     lorashell    ↵
↳ sensors_test spitest    timtest
```

Along with the `targets` directory, `apps` represents the top-level of the build tree for the particular project, and define the dependencies for the rest of the system. Mynewt users and developers can add their own apps to the project’s `apps` directory.

The app definition is contained in a `pkg.yml` file. For example, `blinky`’s `pkg.yml` file is:

```
$ more apps/blinky/pkg.yml
pkg.name: apps/blinky
pkg.type: app
pkg.description: Basic example application which blinks an LED.
pkg.author: "Apache Mynewt <dev@mynewt.apache.org>"
pkg.homepage: "http://mynewt.apache.org/"
pkg.keywords:

pkg.deps:
  - "@apache-mynewt-core/kernel/os"
  - "@apache-mynewt-core/hw/hal"
  - "@apache-mynewt-core/sys/console/stub"
```

This file says that the name of the package is apps/blinky, and it depends on the kernel/os, hw/hal and sys/console/stub packages.

NOTE: @apache-mynewt-core is a repository descriptor, and this will be covered in the “repository” section.

“targets” Package Directory

targets is where targets are stored, and each target is a collection of parameters that must be passed to newt in order to generate a reproducible build. Along with the apps directory, targets represents the top of the build tree. Any packages or parameters specified at the target level cascades down to all dependencies.

Most targets consist of:

- app: The application to build
- bsp: The board support package to combine with that application
- build_profile: Either debug or optimized.

The my_blinky_sim target that is included by default has the following settings:

```
$ newt target show
targets/my_blinky_sim
  app=apps/blinky
  bsp=@apache-mynewt-core/hw/bsp/native
  build_profile=debug
$ ls targets/my_blinky_sim/
  pkg.yml          target.yml
```

There are helper functions to aid the developer specify parameters for a target.

- **vals**: Displays all valid values for the specified parameter type (e.g. bsp for a target)
- **target show**: Displays the variable values for either a specific target or all targets defined for the project
- **target set**: Sets values for target variables

In general, the three basic parameters of a target (app, bsp, and build_profile) are stored in the target’s target.yml file in the targets/<target-name> directory, where target-name is the name of the target. You will also see a pkg.yml file in the same directory. Since targets are packages, a pkg.yml is expected. It contains typical package descriptors, dependencies, and additional parameters such as the following:

- Cflags: Any additional compiler flags you might want to specify to the build
- Aflags: Any additional assembler flags you might want to specify to the build

- Lflags: Any additional linker flags you might want to specify to the build

You can also override the values of the system configuration settings that are defined by the packages that your target includes. You override the values in your target's `syscfg.yml` file (stored in the `targets/<target-name>` directory). You can use the `newt target config show` command to see the configuration settings and values for your target, and use the `newt target set` command to set the `syscfg` variable and override the configuration setting values. You can also use an editor to create your target's `syscfg.yml` file and add the setting values to the file. See [System Configuration Setting Definitions and Values](#) for more information on system configuration settings.

8.2.2 Resolving dependencies

When newt builds a project, it will:

- find the top-level `project.yml` file
- recurse the packages in the package tree, and build a list of all source packages

Newt then looks at the target that the user set, for example, `blinky_sim`:

```
$ more targets/my_blinky_sim/target.yml
## Target: targets/my_blinky_sim
target.app: "apps/blinky"
target.bsp: "@apache-mynewt-core/hw/bsp/native"
target.build_profile: "debug"
```

The target specifies two major things:

- Application (`target.app`): The application to build
- Board Support Package (`target.bsp`): The board support package to build along with that application.

Newt builds the dependency tree specified by all the packages. While building this tree, it does a few other things:

- Sets up the include paths for each package. Any package that depends on another package, automatically gets the include directories from the package it includes. Include directories in the newt structure must always be prefixed by the package name. For example, `kernel/os` has the following include tree and its include directory files contains the package name “os” before any header files. This is so in order to avoid any header file conflicts.

```
$ tree repos/apache-mynewt-core/kernel/os/include/
repos/apache-mynewt-core/kernel/os/include/
└── os
    ├── arch
    │   ├── cortex_m0
    │   │   └── os
    │   │       └── os_arch.h
    │   ├── cortex_m4
    │   │   └── os
    │   │       └── os_arch.h
    │   ├── mips
    │   │   └── os
    │   │       └── os_arch.h
    │   ├── sim
    │   │   └── os
    │   │       └── os_arch.h
    │   └── sim-mips
    │       └── os
    │           └── os_arch.h
```

(continues on next page)

(continued from previous page)

```

└── endian.h
└── os.h
└── os_callout.h
└── os_cfg.h
└── os_cputime.h
└── os_dev.h
└── os_eventq.h
└── os_fault.h
└── os_heap.h
└── os_malloc.h
└── os_mbuf.h
└── os_mempool.h
└── os_mutex.h
└── os_sanity.h
└── os_sched.h
└── os_sem.h
└── os_task.h
└── os_test.h
└── os_time.h
└── queue.h

```

12 directories, 25 files

- Validates API requirements. Packages can export APIs they implement, (i.e. pkg.api: hw-hal-impl), and other packages can require those APIs (i.e. pkg.req_api: hw-hal-impl).
- Reads and validates the configuration setting definitions and values from the package `syscfg.yaml` files. It generates a `syscfg.h` header file that packages include in the source files in order to access the settings. It also generates a system initialization function to initialize the packages. See *System Configuration Setting Definitions and Values* for more information.

In order to properly resolve all dependencies in the build system, newt recursively processes the package dependencies until there are no new dependencies. And it builds a big list of all the packages that need to be build.

Newt then goes through this package list, and builds every package into an archive file.

NOTE: The newt tool generates compiler dependencies for all of these packages, and only rebuilds the packages whose dependencies have changed. Changes in package & project dependencies are also taken into account. It is smart, after all!

8.2.3 Producing artifacts

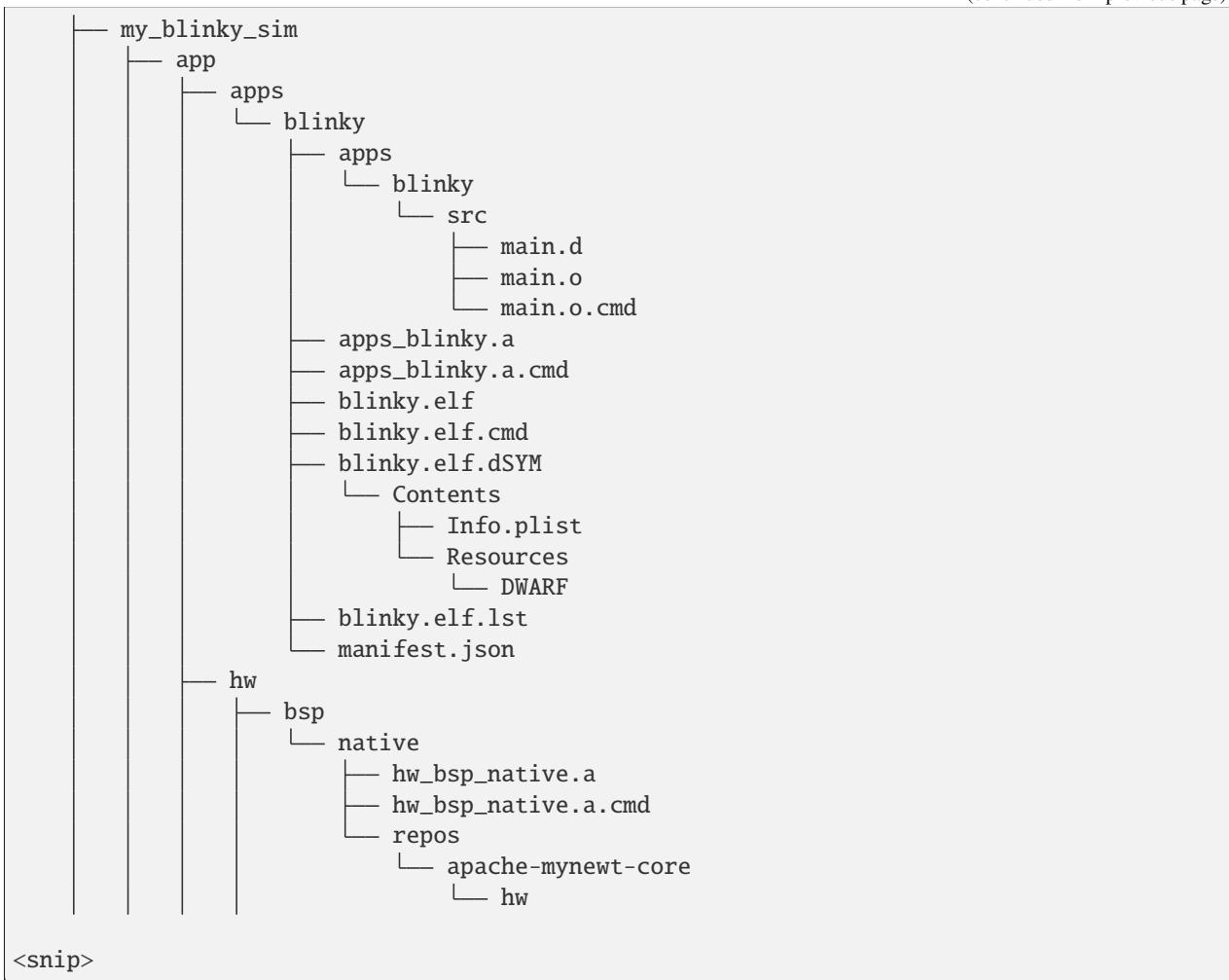
Once newt has built all the archive files, it then links the archive files together. The linkerscript to use is specified by the board support package (BSP.)

The newt tool creates a bin directory under the base project directory, and places a target's build artifacts into the bin/targets/<target-name>/app/apps/<app-name> directory, where `target-name` is the name of the target and `app-name` is the name of the application. As an example, the `blinky.elf` executable for the `blinky` application defined by the `my_blinky_sim` target is stored in the bin/targets/my_blinky_sim/app/apps/blinky directory as shown in the following source tree:

```
$tree -L 9 bin/
bin/
└── targets
```

(continues on next page)

(continued from previous page)



As you can see, a number of files are generated:

- Archive File
- *.cmd: The command use to generate the object or archive file
- *.lst: The list file where symbols are located

Note: The *.o object files that get put into the archive file are stored in the bin/targets/my_blinky_sim/app/apps/blinky/apps/blinky/src directory.

8.2.4 Download/Debug Support

Once a target has been built, there are a number of helper functions that work on the target. These are:

- **load** Download built target to board
- **debug** Open debugger session to target
- **size** Size of target components
- **create-image** Add image header to target binary
- **run** The equivalent of build, create-image, load, and debug on specified target

- **target** Create, delete, configure, and query a target

load and debug handles driving GDB and the system debugger. These commands call out to scripts that are defined by the BSP.

```
$ more repos/apache-mynewt-core/hw/bsp/nrf52dk/nrf52dk_debug.sh
<snip>
. $CORE_PATH/hw/scripts/jlink.sh

FILE_NAME=$BIN_BASENAME.elf

if [ $# -gt 2 ]; then
    SPLIT_ELF_NAME=$3.elf
    # TODO -- this magic number 0x42000 is the location of the second image
    # slot. we should either get this from a flash map file or somehow learn
    # this from the image itself
    EXTRA_GDB_CMDS="add-symbol-file $SPLIT_ELF_NAME 0x8000 -readnow"
fi

JLINK_DEV="nRF52"

jlink_debug
```

The idea is that every BSP will add support for the debugger environment for that board. That way common tools can be used across various development boards and kits.

8.3 Command Structure

Just like verbs are actions in a sentence and adverbs modify verbs, so in the *newt* tool, commands are actions and flags modify actions. A command can have subcommands. Arguments to commands and subcommands, with appropriate flags, dictate the execution and result of a command.

For instance, in the example below, the *newt* command has the subcommand *target set* in which the argument ‘my_target1’ is the target whose attribute, *app*, is set to @apache-mynewt-core/hw/bsp/nrf52dk

```
newt target set my_target1 app=@apache-mynewt-core/hw/bsp/nrf52dk
```

Global flags work uniformly across *newt* commands. Consider the flag *-v*, *--verbose*, It works both for command and subcommands, to generate verbose output. Likewise, the help flag *-h* or *--help*, to print helpful messages.

A command may additionally take flags specific to it. For example, the *-n* flag instructs *newt debug* not to start GDB from command line.

```
newt debug <target-name> -n
```

In addition to the documentation in *Newt Tool Guide*, command-line help is available for each command (and subcommand), through the *-h* or *--help* options.

```
newt target --help
Commands to create, delete, configure, and query targets

Usage:
  newt target [flags]
  newt target [command]
```

(continues on next page)

(continued from previous page)

Available Commands:

amend	Add, change, or delete values for multi-value target variables
config	View or populate a target's system configuration
copy	Copy target
create	Create a target
delete	Delete target
dep	View target's dependency graph
revdep	View target's reverse-dependency graph
set	Set target configuration variable
show	View target configuration variables

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

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

8.3.1 newt build

Build one or more targets.

Usage:

newt build <target-name> [target_name ...] [flags]

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

Compiles, links, and builds an ELF binary for the target named <target-name>. It builds an ELF file for the application specified by the app variable for the target-name target. The image can be loaded and run on the hardware specified by the bsp variable for the target. The command creates the ‘bin/’ directory under the project’s base directory (that the newt new command created) and stores the executable in the ‘bin/targets/<target-name>/app/apps/<app-name>’ directory. A manifest.json build manifest file is also generated in the same directory. This build manifest contains information such as build time, version, image name, a hash to identify the image, packages actually used to create the build, and the target for which the image is built.

You can specify a list of target names, separated by a space, to build multiple targets.

Examples

Usage	Explanation
<code>newt build my_blinky_sim</code>	Builds an executable for the my_blinky_sim target. For example, if the my_blinky_sim target has app set to apps/blinky, you will find the generated .elf, .a, and .lst files in the ‘bin/targets/my_blinky_sim/app/apps/blinky’ directory.
<code>newt build my_blinky_sim myble</code>	Builds the images for the applications defined by the my_blinky_sim and myble targets.

8.3.2 newt clean

Delete build artifacts for one or more targets.

Usage:

```
newt clean <target-name> [target-name...] | all [flags]
```

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

Deletes all the build artifacts generated for the `target-name` target. It does not delete the target definition. You can specify a list of targets, separated by a space, to delete the artifacts for multiple targets, or specify `all` to delete the artifacts for all targets.

Examples

Usage	Explanation
<code>newt clean myble</code>	Deletes the ‘bin/targets/myble’ directory where all the build artifacts generated from the <code>myble</code> target build are stored.
<code>newt clean my_blinky_sim myble</code>	Deletes the ‘bin/targets/my_blinky_sim’ and the ‘bin/targets/myble’ directories where all the artifacts generated from the <code>my_blinky_sim</code> and <code>myble</code> target builds are stored.
<code>newt clean all</code>	Removes the artifacts for all target builds. Deletes the top level ‘bin’ directory.

8.3.3 newt complete

Performs bash autocompletion using tab. It is not intended to be called directly from the command line.

Install bash autocompletion

```
$ brew install bash-completion
Updating Homebrew...
<snip>
Bash completion has been installed to:
/usr/local/etc/bash_completion.d
==> Summary
/usr/local/Cellar/bash-completion/1.3_1: 189 files, 607.8K
```

Enable autocompletion for newt

```
$ complete -C "newt complete" newt
```

Usage

Hit tab and see possible completion options or completed command.

```
$ newt target s
set show
$ newt target show
```

8.3.4 newt create-image

Create and sign an image by adding an image header to the binary file created for a target. Version number in the header is set to <version>. To sign an image provide a .pem file for the signing-key and an optional key-id.

Usage:

```
newt create-image <target-name> <version> [signing-key [key-id]][flags]
```

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

Adds an image header to the created binary file for the `<target-name>` target. The image version is set to `<version>`. It creates a `<app-name>.img` file the image, where `app-name` is the value specified in the target app variable, and stores the file in the `'/bin/targets/<target-name>/app/apps/<app-name>/'` directory. It also creates a `<app-name>.hex` file for the image in the same directory, and adds the version, build id, image file name, and image hash to the `manifest.json` file that the `newt build` command created.

To sign an image, provide a .pem file for the `signing-key` and an optional `key-id`. `key-id` must be a value between 0-255.

Examples

Usage	Explanation
<code>newt create-image myble2 1.0.1.0</code>	<p>Creates an image for target <code>myble2</code> and assigns it version <code>1.0.1.0</code>.</p> <p>For the following target definition:</p> <pre>targets/myble2 app=@apache-mynewt-core/apps/btshell bsp=@apache-mynewt-core/hw/bsp/nrf52dk build_profile=optimized syscfg=STATS_NAMES=1</pre> <p>the ‘bin/targets/myble2/app/apps/btshell/btshell.img’ and ‘bin/targets/myble2/app/apps/btshell/btshell.hex’ files are created, and the manifest in ‘bin/targets/myble2/app/apps/btshell/manifest.json’ is updated with the image information.</p>
<code>newt create-image myble2 1.0.1.0 private.pem</code>	<p>Creates an image for target <code>myble2</code> and assigns it the version <code>1.0.1.0</code>. Signs the image using private key specified by the <code>private.pem</code> file.</p>

8.3.5 newt debug

Open a debugger session to a target.

Usage:

```
newt debug <target-name> [flag]
```

Flags:

<code>--extraJTAGcmd</code> string	Extra commands to send to JTAG software
<code>-n, --noGDB</code>	Do not start GDB from command line

Global Flags:

<code>-h, --help</code>	Help for newt commands
<code>-j, --jobs</code> int	Number of concurrent build jobs (default 8)
<code>-l, --loglevel</code> string	Log level (default "WARN")
<code>-o, --outfile</code> string	Filename to tee output to
<code>-q, --quiet</code>	Be quiet; only display error output
<code>-s, --silent</code>	Be silent; don't output anything
<code>-v, --verbose</code>	Enable verbose output when executing commands

Description

Opens a debugger session to the image built for the <target-name> target.

Examples

Usage	Explanation
<code>newt debug myble2</code>	Opens a J-Link connection and starts a GNU gdb session to debug bin/targets/myble2/app/apps/btshell/btshell.elf when the target is as follows: targets/myble2 app=@apache-mynewt-core/apps/btshell bsp=@apache-mynewt-core/hw/bsp/nrf52dk build_profile=optimized syscfg=STATS_NAMES=1
<code>newt debug myble2 -n</code>	Opens a J-Link connection bin/targets/myble2/app/apps/btshell/btshell.elf but do not start GDB on the command line.

8.3.6 newt help

Display the help text for the newt command line tool:

Newt allows you to create your own embedded application based on the Mynewt operating system. Newt provides both build and package management in a single tool, which allows you to compose an embedded application, and set of projects, and then build the necessary artifacts from those projects. For more information on the Mynewt operating system, please visit
<https://mynewt.apache.org/>.

Usage:

```
newt help [command]
```

Global Flags:

<code>-h, --help</code>	Help for newt commands
<code>-j, --jobs int</code>	Number of concurrent build jobs (default 8)
<code>-l, --loglevel string</code>	Log level (default "WARN")
<code>-o, --outfile string</code>	Filename to tee output to
<code>-q, --quiet</code>	Be quiet; only display error output
<code>-s, --silent</code>	Be silent; don't output anything
<code>-v, --verbose</code>	Enable verbose output when executing commands

Available Commands:

build	Build one or more targets
clean	Delete build artifacts for one or more targets
create-image	Add image header to target binary
debug	Open debugger session to target
info	Show project info
install	Install project dependencies
load	Load built target to board
mfg	Manufacturing flash image commands
new	Create a new project
pkg	Create and manage packages in the current workspace
run	build/create-image/download/debug <target>
size	Size of target components
sync	Synchronize project dependencies
target	Command for manipulating targets
test	Executes unit tests for one or more packages
upgrade	Upgrade project dependencies
vals	Display valid values for the specified element type(s)
version	Display the Newt version number

Examples

Usage	Explanation
<code>newt help target</code>	Displays the help text for the newt target command
<code>newt help</code>	Displays the help text for newt tool

8.3.7 newt info

Show information about the current project.

Usage:

```
newt info [flags]
```

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

Displays the repositories in the current project (the local as well as all the external repositories fetched). It also displays the packages in the local repository.

8.3.8 newt load

Load application image onto the board for a target.

Usage:

```
newt load <target-name> [flags]
```

Flags:

```
--extraJTAGcmd string    Extra commands to send to JTAG software
```

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

Uses download scripts to automatically load, onto the connected board, the image built for the app defined by the `target-name` target. If the wrong board is connected or the target definition is incorrect (i.e. the wrong values are given for bsp or app), the command will fail with error messages such as `Can not connect to J-Link via USB` or `Unspecified error -1`.

8.3.9 newt mfg

Commands to create, build, and upload manufacturing image.

Usage:

```
newt mfg [flags]
newt mfg [command]
```

Available Commands:

create	Create a manufacturing flash image
deploy	Build and upload a manufacturing image (build + load)
load	Load a manufacturing flash image onto a device

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

Sub-command	Explanation
create	A manufacturing image specifies 1) a boot loader target, and 2) one or more image targets. Assuming the manufacturing entity has been created and defined in the <code>mfgs/<mfg image name>/</code> package (see Examples below), this command collects the manufacturing related files in the newly created <code>bin/mfgs/<mfg image name></code> directory. The collection includes the image file, the hex file, and the manifests with the image build time, version, manufacturing package build time, image ID (or hash) etc. It is essentially a snapshot of the image data and metadata uploaded to the device flash at manufacturing time. Note that the command expects the targets and images to have already been built using <code>newt build</code> and <code>newt create-image</code> commands.
deploy	A combination of build and load commands to put together and upload manufacturing image on to the device.
load	Loads the manufacturing package onto to the flash of the connected device.

Examples

Suppose you have created two targets (one for the bootloader and one for the `blinky` app).

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

(continues on next page)

(continued from previous page)

```
targets/rb_blinky
    app=apps/blinky
    bsp=@apache-mynewt-core/hw/bsp/rb-nano2
    build_profile=debug
targets/rb_boot
    app=@apache-mynewt-core/apps/boot
    bsp=@apache-mynewt-core/hw/bsp/rb-nano2
    build_profile=optimized
```

Create the directory to hold the mfg packages.

```
$ mkdir -p mfgs/rb_blinky_rsa
```

The rb_blinky_rsa package needs a pkg.yml file. In addition, it needs a mfg.yml file to specify the two constituent targets. An example of each file is shown below.

```
$ more mfgs/rb_blinky_rsa/pkg.yml
pkg.name: "mfgs/rb_blinky_rsa"
pkg.type: "mfg"
pkg.description:
pkg.author:
pkg.homepage:
```

```
$ more mfgs/rb_blinky_rsa/mfg.yml
mfg.bsp: "@apache-mynewt-core/hw/bsp/rb-nano2"
mfg.targets:
  - rb_boot:
      name: "targets/rb_boot"
      area: FLASH_AREA_BOOTLOADER
      offset: 0x0
  - rb_blinky:
      name: "targets/rb_blinky"
      area: FLASH_AREA_IMAGE_0
      offset: 0x0
mfg.meta:
  area: FLASH_AREA_BOOTLOADER
```

Build the bootloader and app images.

```
$ newt build rb_boot
$ newt create-image rb_blinky 0.0.1
```

Run the `newt mfg create` command to collect all the manufacturing snapshot files.

```
$ newt mfg create rb_blinky_rsa 0.0.1
Creating a manufacturing image from the following files:
<snip>
Generated the following files:
<snip>
```

A description of the generated files is available in the implementation's [readme](#)

8.3.10 newt new

Create a new project from a skeleton. Currently, the default skeleton is the [blinky](#) repository.

Usage:

```
newt new <project-name> [flags]
```

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

Creates a new project named `project-name` from the default skeleton [blinky](#) repository.

Examples

Usage	Explanation
<code>newt new test_project</code>	Creates a new project named <code>test_project</code> using the default skeleton from the apache/mynewt-blinky repository.

8.3.11 newt pkg

Commands for creating and manipulating packages.

Usage:

```
newt pkg [command] [flags]
```

Flags:

```
-t, --type string    Type of package to create: app, bsp, lib, sdk, unittest. (default "lib")
```

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

The `pkg` command provides subcommands to create and manage packages. The subcommands take one or two package-name arguments.

Sub-command	Explanation
copy	The copy <src-pkg> <dst-pkg> command creates the new dst-pkg package by cloning the src-pkg package.
move	The move <old-pkg> <new-pkg> command moves the old-pkg package to the new-pkg package.
new	The new <new-pkg> command creates a new package named new-pkg, from a template, in the current directory. You can create a package of type app, bsp, lib, sdk, or unittest. The default package type is lib. You use the -t flag to specify a different package type.
remove	The remove <my-pkg> command deletes the my-pkg package.

Examples

Sub-command	Usage	Explanation
copy	<code>newt pkg copy apps/btshell apps/new_btshell</code>	Copies the apps/btshell package to the apps/new_btshell.
move	<code>newt pkg move apps/slinky apps/new_slinky</code>	Moves the apps/slinky package to the apps/new_slinky package.
new	<code>newt pkg new apps/new_slinky</code>	Creates a package named apps/new_slinky of type pkg in the current directory.
new	<code>newt pkg new hw/bsp/myboard -t bsp</code>	Creates a package named hw/bsp/myboard of type bsp in the current directory.
remove	<code>newt pkg remove hw/bsp/myboard</code>	Removes the hw/bsp/myboard package.

8.3.12 newt resign-image

Sign or re-sign an existing image file.

Usage:

```
newt resign-image <image-file> [signing-key [key-id]][flags]
```

Global Flags:

<code>-h, --help</code>	Help for newt commands
<code>-j, --jobs int</code>	Number of concurrent build jobs (default 8)
<code>-l, --loglevel string</code>	Log level (default "WARN")
<code>-o, --outfile string</code>	Filename to tee output to
<code>-q, --quiet</code>	Be quiet; only display error output
<code>-s, --silent</code>	Be silent; don't output anything
<code>-v, --verbose</code>	Enable verbose output when executing commands

Description

Changes the signature of an existing image file. To sign an image, specify a .pem file for the `signing-key` and an optional `key-id`. `key-id` must be a value between 0-255. If a signing key is not specified, the command strips the current signature from the image file.

A new image header is created. The rest of the image is byte-for-byte equivalent to the original image.

Warning: The image hash will change if you change the `key-id` or the type of key used for signing.

Examples

Usage	Explanation
<code>newt resign-image bin/targets/myble/app/apps/btshell/btshell.img private.pem</code>	Signs the <code>bin/targets/myble/app/apps/btshell/btshell.img</code> file with the <code>private.pem</code> key.
<code>newt resign-image bin/targets/myble/app/apps/btshell/btshell.img</code>	Strips the current signature from <code>bin/targets/myble/app/apps/btshell/btshell.img</code> file.

8.3.13 newt run

A single command to do four steps - build a target, create-image, load image on a board, and start a debug session with the image on the board.

Note: If the version number is omitted:

- The create-image step is skipped for a bootloader target.
- You will be prompted to enter a version number for an application target.

Usage:

```
newt run <target-name> [<version>] [flags]
```

Flags:

--extraJTAGcmd	string	Extra commands to send to JTAG software
-n, --noGDB		Do not start GDB from the command line

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

Same as running `build <target-name>`, `create-image <target-name> <version>`, `load <target-name>`, and `debug <target-name>`.

Examples

Usage	Explanation
<code>newt run blink_rigado</code>	Compiles and builds the image for the app and the bsp defined for target <code>blink_rigado</code> , loads the image onto the board, and opens an active GNU gdb debugging session to run the image.
<code>newt run ble_rigado 0.1.0.0</code>	Compiles and builds the image for the app and the bsp defined for target <code>ble_rigado</code> , signs and creates the image with version number 0.1.0.0, loads the image onto the board, and opens an active GNU gdb debugging session to run the image. Note that if there is no bootloader available for a particular board/kit, a signed image creation step is not necessary.

8.3.14 newt size

Calculates the size of target components for a target.

Usage:

```
newt size <target-name> [flags]
```

Flags:

<code>-F, --flash</code>	Print FLASH statistics
<code>-R, --ram</code>	Print RAM statistics

Global Flags:

<code>-h, --help</code>	Help for newt commands
<code>-j, --jobs int</code>	Number of concurrent build jobs (default 8)
<code>-l, --loglevel string</code>	Log level (default "WARN")
<code>-o, --outfile string</code>	Filename to tee output to
<code>-q, --quiet</code>	Be quiet; only display error output
<code>-s, --silent</code>	Be silent; don't output anything
<code>-v, --verbose</code>	Enable verbose output when executing commands

Description

Displays the RAM and FLASH size of each component for the `target-name` target.

Examples

Usage	Explanation
<code>newt size blink_rigado</code>	Inspects and lists the RAM and Flash memory that each component (object files and libraries) for the <code>blink_rigado</code> target.

Example output for `newt size blink_rigado`:

```
$ newt size blink_rigado
FLASH      RAM
  9        223 *fill*
 1052       0 baselibc.a
 195      1116 blinky.a
 616      452 bmd300eval.a
  64        0 cmsis-core.a
 124       0 crt0.o
   8        0 crtio.o
  16        0 crtno.o
 277      196 full.a
  20        8 hal.a
   96     1068 libg.a
1452       0 libgcc.a
```

(continues on next page)

(continued from previous page)

```
332      28 nrf52xxx.a
3143     677 os.a

objsize
text      data      bss      dec      hex filename
7404     1172     2212    10788    2a24 /Users/<username>/dev/rigado/bin/blink_rigado/
→apps/blinky/blinky.elf
```

8.3.15 newt target

Commands to create, delete, configure and query targets.

Usage:

```
newt target [command] [flags]
```

Available Commands:

amend	Add, change, or delete values for multi-value target variables
config	View or populate a target's system configuration settings
copy	Copy target
create	Create a target
delete	Delete target
dep	View target's dependency graph
revdep	View target's reverse-dependency graph
set	Set target configuration variable
show	View target configuration variables

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

The target command provides subcommands to create, build, delete, and query targets. The subcommands take one or two `<target-name>` arguments.

Sub-command	Explanation
amend	<p>The amend command allows you to add, change, or delete values for multi-value target variables that you have set with the <code>newt target set</code> command. The format of the amend command is:</p> <pre>newt target amend <target-name> <var-name=var-value> [var-name=var-value...]</pre> <p>Specify the <code>-d</code> flag to delete values.</p> <p>The following multi-value variables can be amended: <code>aflags</code>, <code>cflags</code>, <code>lflags</code>, <code>syscfg</code>.</p> <p>The <code>var-value</code> format depends on the <code>var-name</code> as follows:</p> <ul style="list-style-type: none"> <code>aflags</code>, <code>cflags</code>, <code>lflags</code>: A string of flags, with each flag separated by a space. These variables are saved in the target's <code>pkg.yml</code> file. <code>syscfg</code>: The <code>syscfg</code> variable allows you to assign values to configuration settings in your target's <code>syscfg.yml</code> file. The format is: <p><code>syscfg=setting-name1=setting-value1[:setting-name2=setting-value2...]</code>, where <code>setting-name1</code> is a configuration setting name and <code>setting-value1</code> is the value to assign to <code>setting-name1</code>. If <code>setting-value1</code> is not specified, the setting is set to value 1. You use a <code>:</code> to delimit each setting when you amend multiple settings.</p> <p>To delete a system configuration setting, you only need to specify the setting name. For example, <code>syscfg=setting-name1:setting-name2</code> deletes configuration settings named <code>setting-name1</code> and <code>setting-name2</code>.</p>
config	<p>The config command allows you to view or populate a target's system configuration settings. A target's system configuration settings include the settings of all the packages it includes. The settings for a package are listed in the package's <code>syscfg.yml</code> file. The config command has two subcommands: <code>show</code> and <code>init</code>. The config <code>show <target-name></code> command displays the system configuration setting definitions and values for all the packages that the <code>target-name</code> target includes. The config <code>init <target-name></code> command populates the target's <code>syscfg.yml</code> file with the system configuration values for all the packages that the <code>target-name</code> target includes.</p>
copy	<p>The copy <code><src-target> <dst-target></code> command creates a new target named <code>dst-target</code> by cloning the <code>src-target</code> target.</p>
create	<p>The create <code><target-name></code> command creates an empty target named <code>target-name</code>. It creates the <code>targets/target-name</code> directory and the skeleton <code>pkg.yml</code> and <code>target.yml</code> files in the directory.</p>
delete	<p>The delete <code><target-name></code> command deletes the description for the <code>target-name</code> target. It deletes the '<code>targets/target-name</code>' directory. It does not delete the '<code>bin/targets/target-name</code>' directory where the build artifacts are stored. If you want to delete the build artifacts, run the <code>newt clean <target-name></code> command before deleting the target.</p>
dep	<p>The dep <code><target-name></code> command displays a dependency tree for the packages that the <code>target-name</code> target includes. It shows each package followed by the list of libraries or packages that it depends on.</p>
revdep	<p>The revdep <code><target-name></code> command displays the reverse dependency tree for the packages that the <code>target-name</code> target includes. It shows each package followed by the list of libraries or packages that depend on it.</p>

continues on next page

Table 1 – continued from previous page

Sub-command	Explanation
set	<p>The set <target-name> <var-name=var-value> [var-name=var-value...] command sets variables (attributes) for the <target-name> target. The set command overwrites your current variable values. The valid var-name values are: app, bsp, loader, build_profile, cflags, lflags, aflags, syscfg.</p> <p>The var-value format depends on the var-name as follows:</p> <p>app, bsp, loader: @<source-path>, where source-path is the directory containing the application or bsp source. These variables are stored in the target's target.yml file. For a simulated target, e.g. for software testing purposes, set bsp to @apache-mynewt-core/hw/bsp/native.</p> <p>build_profile: optimized or debug</p> <p>aflags, cflags, lflags: A string of flags, with each flag separated by a space. These variables are saved in the target's pkg.yml file.</p> <p>syscfg: The syscfg variable allows you to assign values to configuration settings in your target's syscfg.yml file. The format is: syscfg=setting-name1=setting-value1[:setting-name2=setting-value2...], where setting-name1 is a configuration setting name and setting-value1 is the value to assign to setting-name1. If setting-value1 is not specified, the setting is set to value 1. You use a : to delimit each setting when you set multiple settings. You can specify var-name= or var-name="" to unset a variable value. Warning: For multi-value variables, the command overrides all existing values. Use the newt target amend command to change or add new values for a multi-value variable after you have set the variable value. The multi-value variables are: aflags, cflags, lflags, and syscfg To display all the existing values for a target variable (attribute), you can run the newt vals <variable-name> command. For example, newt vals app displays the valid values available for the variable app for any target.</p>
show	The show [target-name] command shows the values of the variables (attributes) for the target-name target. When target-name is not specified, the command shows the variables for all the targets that are defined for your project.

Examples

Sub-command	Usage	Explanation
amend	<code>newt target amend myble syscfg=CONFIG_NEWTMGR=0 cflags="-DTEST"</code>	Changes (or adds) the CONFIG_NEWTMGR variable to value 0 in the syscfg.yml file and adds the -DTEST flag to pkg.cflags in the pkg.yml file for the myble target. Other syscfg setting values and cflags values are not changed.
amend	<code>newt target amend myble -d syscfg=LOG_LEVEL:CONFIG_NEWTMGR cflags="-DTEST"</code>	Deletes the LOG_LEVEL and CONFIG_NEWTMGR settings from the syscfg.yml file and the -DTEST flag from pkg.cflags for the myble target. Other syscfg setting values and cflags values are not changed.
config show	<code>newt target config show rb_blinky</code>	Shows the system configuration settings for all the packages that the rb_blinky target includes.
config init	<code>newt target config init my_blinky</code>	Creates and populates the my_blinky target's syscfg.yml file with the system configuration setting values from all the packages that the my_blinky target includes.
copy	<code>newt target copy rb_blinky rb_btshell</code>	Creates the rb_btshell target by cloning the rb_blinky target.
create	<code>newt target create my_new_target</code>	Creates the my_new_target target. It creates the targets/my_new_target directory and creates the skeleton pkg.yml and target.yml files in the directory.
delete	<code>newt target delete rb_btshell</code>	Deletes the rb_btshell target. It deletes the targets/rb_btshell directory.
dep	<code>newt target dep myble</code>	Displays the dependency tree of all the package dependencies for the myble target. It lists each package followed by a list of packages it depends on.
revdep	<code>newt target revdep myble</code>	Displays the reverse dependency tree of all the package dependencies for the myble target. It lists each package followed by a list of packages that depend on it.
set	<code>newt target set myble app=@apache-mynewt-core/apps/ btshell</code>	Use btshell as the application to build for the myble target.
set	<code>newt target set myble cflags="-DNDEBUG -Werror"</code>	Set pkg.cflags variable with -DNDEBUG -Werror in the myble target's pkg.yml file..
set	<code>newt target set myble syscfg=LOG_NEWTMGR=0:CONFIG_NEWTMGR</code>	Sets the syscfg.vals variable in the myble target's syscfg.yml file with the setting values: LOG_NEWTMGR: 0 and CONFIG_NEWTMGR: 1. CONFIG_NEWTMGR is set to 1 because a value is not specified.
set	<code>newt target set myble cflags=</code>	Unsets the pkg.cflags variable in the myble target's pkg.yml file.
show	<code>newt target show myble</code>	Shows all variable settings for the myble target, i.e. the values that app, bsp, build_profile, cflags, aflags, ldflags, syscfg variables are set to. Note that not all variables have to be set for a target.
show	<code>newt target show</code>	Shows all the variable settings for all the targets defined for the project.

8.3.16 newt test

Execute unit tests for one or more packages.

Usage:

```
newt test <package-name> [package-names...] | all [flags]
```

Flags:

```
-e, --exclude string    Comma separated list of packages to exclude
```

Global Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

Description

Executes unit tests for one or more packages. You specify a list of packages, separated by space, to test multiple packages in the same command, or specify `all` to test all packages. When you use the `all` option, you may use the `-e` flag followed by a comma separated list of packages to exclude from the test.

Examples

Usage	Explanation
<code>newt test @apache-mynewt-core/kernel/os</code>	Tests the <code>kernel/os</code> package in the <code>apache-mynewt-core</code> repository.
<code>newt test kernel/os encoding/json</code>	Tests the <code>kernel/os</code> and <code>encoding/json</code> packages in the current repository.
<code>newt test all</code>	Tests all packages.
<code>newt test all -e net/oic, encoding/json</code>	Tests all packages except for the <code>net/oic</code> and the <code>encoding/json</code> packages.

8.3.17 newt upgrade

Upgrade project dependencies.

Usage:

```
newt upgrade [flags]
```

Flags:

```
-f, --force    Force upgrade of the repositories to latest state in project.yml
```

Global Flags:

<code>-h, --help</code>	Help for newt commands
<code>-j, --jobs int</code>	Number of concurrent build jobs (default 8)
<code>-l, --loglevel string</code>	Log level (default "WARN")
<code>-o, --outfile string</code>	Filename to tee output to
<code>-q, --quiet</code>	Be quiet; only display error output
<code>-s, --silent</code>	Be silent; don't output anything
<code>-v, --verbose</code>	Enable verbose output when executing commands

Description

Upgrades your project and package dependencies. If you have changed the project.yml description for the project, you need to run this command to update all the package dependencies.

8.3.18 newt vals

Display valid values for the specified element type(s).

Usage:

```
newt vals <element-type> [<element-types...>] [flags]
```

Global Flags:

<code>-h, --help</code>	Help for newt commands
<code>-j, --jobs int</code>	Number of concurrent build jobs (default 8)
<code>-l, --loglevel string</code>	Log level (default "WARN")
<code>-o, --outfile string</code>	Filename to tee output to
<code>-q, --quiet</code>	Be quiet; only display error output
<code>-s, --silent</code>	Be silent; don't output anything
<code>-v, --verbose</code>	Enable verbose output when executing commands

Description

Displays valid values for the specified element type(s). You must set valid values for one or more elements when you define a package or a target. Valid element types are:

- api
- app
- bsp
- build_profile
- compiler
- lib
- sdk
- target

Examples

Usage	Explanation
<code>newt vals api</code>	Shows the possible values for APIs a package may specify as required. For example, the <code>pkg.yml</code> for <code>adc</code> specifies that it requires the <code>api</code> named <code>ADC_HW_IMPL</code> , one of the values listed by the command.

Example output for `newt vals bsp`:

This lists all possible values that may be assigned to a target's `bsp` attribute.

```
$ newt vals bsp
bsp names:
@apache-mynewt-core/hw/bsp/arduino_primo_nrf52
@apache-mynewt-core/hw/bsp/bmd300eval
@apache-mynewt-core/hw/bsp/ci40
@apache-mynewt-core/hw/bsp/frdm-k64f
@apache-mynewt-core/hw/bsp/native
@apache-mynewt-core/hw/bsp/nrf51-arduino_101
@apache-mynewt-core/hw/bsp/nrf51-blenano
@apache-mynewt-core/hw/bsp/nrf51dk
@apache-mynewt-core/hw/bsp/nrf51dk-16kbram
@apache-mynewt-core/hw/bsp/nrf52dk
@apache-mynewt-core/hw/bsp/nucleo-f401re
@apache-mynewt-core/hw/bsp/olimex_stm32-e407_devboard
@apache-mynewt-core/hw/bsp/rb-nano2
@apache-mynewt-core/hw/bsp/stm32f4discovery
$ newt target set sample_target bsp=@apache-mynewt-core/hw/bsp/rb-nano2
```

Obviously, this output will grow as more board support packages are added for new boards and MCUs.

8.3.19 newt version

Display the version of the newt tool you have installed

Usage:

```
newt version [flags]
```

Global Flags:

<code>-h, --help</code>	Help for newt commands
<code>-j, --jobs int</code>	Number of concurrent build jobs (default 8)
<code>-l, --loglevel string</code>	Log level (default "WARN")
<code>-o, --outfile string</code>	Filename to tee output to
<code>-q, --quiet</code>	Be quiet; only display error output
<code>-s, --silent</code>	Be silent; don't output anything
<code>-v, --verbose</code>	Enable verbose output when executing commands

Examples

Usage	Explanation
<code>newt version</code>	Displays the version of the newt tool you have installed

8.4 Install

8.4.1 Installing Newt on macOS

Newt is supported on macOS 64 bit platforms and has been tested on macOS 10.12 and higher.

This page shows you how to:

- Upgrade to or install the latest release version of newt.
- Install the latest newt from the master branch (unstable).

See [Installing Previous Releases of Newt](#) to install an earlier version of newt.

Note: If you would like to contribute to the newt tool, see [Setting Up Go Environment to Contribute to Newt and Newtmgr Tools](#).

Installing Homebrew

If you do not have Homebrew installed, run the following command. You will be prompted for your sudo password.

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

You can also extract (or `git clone`) Homebrew and install it to `/usr/local`.

Adding the Mynewt Homebrew Tap

If this is your first time installing newt, add the **JuulLabs-OSS/mynewt** tap:

```
$ brew tap JuulLabs-OSS/mynewt
==> Tapping juullabs-oss/mynewt
Cloning into '/usr/local/Homebrew/Library/Taps/juullabs-oss/homebrew-mynewt'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 16 (delta 12), reused 3 (delta 3), pack-reused 0
Unpacking objects: 100% (16/16), done.
Tapped 14 formulae (52 files, 51.9KB).
$ brew update
```

Upgrading to or Installing the Latest Release Version

Perform the following to upgrade or install the latest release version of newt.

Note: The homebrew tap used to live under `runtim eco/mynewt`, and although updating should still work, if the tap stops pulling in the latest releases, please try removing the old tap and adding the new one:

```
$ brew untap runtim eco/mynewt
$ brew tap JuulLabs-OSS/mynewt
```

Upgrading to the Latest Release Version of Newt

If you previously installed newt using brew, run the following commands to upgrade to newt latest:

```
$ brew update
$ brew upgrade mynewt-newt
==> Upgrading 1 outdated package, with result:
juullabs-oss/mynewt/mynewt-newt 1.5.0
==> Upgrading juullabs-oss/mynewt/mynewt-newt
==> Downloading https://github.com/juullabs-oss/binary-releases/raw/master/mynewt-newt-
→ tools_1.5.0/mynewt-newt-1.5.0.sierra.bottle.tar.gz
==> Downloading from https://raw.githubusercontent.com/juullabs-oss/binary-releases/
→ master/mynewt-newt-tools_1.5.0/mynewt-newt-1.5.0.sierra.bottle.tar.gz
#####
==> Pouring mynewt-newt-1.5.0.sierra.bottle.tar.gz
/usr/local/Cellar/mynewt-newt/1.5.0: 3 files, 8.1MB
```

Installing the Latest Release Version of Newt

Run the following command to install the latest release version of newt:

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

Notes: Homebrew bottles for newt are available for macOS Sierra. If you are running an earlier version of macOS, the installation will install the latest version of Go and compile newt locally.

Checking the Installed Version

Check that you are using the installed version of newt:

```
$ which newt
/usr/local/bin/newt
$ newt version
Apache Newt version: 1.5.0
```

Note: If you previously built newt from source and the output of `which newt` shows “`$GOPATH/bin/newt`”, you will need to move “`$GOPATH/bin`” after “`/usr/local/bin`” for your PATH in `~/.bash_profile`, and source `~/.bash_profile`.

Get information about newt:

```
$ newt help
Newt allows you to create your own embedded application based on the Mynewt
operating system. Newt provides both build and package management in a single
tool, which allows you to compose an embedded application, and set of
projects, and then build the necessary artifacts from those projects. For more
information on the Mynewt operating system, please visit
https://mynewt.apache.org/.
```

Please use the `newt help` command, and specify the name of the command you want
help for, for help on how to use a specific command

Usage:

```
newt [flags]
newt [command]
```

Examples:

```
newt
newt help [<command-name>]
For help on <command-name>. If not specified, print this message.
```

Available Commands:

(continues on next page)

(continued from previous page)

build	Build one or more targets
clean	Delete build artifacts for one or more targets
create-image	Add image header to target binary
debug	Open debugger session to target
help	Help about any command
info	Show project info
install	Install project dependencies
load	Load built target to board
mfg	Manufacturing flash image commands
new	Create a new project
pkg	Create and manage packages in the current workspace
resign-image	Re-sign an image.
run	build/create-image/download/debug <target>
size	Size of target components
sync	Synchronize project dependencies
target	Commands to create, delete, configure, and query targets
test	Executes unit tests for one or more packages
upgrade	Upgrade project dependencies
vals	Display valid values for the specified element type(s)
version	Display the Newt version number

Flags:

-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 4)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

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

Installing Newt from the Master Branch

We recommend that you use the latest release version of newt. If you would like to use the master branch with the latest updates, you can install newt from the HEAD of the master branch.

Notes:

- The master branch may be unstable.
- This installation will install the latest version of Go on your computer, if it is not installed, and compile newt locally.

If you previously installed newt using brew, unlink the current version. Depending on your previous newt version you may have to also uninstall newt.

```
$ brew unlink mynewt-newt
$ brew uninstall mynewt-newt
```

Install the latest unstable version of newt from the master branch:

```
$ brew install mynewt-newt --HEAD
```

To switch back to the latest stable release version of newt, you can run:

```
$ brew switch mynewt-newt 1.5.0
```

8.4.2 Installing Newt on Linux

You can install the latest release (1.9.0) of the newt tool from a Debian binary package (amd64). You can also download and build the latest release version of newt from source.

This page shows you how to:

1. Set up your computer to download Debian binary packages from the JuulLabs-OSS APT repository.
Note: The key for signing the repository has changed. If you set up your computer before release 1.1.0, you will need to download and import the public key again.
2. Install the latest release version of newt from a Debian binary package. You can use apt-get to install the package or manually download and install the Debian binary package.
3. Download, build, and install the latest release version of newt from source.

If you are installing on an amd64 platform, we recommend that you install from the binary package.

See [Installing Previous Releases of Newt](#) to install an earlier version of newt.

Note: We have tested the newt tool binary and apt-get install from the JuulLabs-OSS APT repository for Ubuntu version 1704. Earlier Ubuntu versions (for example: Ubuntu 14) may have incompatibility with the repository. You can manually download and install the Debian binary package.

Note: See [Contributing to Newt or Newtmgr Tools](#) if you want to:

- Use the newt tool with the latest updates from the master branch. The master branch may be unstable and we recommend that you use the latest stable release version.
- Contribute to the newt tool.

Setting Up Your Computer to use apt-get to Install the Package

The newt Debian packages are stored in a private APT repository on <https://github.com/JuulLabs-OSS/debian-mynewt>. To use apt-get, you must set up the following on your computer to retrieve packages from the repository:

Note: You only need to perform this setup once on your computer. However, if you previously downloaded and imported the public key for the JuulLabs-OSS APT repository, you will need to perform step 2 again as the key has changed.

1. Download the public key for the JuulLabs-OSS APT repository and import the key into the apt keychain.
2. Add the repository for the binary and source packages to the apt source list.

Download the public key for the JuulLabs-OSS apt repo (**Note:** There is a - after apt-key add):

```
$ wget -qO - https://raw.githubusercontent.com/JuulLabs-OSS/debian-mynewt/master/mynewt.gpg.key | sudo apt-key add -
```

Add the repository for the binary and source packages to the `mynewt.list` apt source list file:

```
$ sudo tee /etc/apt/sources.list.d/mynewt.list <<EOF
deb https://raw.githubusercontent.com/JuulLabs-OSS/debian-mynewt/master latest main
EOF
```

Note: Previously the repository lived under `runtim eco/debian-mynewt`, and although updating should remain working, if it stops pulling in the latest releases, please try updating `/etc/apt/sources.list.d/mynewt.list` and substitute `runtim eco` by `JuulLabs-OSS`.

Update the available packages:

```
$ sudo apt-get update
```

Note: If you are not using Ubuntu version 1704, you may see the following errors. We have provided instructions on how to manually download and install the binary package.

```
W: Failed to fetch https://raw.githubusercontent.com/JuulLabs-OSS/debian-mynewt/master/  ↪dists/latest/main/source/Sources  HttpError404
```

Installing the Latest Release of Newt from a Binary Package

You can use either `apt-get` to install the package, or manually download and install the Debian binary package.

Method 1: Using `apt-get` to Upgrade or to Install

Run the following commands to upgrade or install the latest version of newt:

```
$ sudo apt-get update  
$ sudo apt-get install newt
```

Note: If you encounter build errors (such as missing `sys/mman.h`), please make sure you have a 32-bit glibc:

```
$ sudo apt-get install gcc-multilib
```

Method 2: Downloading and Installing the Debian Package Manually

Download and install the package manually.

```
$ wget https://raw.githubusercontent.com/JuulLabs-OSS/binary-releases/master/mynewt-newt-  ↪tools_1.9.0/newt_1.9.0-1_amd64.deb  
$ sudo dpkg -i newt_1.9.0-1_amd64.deb
```

See [Checking the Installed Version of Newt](#) to verify that you are using the installed version of newt.

Installing the Latest Release of Newt from a Source Package

If you are running Linux on a different architecture, you can build and install the latest release version of newt from source.

The newt tool is written in Go (<https://golang.org/>). In order to build Apache Mynewt, you must have Go 1.12 or later installed on your system. Please visit the Golang website for more information on installing Go (<https://golang.org/>).

1. Download and unpack the newt source:

```
$ wget -P /tmp https://github.com/apache/mynewt-newt/archive/mynewt_1_9_0_tag.tar.gz  
$ tar -xzf /tmp/mynewt_1_9_0_tag.tar.gz
```

- Run the build.sh to build the newt tool.

```
$ cd mynewt-newt-mynewt_1_9_0_tag
$ ./build.sh
$ rm /tmp/mynewt_1_9_0_tag.tar.gz
```

- You should see the newt/newt executable. Move the executable to a bin directory in your PATH:

- If you previously built newt from the master branch, you can move the binary to your \$GOPATH/bin directory.

```
$ mv newt/newt $GOPATH/bin
```

- If you are installing newt for the first time and do not have a Go workspace set up, you can move the binary to /usr/bin or a directory in your PATH:

```
$ mv newt/newt /usr/bin
```

Checking the Installed Version of Newt

- Check which newt you are using and that the version is the latest release version.

```
$ which newt
/usr/bin/newt
$ newt version
Apache Newt version: 1.9.0
```

- Get information about newt:

```
$ newt
Newt allows you to create your own embedded application based on the Mynewt
operating system. Newt provides both build and package management in a single
tool, which allows you to compose an embedded application, and set of
projects, and then build the necessary artifacts from those projects. For more
information on the Mynewt operating system, please visit
https://mynewt.apache.org/.
```

Please use the newt help command, and specify the name of the command you want help for, for help on how to use a specific command

Usage:

```
newt [flags]
newt [command]
```

Examples:

```
newt
newt help [<command-name>]
For help on <command-name>. If not specified, print this message.
```

Available Commands:

apropos	Search manual page names and descriptions
build	Build one or more targets
clean	Delete build artifacts for one or more targets

(continues on next page)

(continued from previous page)

```

create-image Add image header to target binary
debug      Open debugger session to target
docs       Project documentation generation commands
help       Help about any command
info       Show project info
load       Load built target to board
man        Browse the man-page for given argument
man-build  Build man pages
mfg       Manufacturing flash image commands
new        Create a new project
pkg        Create and manage packages in the current workspace
resign-image Obsolete
run        build/create-image/download/debug <target>
size       Size of target components
target     Commands to create, delete, configure, and query targets
test       Executes unit tests for one or more packages
upgrade   Upgrade project dependencies
vals       Display valid values for the specified element type(s)
version   Display the Newt version number

```

Flags:

--escape	Apply Windows escapes to shell commands
-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 4)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

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

8.4.3 Installing Newt on Windows

You can develop and build Mynewt OS applications for your target boards on the Windows platform. This guide shows you how to install the latest release version of newt from binary or from source. The tool is written in Go (golang).

In Windows, we use MinGW as the development environment to build and run Mynewt OS applications for target boards. MinGW runs the bash shell and provides a Unix-like environment. This provides a uniform way to build Mynewt OS applications. The Mynewt documentation and tutorials use Unix commands and you can use the same Unix commands on MinGW to follow the tutorials. The documentation will note any commands or behaviors that are specific to Windows.

This guide shows you how to perform the following:

1. Install MSYS2/MinGW.
2. Install Git.
3. Install latest release (1.8.0) of newt from binary.
4. Install go for building newt from source.
5. Install latest release of newt from source.

See [Installing Previous Releases of Newt](#) to install an earlier version of newt. You still need to set up your MinGW development environment.

Note: If you would like to contribute to the newt tool, see [Contributing to Newt or Newtmg Tools](#).

Installing MSYS2/MinGW

MSYS2/MinGW provides a bash shell and tools to build applications that run on Windows. It includes three subsystems:

- MSYS2 toolchain to build POSIX applications that run on Windows.
- MinGW32 toolchains to build 32 bit native Windows applications.
- MinGW64 toolchains to build 64 bit native Windows applications.

The subsystems run the bash shell and provide a Unix-like environment. You can also run Windows applications from the shell. We will use the MinGW subsystem.

Note: You can skip this installation step if you already have MinGW installed (from an earlier MSYS2/MinGW or Git Bash installation), but you must list the **bin** path for your installation in your Windows Path. For example: if you installed MSYS2/MinGW in the **C:\msys64** directory, add **C:\msys64\usr\bin** to your Windows Path. If you are using Windows 10 WSL, ensure that you use the **C:\msys64\usr\bin\bash.exe** and not the Windows 10 WSL bash.

To install and setup MSYS2 and MinGW:

1. Download and run the [MSYS2 installer](#). Select the 64 bit version if you are running on a 64 bit platform. Follow the prompts and check the **Run MSYS2 now** checkbox on the **Installation Complete** dialog.
2. In the MSYS2 terminal, run the `pacman -Syuu` command. If you get a message to run the update again, close the terminal and run the `pacman -Syuu` command in a new terminal.

To start a new MSYS2 terminal, select the “MSYS2 MSYS” application from the Windows start menu.

3. Add a new user variable named **MSYS2_PATH_TYPE** and set the value to **inherit** in your Windows environment. This enables the MSYS2 and MinGW bash to inherit your Windows user **Path** values.

To add the variable, select properties for your computer > Advanced system settings > Environment Variables > New

4. Add the MinGW **bin** path to your Windows Path. For example: if you install MSYS2/MinGW in the **C:\msys64** directory, add **C:\msys64\usr\bin** to your Windows Path.

Note: If you are using Windows 10 WSL, ensure that you use the **C:\msys64\usr\bin\bash.exe** and not the Windows 10 WSL bash.

5. Run the `pacman -Su vim` command to install the vim editor.

Note: You can also use a Windows editor. You can access your files from the **C:<msys-install-folder>\home\<username>** folder, where **msys-install-folder** is the folder you installed MSYS2 in. For example, if you installed MSYS2 in the **msys64** folder, your files are stored in **C:\msys64\home\<username>**

6. Run the `pacman -Su tar` command to install the tar tool.

You will need to start a MinGW terminal to run the commands specified in the Mynewt documentation and tutorials. To start a MinGW terminal, select the “MSYS2 Mingw” application from the start Menu (you can use either MinGW32 or MinGW64). In Windows, we use the MinGW subsystem to build Mynewt tools and applications.

Installing Git

Git can be installed as MinGW package from MinGW terminal:

```
$ pacman -S git
```

Alternatively download and install Git for Windows.

Installing the Latest Release of the Newt Tool from Binary

You can install the latest release of newt from binary. It has been tested on Windows 10 64 bit platform.

1. Start a MinGW terminal.
2. Download the newt binary tar file `apache-mynewt-newt-bin-windows-1.8.0.tgz`.
3. Extract the file:
 - If you previously built newt from the master branch, you can extract the file into your \$GOPATH/bin directory. Note: This overwrites the current newt.exe in the directory and assumes that you are using \$GOPATH/bin for your Go applications.

```
$ tar -xzf apache-mynewt-newt-bin-windows-1.8.0.tgz -C $GOPATH/bin --strip-  
-components=1 apache-mynewt-newt-bin-windows-1.8.0/newt.exe
```

- If you are installing newt for the first time and do not have a Go workspace setup, you can extract into /usr/bin directory:

```
$ tar -xzf apache-mynewt-newt-bin-windows-1.8.0.tgz -C /usr/bin --strip-  
-components=1 apache-mynewt-newt-bin-windows-1.8.0/newt.exe
```

4. Verify the installed version of newt. See [Checking the Installed Version](#).

Installing Go

Newt requires **Go** version **1.13** or higher. If you do not have Go installed, it can be installed from MinGW package repository.

1. Open a MinWG terminal.
2. Install go package.

```
$ pacman -S mingw-w64-x86_64-go
```

Alternatively newest version of Go from [golang.org](#) can be used. To download and install a newer version of Go.

Installing the Latest Release of Newt From Source

If you have an older version of Windows or a 32 bit platform, you can build and install the latest release version of newt from source.

1. Start a MinGw terminal.
2. Download and unpack the newt source:

```
$ wget -P /tmp https://github.com/apache/mynewt-newt/archive/mynewt_1_8_0_tag.tar.gz
$ tar -xzf /tmp/mynewt_1_8_0_tag.tar.gz
```

3. Run the build.sh to build the newt tool.

```
$ cd mynewt-newt-mynewt_1_8_0_tag
$ MSYS=winsymlinks:nativestrict ./build.sh
$ rm /tmp/mynewt_1_8_0_tag.tar.gz
```

4. You should see the `newt/newt.exe` executable. Move the executable to a bin directory in your PATH:

- If you previously built newt from the master branch, you can move the executable to the `$GOPATH/bin` directory.

```
$ mv newt/newt.exe $GOPATH/bin
```

- If you are installing newt for the first time and do not have a Go workspace set up, you can move the executable to `/usr/bin` or a directory in your PATH:

```
$ mv newt/newt.exe /usr/bin
```

Checking the Installed Version

1. Check the version of newt:

```
$ newt version
Apache Newt 1.8.0 / ab96a8a-dirty / 2020-03-18_23:25
```

2. Get information about newt:

```
$ newt help
```

Newt allows you to create your own embedded application based on the Mynewt operating system. Newt provides both build and package management in a single tool, which allows you to compose an embedded application, and set of projects, and then build the necessary artifacts from those projects. For more information on the Mynewt operating system, please visit <https://mynewt.apache.org/>.

Please use the `newt help` command, and specify the name of the command you want help for, for help on how to use a specific command

Usage:

```
newt [flags]
newt [command]
```

Examples:

(continues on next page)

(continued from previous page)

```
newt
newt help [<command-name>]
    For help on <command-name>. If not specified, print this message.
```

Available Commands:

apropos	Search manual page names and descriptions
build	Build one or more targets
clean	Delete build artifacts for one or more targets
create-image	Add image header to target binary
debug	Open debugger session to target
docs	Project documentation generation commands
help	Help about any command
info	Show project info
load	Load built target to board
man	Browse the man-page for given argument
man-build	Build man pages
mfg	Manufacturing flash image commands
new	Create a new project
pkg	Create and manage packages in the current workspace
resign-image	Obsolete
run	build/create-image/download/debug <target>
size	Size of target components
target	Commands to create, delete, configure, and query targets
test	Executes unit tests for one or more packages
upgrade	Upgrade project dependencies
vals	Display valid values for the specified element type(s)
version	Display the Newt version number

Flags:

--escape	Apply Windows escapes to shell commands (default true)
-h, --help	Help for newt commands
-j, --jobs int	Number of concurrent build jobs (default 8)
-l, --loglevel string	Log level (default "WARN")
-o, --outfile string	Filename to tee output to
-q, --quiet	Be quiet; only display error output
-s, --silent	Be silent; don't output anything
-v, --verbose	Enable verbose output when executing commands

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

8.4.4 Installing Previous Releases of Newt

This page shows you how to install previous releases of newt for macOS, Linux, and Windows.

macOS

You can install previous releases of newt using `mynewt-newt@X.Y` Homebrew formulas, where X.Y is a version number.

For example, if you want to install newt 1.0, run the following commands:

```
$ brew update
$ brew install mynewt-newt@1.0
```

Note: This is a keg-only installation. newt 1.0 is installed in `/usr/local/Cellar/mynewt-newt@1.0/1.0.0/bin` but not symlinked into `/usr/local/bin`.

If you need this version of newt first in your PATH, run the following commands:

```
$ echo 'export PATH=/usr/local/Cellar/mynewt-newt@1.0/1.0.0/bin:$PATH' >> ~/.bash_profile
$ source ~/.bash_profile
```

You can also manually symlink into `/usr/local/bin` as follows:

1. Unlink newt if you have the latest version of newt installed:

```
$ brew unlink mynewt-newt
```

2. Link `mynewt-newt@1.0` into `/usr/local/bin`:

```
$ brew link -f mynewt-newt@1.0
```

Linux

1. Download the binary:

Version	Download
1.0.0	newt_1.0.0-1_amd64.deb
1.1.0	newt_1.1.0-1_amd64.deb
1.3.0	newt_1.3.0-1_amd64.deb
1.4.0	newt_1.4.0-1_amd64.deb
1.4.1	newt_1.4.1-1_amd64.deb

2. Run the `sudo apt-get remove newt` command to remove the current installation.
3. Install the package. For example, run `sudo dpkg -i newt_1.0.0-1_amd64.deb` to install newt 1.0.0

Windows

1. Download the binary:

Version	Download
1.1.0	newt_1_1_0_windows_amd64.tar.gz
1.3.0	newt_3_1_0_windows_amd64.tar.gz

2. Extract the file:

- If you previously built newt from the master branch, you can extract the file into your \$GOPATH/bin directory. Note: This overwrites the current newt.exe in the directory and assumes that you are using \$GOPATH/bin for your Go applications.

```
tar -xzf newt_1_1_0_windows_amd64.tar.gz -C $GOPATH/bin
```

- If you are installing newt for the first time and do not have a Go workspace setup, you can extract into /usr/bin directory:

```
tar -xzf newt_1_1_0_windows_amd64.tar.gz -C /usr/bin
```

NEWT MANAGER GUIDE

Newt Manager (newtmgr) is the application tool that enables a user to communicate with and manage remote devices running the Mynewt OS. It uses a connection profile to establish a connection with a device and sends command requests to the device. The tool follows the same command structure as the [newt tool](#).

9.1 Command List

9.1.1 Overview

The following are the high-level newtmgr commands. Some of these commands have subcommands. You can use the -h flag to get help for each command. See the documentation for each command in this guide if you need more information and examples.

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)
```

9.1.2 newtmgr config

Read and write config values on a device.

Usage:

```
newtmgr config <var-name> [var-value] -c <conn_profile> [flags]
```

Global 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)

Description

Reads and sets the value for the `var-name` config variable on a device. Specify a `var-value` to set the value for the `var-name` variable. Newtmgr uses the `conn_profile` connection profile to connect to the device.

Examples

Usage	Explanation
<code>newtmgr config myvar -c profile01</code>	Reads the <code>myvar</code> config variable value from a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
<code>newtmgr config myvar 2 -c profile01</code>	Sets the <code>myvar</code> config variable to the value 2 on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.

9.1.3 newtmgr conn

Manage newtmgr connection profiles.

- *Usage:*
- *Global Flags:*
- *Description*
 - *Add Sub-Command*
 - *Delete Sub-Command*
 - *Show Sub-Command*
- *Examples*

Usage:

```
newtmgr conn [command] [flags]
```

Global Flags:

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

Description

The `conn` command provides subcommands to add, delete, and view connection profiles. A connection profile specifies information on how to connect and communicate with a remote device. Newtmgr commands use the information from a connection profile to send newtmgr requests to remote devices.

Add Sub-Command

The `newtmgr conn add <conn_profile> <var-name=value ...>` command creates a connection profile named `conn_profile`. The command requires the `conn_profile` name and a list of, space separated, `var-name=value` pairs.

The var-names are: `type`, and `connstring`. The valid values for each var-name parameter are:

- **type:** The connection type. Valid values are:
 - **serial:** Newtmgr protocol over a serial connection.
 - **oic_serial:** OIC protocol over a serial connection.
 - **udp:** Newtmgr protocol over UDP.
 - **oic_udp:** OIC protocol over UDP.
 - **ble:** Newtmgr protocol over BLE. This type uses native OS BLE support
 - **oic_ble:** OIC protocol over BLE. This type uses native OS BLE support.
 - **bhd:** Newtmgr protocol over BLE. This type uses the blehostd implementation.
 - **oic_bhd:** OIC protocol over BLE. This type uses the blehostd implementation.

Note: newtmgr does not support BLE on Windows.

- **connstring:** The physical or virtual address for the connection. The format of the `connstring` value depends on the connection `type` value as follows:
 - **serial** and **oic_serial**: A quoted string with two, comma separated, `attribute=value` pairs. The attribute names and value format for each attribute are:
 - * **dev:** (Required) The name of the serial port to use. For example: **/dev/ttyUSB0** on a Linux platform or **COM1** on a Windows platform .
 - * **baud:** (Optional) A number that specifies the baud rate for the connection. Defaults to **115200** if the attribute is not specified.

Example: `connstring="dev=/dev/ttyUSB0,baud=9600"` **Note:** The 1.0 format, which only requires a serial port name, is still supported. For example, `connstring=/dev/ttyUSB0`.

– **udp** and **oic_udp**: The peer ip address and port number that the newtmgr or oicmgr on the remote device is listening on. It must be of the form: [`<ip-address>`]:`<port-number>`.

– **ble** and **oic_ble**: The format is a quoted string of, comma separated, `attribute=value` pairs. The attribute names and the value for each attribute are:

- * **peer_name**: A string that specifies the name the peer BLE device advertises. **Note:** If this attribute is specified, you do not need to specify a value for the **peer_id** attribute.

- * **peer_id**: The peer BLE device address or UUID. The format depends on the OS that the newtmgr tool is running on:

Linux: 6 byte BLE address. Each byte must be a hexadecimal number and separated by a colon.

MacOS: 128 bit UUID.

Note: This value is only used when a peer name is not specified for the connection profile or with the `--name` flag option.

- * **ctlr_name**: (Optional) Controller name. This value depends on the OS that the newtmgr tool is running on.

Notes:

- * You must specify `connstring=" "` if you do not specify any attribute values.

- * You can use the `--name` flag to specify a device name when you issue a newtmgr command that communicates with a BLE device. You can use this flag to override or in lieu of specifying a **peer_name** or **peer_id** attribute in the connection profile.

– **bhd** and **oic_bhd**: The format is a quoted string of, comma separated, `attribute=value` pairs. The attribute names and the value format for each attribute are:

- * **peer_name**: A string that specifies the name the peer BLE device advertises. **Note:** If this attribute is specified, you do not need to specify values for the **peer_addr** and **peer_addr_type** attributes.

- * **peer_addr**: A 6 byte peer BLE device address. Each byte must be a hexadecimal number and separated by a colon. You must also specify a **peer_addr_type** value for the device address. **Note:** This value is only used when a peer name is not specified for the connection profile or with the `--name` flag option.

- * **peer_addr_type**: The peer address type. Valid values are:

- **public**: Public address assigned by the manufacturer.

- **random**: Static random address.

- **rpa_pub**: Resolvable Private Address with public identity address.

- **rpa_rnd**: Resolvable Private Address with static random identity address.

Note: This value is only used when a peer name is not specified for the connection profile or with the `--name` flag option.

- * **own_addr_type**: (Optional) The address type of the BLE controller for the host that the newtmgr tool is running on. See the **peer_addr_type** attribute for valid values. Defaults to **random**.

- * **ctlr_path**: The path of the port that is used to connect the BLE controller to the host that the newtmgr tool is running on.

Note: You can use the `--name` flag to specify a device name when you issue a `newtmgr` command that communicates with a BLE device. You can use this flag to override or in lieu of specifying a `peer_name` or `peer_addr` attribute in the connection profile.

Delete Sub-Command

The `newtmgr conn delete <conn_profile>` command deletes the `conn_profile` connection profile.

Show Sub-Command

The `newtmgr conn show [<conn_profile>]` command shows the information for the `conn_profile` connection profile. It shows information for all the connection profiles if `conn_profile` is not specified.

Examples

Sub-command	Usage	Explanation
add	<code>newtmgr conn add myserial02 type=oic_serial connstring=/dev/ttys002</code>	Creates a connection profile, named <code>myserial02</code> , to communicate over a serial connection at 115200 baud rate with the oicmgr on a device that is connected to the host on port <code>/dev/ttys002</code> .
add	<code>newtmgr conn add myserial03 type=serial connstring="dev=/dev/ttys003, baud=57600"</code>	Creates a connection profile, named <code>myserial03</code> , to communicate over a serial connection at 57600 baud rate with the newtmgr on a device that is connected to the host on port <code>/dev/ttys003</code> .
add	<code>newtmgr conn add myudp5683 type=oic_udpconnstring=[127.0.0.1]:5683</code>	Creates a connection profile, named <code>myudp5683</code> , to communicate over UDP with the oicmgr on a device listening on localhost and port 5683.
add	<code>newtmgr conn add mybleprph type=ble connstring="peer_name=nimble</code>	Creates a connection profile, named <code>mybleprph</code> , to communicate over BLE, using the native OS BLE support, with the newtmgr on a device named <code>nimble-bleprph</code> .
add	<code>newtmgr conn add mybletype=ble connstring=""</code>	Creates a connection profile, named <code>myble</code> , to communicate over BLE, using the native OS BLE support, with the newtmgr on a device. You must use the <code>--name</code> flag to specify the device name when you issue a <code>newtmgr</code> command that communicates with the device.
add	<code>newtmgr conn add myblehostd type=oic_bhd connstring="peer_name=nimble ctrr_path=/dev/cu.usbmodem14221"</code>	Creates a connection profile, named <code>myblehostd</code> , to communicate over BLE, using the blehostd implementation, with the oicmgr on a device named <code>nimble-bleprph</code> . The BLE controller is connected to the host on USB port <code>/dev/cu.usbmodem14211</code> and uses static random address.
delete	<code>newtmgr conn delete myserial02</code>	Deletes the connection profile named <code>myserial02</code>
delete	<code>newtmgr conn delete myserial02</code>	Deletes the connection profile named <code>myserial02</code>
show	<code>newtmgr conn show myserial01</code>	Displays the information for the <code>myserial01</code> connection profile.
show	<code>newtmgr conn show</code>	Displays the information for all connection profiles.

9.1.4 newtmgr crash

Send a crash command to a device.

Usage:

```
newtmgr crash <assert|div0|jump0|ref0|wdog> -c <conn_profile> [flags]
```

Global 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)

Description

Sends a crash command to a device to run one of the following crash tests: `div0`, `jump0`, `ref0`, `assert`, `wdog`. Newtmgr uses the `conn_profile` connection profile to connect to the device.

Examples

Usage	Explanation
<code>newtmgr crash div0 -c profile01</code>	Sends a request to a device to execute a divide by 0 test. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
<code>newtmgr crash ref0 -c profile01</code>	Sends a request to a device to execute a nil pointer reference test. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.

9.1.5 newtmgr datetime

Manage datetime on a device.

Usage:

```
newtmgr datetime [rfc-3339-date-string] -c <conn_profile> [flags]
```

Global 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)

Description

Reads or sets the datetime on a device. Specify a `datetime-value` in the command to set the datetime on the device. Newtmgr uses the `conn_profile` connection profile to connect to the device.

Note: You must specify the `datetime-value` in the RFC 3339 format.

Examples

Usage	Explanation
<code>newtmgr datetime-c profile01</code>	Reads the datetime value from a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
<code>newtmgr datetime 2017-03-01T22:44:00-c profile01</code>	Sets the datetime on a device to March 1st 2017 22:44:00 UTC. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
<code>newtmgr datetime 2017-03-01T22:44:00-08:00-c profile01</code>	Sets the datetime on a device to March 1st 2017 22:44:00 PST. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.

9.1.6 newtmgr echo

Send data to a device and display the echoed back data.

Usage:

```
newtmgr echo <text> -c <conn_profile> [flags]
```

Global 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)

Description

Sends the `text` to a device and outputs the text response from the device. Newtmgr uses the `conn_profile` connection profile to connect to the device.

Examples

Usage	Explanation
<code>newtmgr echo hello-c profile01</code>	Sends the text 'hello' to a device and displays the echoed back data. Newtmgr connects to the device over a connection specified in the <code>profile01</code> profile.

9.1.7 newtmgr fs

Access files on a device.

Usage:

```
newtmgr fs [command] -c <conn_profile> [flags]
```

Global Flags:

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

Description

The `fs` command provides the subcommands to download a file from and upload a file to a device. Newtmgr uses the `conn_profile` connection profile to connect to the device.

Sub-command	Explanation
<code>download</code>	The <code>newtmgr download <src-filename> <dst-filename></code> command downloads the file named <code><src-filename></code> from a device and names it <code><dst-filename></code> on your host.
<code>upload</code>	The <code>newtmgr upload <src-filename> <dst-filename></code> command uploads the file named <code><src-filename></code> to a device and names the file <code><dst-filename></code> on the device.

Examples

Usage	Explanation
<code>newtmgr fs download /cfg/mfg mfg.txt -c profile01</code>	Downloads the file name /cfg/mfg from a device and names the file mfg.txt on your host. Newtmgr connects to the device over a connection specified in the profile01 connection profile.
<code>newtmgr fs upload mymfg.txt /cfg/mfg -c profile01</code>	Uploads the file name mymfg.txt to a device and names the file cfg/mfg on the device. Newtmgr connects to the device over a connection specified in the profile01 connection profile.

9.1.8 newtmgr image

Manage images on a device.

Usage:

```
newtmgr image [command] -c <connection_profile> [flags]
```

Flags:

The coredownload subcommand uses the following local flags:

<code>-n, --bytes uint32</code>	Number of bytes of the core to download
<code>-e, --elfify</code>	Create an ELF file
<code>--offset uint32</code>	Offset of the core file to start the download

Global Flags:

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

Description

The image command provides subcommands to manage core and image files on a device. Newtmgr uses the conn_profile connection profile to connect to the device.

Sub-command	Explanation
confirm	The <code>newtmgr image confirm [hex-image-hash]</code> command makes an image setup permanent on a device. If a hex-image-hash hash value is specified, Mynewt permanently switches to the image identified by the hash value. If a hash value is not specified, the current image is made permanent.
coreconvert	The <code>newtmgr image coreconvert <core-filename> <elf-file></code> command converts the core-filename core file to an ELF format and names it elf-file. Note: This command does not download the core file from a device. The core file must exist on your host.
coredownload	The <code>newtmgr image coredownload <core-filename></code> command downloads the core file from a device and names the file core-filename on your host. Use the local flags under Flags to customize the command.
coreerase	The <code>newtmgr image coreerase</code> command erases the core file on a device.
corelist	The <code>newtmgr image corelist</code> command lists the core(s) on a device.
erase	The <code>newtmgr image erase</code> command erases an unused image from the secondary image slot on a device. The image cannot be erased if the image is a confirmed image, is marked for test on the next reboot, or is an active image for a split image setup.
list	The <code>newtmgr image list</code> command displays information for the images on a device.
test	The <code>newtmgr test <hex-image-hash></code> command tests the image, identified by the hex-image-hash hash value, on next reboot.
upload	The <code>newtmgr image upload <image-file></code> command uploads the image-file image file to a device.

Examples

Sub-command	Usage	Explanation
confirm	<code>newtmgr confirm -c profile01</code>	Makes the current image setup on a device permanent. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
confirm	<code>newtmgr confirmbe9699809a049...73d77f -c profile01</code>	Makes the image, identified by the <code>be9699809a049...73d77f</code> hash value, setup on a device permanent. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
core-convert	<code>newtmgr image coreconvert mycore mycore.elf</code>	Converts the <code>mycore</code> file to the ELF format and saves it in the <code>mycore.elf</code> file.
core-download-load	<code>newtmgr image coredownload mycore -c profile01</code>	Downloads the core from a device and saves it in the <code>mycore</code> file. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
core-download-load	<code>newtmgr image coredownload mycore -e -c profile01</code>	Downloads the core from a device, converts the core file into the ELF format, and saves it in the <code>mycore</code> file. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
core-download-load	<code>newtmgr image coredownload mycore --offset 10 -n 30 -c profile01</code>	Downloads 30 bytes, starting at offset 10, of the core from a device and saves it in the <code>mycore</code> file. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
coreera	<code>newtmgr image coreerase -c profile01</code>	Erases the core file on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
corelist	<code>newtmgr image corelist -c profile01</code>	Lists the core files on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
erase	<code>newtmgr image erase -c profile01</code>	Erases the image, if unused, from the secondary image slot on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
list	<code>newtmgr image list -c profile01</code>	Lists the images on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
test	<code>newtmgr image test be9699809a049...73d77f</code>	Tests the image, identified by the <code>be9699809a049...73d77f</code> hash value, during the next reboot on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
upload	<code>newtmgr image upload btshell.img -c profile01</code>	Uploads the <code>btshell.img</code> image to a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.

9.1.9 newtmgr log

Manage logs on a device.

Usage:

```
newtmgr log [command] -c <conn_profile> [flags]
```

Global 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)

Description

The log command provides subcommands to manage logs on a device. Newtmgr uses the `conn_profile` connection profile to connect to the device.

Sub-command	Explanation
clear	The <code>newtmgr log clear</code> command clears the logs on a device.
level_list	The <code>newtmgr level_list</code> command shows the log levels on a device.
list	The <code>newtmgr log list</code> command shows the log names on a device.
module_list	The <code>newtmgr log module_list</code> command shows the log module names on a device.
show	<p>The <code>newtmgr log show</code> command displays logs on a device. The command format is: <code>newtmgr log show [log_name [min-index [min-timestamp]]] -c <conn_profile></code></p> <p>The following optional parameters can be used to filter the logs to display:</p> <p>log_name:</p> <p>Name of log to display. If <code>log_name</code> is not specified, all logs are displayed.</p> <p>min-index:</p> <p>Minimum index of the log entries to display. This value is only valid when a <code>log_name</code> is specified. The value can be <code>last</code> or a number. If the value is <code>last</code>, only the last log entry is displayed. If the value is a number, log entries with an index equal to or higher than <code>min-index</code> are displayed.</p> <p>min-timestamp:</p> <p>Minimum timestamp of log entries to display. The value is only valid if <code>min-index</code> is specified and is not the value <code>last</code>. Only log entries with a timestamp equal to or later than <code>min-timestamp</code> are displayed. Log entries with a timestamp equal to <code>min-timestamp</code> are only displayed if the log entry index is equal to or higher than <code>min-index</code>.</p>

Examples

Sub-comm	Usage	Explanation
clear	<code>newtmgr log clear-c profile01</code>	Clears the logs on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
level_l	<code>newtmgr log level_list -c profile01</code>	Shows the log levels on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
list	<code>newtmgr log list-c profile01</code>	Shows the log names on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
mod- ule_lis	<code>newtmgr log module_list-c profile01</code>	Shows the log module names on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
show	<code>newtmgr log show -c profile01</code>	Displays all logs on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
show	<code>newtmgr log show reboot_log -c profile01</code>	Displays all log entries for the <code>reboot_log</code> on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
show	<code>newtmgr log show reboot_log last -c profile01</code>	Displays the last entry from the <code>reboot_log</code> on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
show	<code>newtmgr log show reboot_log 2 -c profile01</code>	Displays the <code>reboot_log</code> log entries with an index 2 and higher on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
show	<code>newtmgr log show reboot_log 5 123456 -c profile01</code>	Displays the <code>reboot_log</code> log entries with a timestamp higher than 123456 and log entries with a timestamp equal to 123456 and an index equal to or higher than 5. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.

9.1.10 newtmgr mpstat

Read memory pool statistics from a device.

Usage:

```
newtmgr mpstat -c <conn_profile> [flags]
```

Global 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)

Description

Reads and displays the memory pool statistics from a device. Newtmgr uses the `conn_profile` connection profile to connect to the device. It lists the following statistics for each memory pool:

- **name:** Memory pool name
- **blksz:** Size (number of bytes) of each memory block
- **cnt:** Number of blocks allocated for the pool
- **free:** Number of free blocks
- **min:** The lowest number of free blocks that were available

Examples

Usage	Explanation
<code>newtmgr mpstat -c profile01</code>	Reads and displays the memory pool statistics from a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.

Here is an example output for the `myble` application from the [Enabling Newt Manager in any app](#) tutorial:

```
$ newtmgr mpstat -c myserial
      name blksz  cnt  free   min
ble_att_svr_entry_pool    20    75    0    0
ble_att_svr_prep_entry_pool 12    64    64   64
        ble_gap_update    24     1     1    1
        ble_gattc_proc_pool 56     4     4    4
        ble_gatts_clt_cfg_pool 12     2     1    1
ble_hci_ram_evt_hi_pool   72     2     2    1
ble_hci_ram_evt_lo_pool   72     8     8    8
        ble_hs_conn_pool   92     1     1    1
        ble_hs_hci_ev_pool 16    10    10    9
        ble_l2cap_chan_pool 28     3     3    3
        ble_l2cap_sig_proc_pool 20     1     1    1
        btshell_chr_pool   36    64    64   64
```

(continues on next page)

(continued from previous page)

btshell_dsc_pool	28	64	64	64
btshell_svc_pool	36	32	32	32
msys_1	292	12	9	6

9.1.11 newtmgr reset

Send a reset request to a device.

Usage:

```
newtmgr reset -c <conn_profile> [flags]
```

Global 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)

Description

Resets a device. Newtmgr uses the `conn_profile` connection profile to connect to the device.

Examples

Usage	Explanation
<code>newtmgr reset -c profile01</code>	Resets a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.

9.1.12 newtmgr run

Run test procedures on a device.

Usage:

```
newtmgr run [command] -c <conn_profile> [flags]
```

Global Flags:

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

Description

The run command provides subcommands to run test procedures on a device. Newtmgr uses the `conn_profile` connection profile to connect to the device.

Sub-command	Explanation
<code>list</code>	The <code>newtmgr run list</code> command lists the registered tests on a device.
<code>test</code>	The <code>newtmgr run test [all testname] [token-value]</code> command runs the <code>testname</code> test or all tests on a device. All tests are run if <code>all</code> or no <code>testname</code> is specified. If a <code>token-value</code> is specified the token value is output with the log messages.

Examples

Usage	Explanation
<code>newtmgr run list -c profile01</code>	Lists all the registered tests on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
<code>newtmgr run test all201612161220-c profile01</code>	Runs all the tests on a device. Outputs the <code>201612161220</code> token with the log messages. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
<code>newtmgr run test mynewtsanity-c profile01</code>	Runs the <code>mynewtsanity</code> test on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.

9.1.13 newtmgr stat

Read statistics from a device.

Usage:

```
newtmgr stat <stats_name> -c <conn_profile> [flags]
newtmgr stat [command] -c <conn_profile> [flags]
```

Global 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)

Description

Displays statistic for the stats named <stats_name> from a device. You can use the `list` subcommand to get a list of the stats names from the device. Newtmgr uses the `conn_profile` connection profile to connect to the device.

Sub-command	Explanation
stat	The <i>newtmgr stat command</i> displays the statistics for the <code>stats_name</code> Stats from a device.
list	The <code>newtmgr stat list</code> command displays the list of Stats names from a device.

Examples

Sub-command	Usage
<code>newtmgr stat ble_att -c profile01</code>	Displays the <code>ble_att</code> statistics on a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.
<code>newtmgr stat list -c profile01</code>	Displays the list of Stats names from a device. Newtmgr connects to the device over a connection specified in the <code>profile01</code> connection profile.

Here are some example outputs for the `myble` application from the [Enabling Newt Manager in any app](#) tutorial:

The statistics for the `ble_att` Stats:

```
$ newtmgr stat ble_att -c myserial
stat group: ble_att
    0 error_rsp_rx
    0 error_rsp_tx
    0 exec_write_req_rx
    0 exec_write_req_tx
```

(continues on next page)

(continued from previous page)

```
0 exec_write_rsp_rx
0 exec_write_rsp_tx
0 find_info_req_rx
0 find_info_req_tx
0 find_info_rsp_rx
0 find_info_rsp_tx
0 find_type_value_req_rx
0 find_type_value_req_tx
0 find_type_value_rsp_rx
0 find_type_value_rsp_tx

...
0 read_rsp_rx
0 read_rsp_tx
0 read_type_req_rx
0 read_type_req_tx
0 read_type_rsp_rx
0 read_type_rsp_tx
0 write_cmd_rx
0 write_cmd_tx
0 write_req_rx
0 write_req_tx
0 write_rsp_rx
0 write_rsp_tx
```

The list of Stats names using the list subcommand:

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

9.1.14 newtmgr taskstat

Read task statistics from a device.

Usage:

```
newtmgr taskstat -c <conn_profile> [flags]
```

Global 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)

Description

Reads and displays the task statistics from a device. Newtmgr uses the `conn_profile` connection profile to connect to the device. It lists the following statistics for each task:

- **task**: Task name
- **pri**: Task priority
- **runtime**: The time (ms) that the task has been running for
- **csw**: Number of times the task has switched context
- **stksz**: Stack size allocated for the task
- **stkuse**: Actual stack size the task uses
- **last_checkin**: Last sanity checkin with the *Sanity Task*
- **next_checkin**: Next sanity checkin

Examples

Usage	Explanation
<code>newtmgr taskstat -c profile0</code>	Reads and displays the task statistics from a device. Newtmgr connects to the device over a connection specified in the <code>profile0</code> connection profile.

Here is an example output for the `myble` application from the [Enabling Newt Manager in any app](#) tutorial:

```
$ newtmgr taskstat -c myserial
  task pri tid runtime      csw      stksz      stkuse last_checkin next_checkin
  ble_ll  0    2        0       12       80        58        0        0
  idle   255  0    16713       95       64        31        0        0
  main   127  1        2       81      512       275        0        0
```

9.2 Install

9.2.1 Installing Newtmgr on Mac OS

Newtmgr is supported on Mac OS X 64 bit platforms and has been tested on Mac OS 10.11 and higher.

This page shows you how to install the following versions of newtmgr:

- Upgrade to or install the latest release version (1.4.1).
- Install the latest from the master branch (unstable).

- *Adding the Mynewt Homebrew Tap*
- *Upgrading to or Installing the Latest Release Version*
 - *Upgrading to the Latest Release Version of Newtmgr*
 - *Installing the Latest Release Version of Newtmgr*
- *Checking the Installed Version*
- *Installing Newtmgr from the Master Branch*

See [Installing Previous Releases of Newtmgr](#) to install an earlier version of newtmgr.

Note: If you would like to contribute to the newtmgr tool, see [Contributing to Newt or Newtmgr Tools](#).

Adding the Mynewt Homebrew Tap

You should have added the **runtim eco/homebrew-mynewt** tap when you installed the **newt** tool. Run the following commands if you have not done so:

```
$ brew tap runtim eco/homebrew-mynewt  
$ brew update
```

Upgrading to or Installing the Latest Release Version

Perform the following to upgrade or install the latest release version of newtmgr.

Upgrading to the Latest Release Version of Newtmgr

If you have installed an earlier version of newtmgr using brew, run the following commands to upgrade to the latest version of newtmgr:

```
$ brew update  
$ brew upgrade mynewt-newtmgr
```

Installing the Latest Release Version of Newtmgr

Run the following command to install the latest release version of newtmgr:

```
$ brew update
$ brew install mynewt-newtmgr
==> Installing mynewt-newtmgr from runtimeco/mynewt
==> Downloading https://github.com/runtimeco/binary-releases/raw/master/mynewt-newt-
   ↵ tools_1.4.1/mynewt-newtmgr-1.4.1.sierra.bottle.tar.gz
==> Downloading from https://raw.githubusercontent.com/runtimeco/binary-releases/master/
   ↵ mynewt-newt-tools_1.4.1/mynewt-newtmgr-1.4.1.sierra.bottle.tar.gz
#####
# ##################################################### 100.0%
==> Pouring mynewt-newtmgr-1.4.1.sierra.bottle.tar.gz
/usr/local/Cellar/mynewt-newtmgr/1.4.1: 3 files, 17.3MB
```

Notes: Homebrew bottles for newtmgr 1.4.1 are available for Mac OS Sierra, El Captian. If you are running an earlier version of Mac OS, the installation will install the latest version of Go and compile newtmgr locally.

Checking the Installed Version

Check that you are using the installed version of newtmgr:

```
$ which newtmgr
/usr/local/bin/newtmgr
ls -l /usr/local/bin/newtmgr
lrwxr-xr-x 1 user staff 42 Jul 1 21:15 /usr/local/bin/newtmgr -> ../Cellar/mynewt-
   ↵ newtmgr/1.4.1/bin/newtmgr
```

Note: If you previously built newtmgr from source and the output of `which newtmgr` shows “`$GOPATH/bin/newtmgr`”, you will need to move “`$GOPATH/bin`” after “`/usr/local/bin`” for your PATH in `~/.bash_profile`, and source `~/.bash_profile`.

Get information about newtmgr:

```
$ newtmgr help
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
```

(continues on next page)

(continued from previous page)

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.

Installing Newtmgr from the Master Branch

We recommend that you use the latest release version of newtmgr. If you would like to use the master branch with the latest updates, you can install newtmgr from the HEAD of the master branch.

Notes:

- The master branch may be unstable.
- This installation will install the latest version of Go on your computer, if it is not installed, and compile newtmgr locally.

If you already installed newtmgr, unlink the current version:

```
$ brew unlink mynewt-newtmgr
```

Install the latest unstable version of newtmgr from the master branch:

```
$ brew install mynewt-newtmgr --HEAD
==> Installing mynewt-newtmgr from runtimeco/mynewt
==> Cloning https://github.com/apache/mynewt-newtmgr.git
Cloning into '/Users/wanda/Library/Caches/Homebrew/mynewt-newtmgr--git'...
remote: Counting objects: 2169, done.
remote: Compressing objects: 100% (1752/1752), done.
remote: Total 2169 (delta 379), reused 2042 (delta 342), pack-reused 0
Receiving objects: 100% (2169/2169), 8.13 MiB | 5.47 MiB/s, done.
Resolving deltas: 100% (379/379), done.
==> Checking out branch master
==> go get github.com/currantlabs/ble
==> go get github.com/raff/goble
==> go get github.com/mgutz/logxi/v1
==> go install
/usr/local/Cellar/mynewt-newtmgr/HEAD-2d5217f: 3 files, 17.3MB, built in 1 minute 10.2
seconds
```

To switch back to the latest stable release version of newtmgr, you can run:

```
$ brew switch mynewt-newtmgr 1.4.1
Cleaning /usr/local/Cellar/mynewt-newtmgr/1.4.1
Cleaning /usr/local/Cellar/mynewt-newtmgr/HEAD-2d5217f
1 links created for /usr/local/Cellar/mynewt-newtmgr/1.4.1
```

9.2.2 Installing Newtmgr on Linux

You can install the latest release (1.4.1) of the newtmgr tool from a Debian binary package (amd64). You can also download and build the latest release version of newtmgr from source.

This page shows you how to:

- Set up your computer to download Debian binary packages from the runtimeco APT repository.
Note: The key for signing the repository has changed. If you set up your computer before release 1.1.0, you will need to download and import the public key again.
- Install the latest release version of newtmgr from a Debian binary package. You can use apt-get to install the package or manually download and install the Debian binary package.
- Download, build, and install the latest release version of newtmgr from source.

- *Setting Up Your Computer to use apt-get to Install the Package*
- *Installing the Latest Release of Newtmgr from a Binary Package*
 - *Method 1: Using apt-get to Upgrade or to Install*
 - *Method 2: Downloading and Installing the Debian Package Manually*
- *Installing the Latest Release Version of Newtmgr from Source*
- *Checking the Latest Version of Newtmgr is Installed*

See *Installing Previous Releases of Newtmgr* to install an earlier version of newtmgr.

If you are installing on an amd64 platform, we recommend that you install from the binary package.

Note: We have tested the newtmgr tool binary and apt-get install from the runtimeco APT repository for Ubuntu version 16. Earlier Ubuntu versions (for example: Ubuntu 14) may have incompatibility with the repository. You can manually download and install the Debian binary package.

Note: See *Contributing to Newt or Newtmgr Tools* if you want to:

- Use the newtmgr tool with the latest updates from the master branch. The master branch may be unstable and we recommend that you use the latest stable release version.
- Contribute to the newtmgr tool.

Setting Up Your Computer to use apt-get to Install the Package

The newtmgr Debian packages are stored in a private APT repository on <https://github/runtimeco/debian-mynewt>. To use apt-get, you must set up the following on your computer to retrieve packages from the repository:

Note: You only need to perform this setup once on your computer. However, if you previously downloaded and imported the public key for the runtimeco APT repository, you will need to perform step 2 again as the key has changed.

1. Install the `apt-transport-https` package to use HTTPS to retrieve packages.
 2. Download the public key for the runtimeco APT repository and import the key into the apt keychain.
 3. Add the repository for the binary and source packages to the apt source list.
1. Install the `apt-transport-https` package:

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https
```

2. Download the public key for the runtimeco apt repo (**Note:** There is a - after apt-key add):

```
$ wget -qO - https://raw.githubusercontent.com/runtimeco/debian-mynewt/master/
->mynewt.gpg.key | sudo apt-key add -
```

3. Add the repository for the binary packages to the mynewt.list apt source list file.

```
$ sudo -s
[sudo] password for <user>:
root$ cat > /etc/apt/sources.list.d/mynewt.list <<EOF
deb https://raw.githubusercontent.com/runtimeco/debian-mynewt/master latest main
EOF
root$ exit
```

Note: Do not forget to exit the root shell.

4. Verify the content of the source list file:

```
$ more /etc/apt/sources.list.d/mynewt.list
deb https://raw.githubusercontent.com/runtimeco/debian-mynewt/master latest main
```

5. Update the available packages:

```
$ sudo apt-get update
```

Note: If you are not using Ubuntu version 16, you may see the following errors. We have provided instructions on how to manually download and install the binary package.

```
W: Failed to fetch https://raw.githubusercontent.com/runtimeco/debian-
->mynewt/master/dists/latest/main/source/Sources  HttpError404
```

Installing the Latest Release of Newtmgr from a Binary Package

You can use either apt-get to install the package, or manually download and install the Debian binary package.

Method 1: Using apt-get to Upgrade or to Install

Run the following commands to upgrade or install the latest version of newtmgr:

```
$ sudo apt-get update
$ sudo apt-get install newtmgr
```

Method 2: Downloading and Installing the Debian Package Manually

Download and install the package manually.

```
$ wget https://raw.githubusercontent.com/runtimeco/debian-mynewt/master/pool/main/n/  
↳ newtmgr/newtmgr_1.4.1-1_amd64.deb  
$ sudo dpkg -i newtmgr_1.4.1-1_amd64.deb
```

See *Checking the Latest Version of Newtmgr is Installed* to verify that you are using the installed version of newtmgr.

Installing the Latest Release Version of Newtmgr from Source

If you are running Linux on a different architecture, you can build and install the latest release version of newtmgr from source.

1. Download and install the latest version of [Go](#). Newtmgr requires Go version 1.7.6 or higher.
2. Create a Go workspace in the /tmp directory:

```
$ cd /tmp  
$ mkdir go  
$ cd go  
$ export GOPATH=/tmp/go
```

3. Run `go get` to download the newtmgr source. Note that `go get` pulls down the HEAD from the master branch in git, builds, and installs newtmgr.

```
$ go get mynewt.apache.org/newtmgr/newtmgr  
$ ls -l /tmp/go/bin/newtmgr  
-rwxr-xr-x 1 user staff 17884488 Jul 29 16:25 /tmp/go/bin/newtmgr
```

4. Check out the source from the latest release version:

```
$ cd src/mynewt.apache.org/newtmgr  
$ git checkout mynewt_1_4_1_tag  
Note: checking out 'mynewt_1_4_1_tag'.
```

5. Build newtmgr from the latest release version:

```
$ cd newtmgr  
$ GO111MODULE=on go install  
$ ls /tmp/go/bin/newtmgr  
-rwxr-xr-x 1 user staff 17888680 Jul 29 16:28 /tmp/go/bin/newtmgr
```

6. If you have a Go workspace, remember to reset your GOPATH to your Go workspace.
7. Copy the newtmgr executable to a bin directory in your path. You can put it in the /usr/bin or the \$GOPATH/bin directory.

Checking the Latest Version of Newtmgr is Installed

1. Run `which newtmgr` to verify that you are using the installed version of newtmgr.
2. Get information about the newtmgr tool:

```
$ newtmgr
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.
```

9.2.3 Installing Newtmgr on Windows

This guide shows you how to install the latest release of newtmgr from binary or from source. The tool is written in Go (golang).

It assumes that you have already installed the *newt tool on Windows* and have the Windows development environment set up.

This guide shows you how to perform the following:

1. Install latest release of newtmgr from binary.
2. Install latest release of newtmgr from source.

- *Installing the Latest Release of Newtmgr Tool from Binary*
- *Installing the Latest Release of Newtmgr from Source*
- *Checking the Installed Version*

See *Installing Previous Releases of Newtmgr* to install an earlier version of newtgmr.

Note: If you would like to contribute to the newtmgr tool, see *Contributing to Newt or Newtmgr Tools*.

Installing the Latest Release of Newtmgr Tool from Binary

You can install the latest release of newtmgr from binary. It has been tested on Windows 10 64 bit platform.

1. Start a MinGW terminal.
2. Download the newtmgr binary tar file from one of the mirror sites .:

```
$ wget -P /tmp http://www.apache.org/dyn/closer.lua/mynewt/apache-mynewt-1.4.1/
→apache-mynewt-newtmgr-bin-windows-1.4.1.tgz
```

3. Extract the file:
 - If you previously built newtmgr from the master branch, you can extract the file into your \$GOPATH/bin directory. Note: This overwrites the current newtmgr.exe in the directory and assumes that you are using \$GOPATH/bin for your Go applications.

```
tar -xzf /tmp/apache-mynewt-newtmgr-bin-windows-1.4.1.tgz -C $GOPATH/bin
```

- If you are installing newtmgr for the first time and do not have Go setup, you can extract into /usr/bin directory:

```
tar -xzf /tmp/apache-mynewt-newtmgr-bin-windows-1.4.1.tgz -C /usr/bin
```

4. Verify the installed version of newtmgr. See *Checking the Installed Version*.

Installing the Latest Release of Newtmgr from Source

If you have an older version of Windows or a 32 bit platform, you can build and install the latest release version of newtmgr from source.

1. Download and install the latest version of Go. Newtmgr requires Go version 1.7.6 or higher.
2. Start MinGW terminal.
3. Create a Go workspace in the /tmp directory:

```
$ cd /tmp
$ mkdir go
$ cd go
$ export GOPATH=/tmp/go
```

4. Run go get to download the newtmgr source. Note that go get pulls down the HEAD from the master branch in git, builds, and installs newtmgr.

```
$ go get mynewt.apache.org/newtmgr/newtmgr
```

Note If you get the following error, you may ignore it as we will rebuild newtmgr from the latest release version of newtmgr in the next step:

```
# github.com/currantlabs/ble/examples/lib/dev
..\..\..\github.com\currantlabs\ble\examples\lib\dev\dev.go:7: undefined:↳
DefaultDevice
```

5. Check out the source from the latest release version:

```
$ cd src/mynewt.apache.org/newtmgr
$ git checkout mynewt_1_4_1_tag
Note: checking out 'mynewt_1_4_1_tag'.
```

6. Build newtmgr from the latest release version:

```
$ cd newtmgr
$ GO111MODULE=on go install
$ ls /tmp/go/bin/newtmgr.exe
-rwxr-xr-x 1 user None 15457280 Sep 12 00:30 /tmp/go/bin/newtmgr.exe
```

7. If you have a Go workspace, remember to reset your GOPATH to your Go workspace.
8. Copy the newtmgr executable to a bin directory in your path. You can put it in the /usr/bin or the \$GOPATH/bin directory.

Checking the Installed Version

1. Run `which newtmgr` to verify that you are using the installed version of newtmgr.
2. Get information about the newtmgr tool:

```
$ newtmgr
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
```

(continues on next page)

(continued from previous page)

```
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.
```

9.2.4 Installing Previous Releases of Newtmgr

This page shows you how to install previous releases of newtmgr for Mac OS, Linux, and Windows.

- [Mac OS](#)
- [Linux](#)
- [Windows](#)

Mac OS

You can install previous releases of newtmgr using `mynewt-newtmgr@X.Y` Homebrew formulas, where X.Y is a version number.

For example, if you want to install newtmgr 1.0, run the following commands:

```
$ brew update
$ brew install mynewt-newtmgr@1.0
```

Note: This is a keg-only installation. newtmgr 1.0 is installed in `/usr/local/Cellar/mynewt-newtmgr@1.0/1.0.0/bin` but not symlinked into `/usr/local/bin`.

If you need this version of newtmgr first in your PATH, run the following commands:

```
$ echo 'export PATH=/usr/local/Cellar/mynewt-newtmgr@1.0/1.0.0/bin:$PATH' >> ~/.bash_
↪profile
$ source ~/.bash_profile
```

You can also manually symlink into `/usr/local/bin` as follows:

1. Unlink newtmgr if you have the latest version of newtmgr installed:

```
$ brew unlink mynewt-newtmgr
```

2. Link `mynewt-newtmgr@1.0` into `/usr/local/bin`:

```
$ brew link -f mynewt-newtmgr@1.0
```

Linux

1. Download the binary:

Version	Download
1.0.0	newtmgr_1.0.0-1_amd64.deb
1.1.0	newtmgr_1.1.0-1_amd64.deb
1.3.0	newtmgr_1.3.0-1_amd64.deb

1. Run the `sudo apt-get remove newtmgr` command to remove the the current installation.
2. Install the package. For example, run `sudo dpkg -i newtmgr_1.0.0-1_amd64.deb` to install newtmgr 1.0.0

Windows

1. Download the binary:

Version	Download
1.1.0	newtmgr_1_1_0_windows_amd64.tar.gz
1.3.0	newtmgr_1_3_0_windows_amd64.tar.gz

1. Extract the file:

- If you previously built newtmgr from the master branch, you can extract the file into your `$GOPATH/bin` directory. Note: This overwrites the current `newtmgr.exe` in the directory and assumes that you are using `$GOPATH/bin` for your Go applications.

```
tar -xzf newtmgr_1_1_0_windows_amd64.tar.gz -C $GOPATH/bin
```

- If you are installing newtmgr for the first time and do not have a Go workspace setup, you can extract into `/usr/bin` directory:

```
tar -xzf newtmgr_1_1_0_windows_amd64.tar.gz -C /usr/bin
```


MYNEWT FAQ

Welcome to the Mynewt FAQ. Here are some commonly asked questions asked about the Mynewt OS, separated by categories.

10.1 General FAQs

- Make sure to update to the latest available version to ensure everything remains functional. There may be some new additions that break backwards compatibility or change how certain features work, so be sure to always update.
- If you are making changes to code, adding new features, etc., please update to the latest branch to ensure that any changes you make work with the newest changes and features in Mynewt.

10.1.1 Reduce Code Size for Mynewt Image

Q: How do I reduce the code size for my Mynewt image?

A: Please refer to the tutorial documentation on [reducing application code size](#).

10.1.2 compiler.yml vs. pkg.yml

Q: What's the difference between compiler.yml and pkg.yml?

A: compiler.yml defines a compiler. pkg.yml contains metadata about the package. All packages have a pkg.yml file, even compiler packages.

10.1.3 Version Control Applications with Git

Q: What's the recommended way to work with git when you want to version control your application? As apache-mynewt-core is already a repository, there is a repo in repo problem. Are there any good alternatives/tools to submodules, mirror, etc? Ideally, I want to version control everything from the top level project directory as well as upgrading apache-mynewt-core, pushing pull requests back to Mynewt if needed, etc.

A: You can simply have a separate git for your app. For example, if you followed the Blinky tutorial, your git would be in apps/foo, while repos gits are in repos. You may also keep your app in the core repo, just have your own working branch for it.

Another option is to have your git repository with local packages (including apps) and have repository.yml there so newt upgrade can download all dependencies. Just make sure to put e.g. bin, repos, and project.state, and others in .gitignore so they are not in version control.

10.2 Mynewt FAQ - Administrative

Here is a list of frequently asked questions about the project and administrative processes.

10.2.1 Administrative questions:

- *How do I submit a bug?*
- *How do I request a feature?*
- *I am not on the committer list. How do I submit a patch?*
- *I am a committer in the project. Can I merge my own Pull Request into the git repository?*
- *I would like to make some edits to the documentation. What do I do?*
- *I would like to make some edits to the documentation but want to use an editor on my own laptop. What do I do?*

How do I submit a bug?

If you do not have a JIRA account sign up for an account on [JIRA](#).

Submit a request to the @dev mailing list for your JIRA username to be added to the Apache Mynewt (MYNEWT) project. You can view the issues on JIRA for the MYNEWT project without an account but you need to log in for reporting a bug.

Log in. Choose the “MYNEWT” project. Click on the “Create” button to create a ticket. Choose “Bug” as the Issue Type. Fill in the bug description, how it is triggered, and other details.

How do I request a feature?

If you do not have a JIRA account sign up for an account on [JIRA](#).

Submit a request to the @dev mailing list for your JIRA username to be added to the Apache Mynewt (MYNEWT) project. You can view the issues on JIRA for the MYNEWT project without an account but you need to log in for reporting a bug.

Log in. Choose the “MYNEWT” project. Click on the “Create” button to create a ticket. Choose “Wish” as the Issue Type. Fill in the feature description, benefits, and any other implementation details. Note in the description whether you want to work on it yourself.

If you are not a committer and you wish to work on it, someone who is on the committer list will have to review your request and assign it to you. You will have to refer to this JIRA ticket in your pull request.

I am not on the committer list. How do I submit a patch?

You submit your proposed changes for your peers with committer status to review and merge.

The process to submit a Pull Request on github.com is described on the [Confluence page for the project](#).

I am a committer in the project. Can I merge my own Pull Request into the git repository?

Yes, but only if your Pull Request has been reviewed and approved by another committer in Apache Mynewt. The process to merge a Pull Request is described on the [Confluence page for the project](#).

I would like to make some edits to the documentation. What do I do?

You submit your proposed changes for your peers with committer status to review and merge.

Each Mynewt repository has its own set of related documentation in the docs/ folder. The overall project documentation is in [mynewt-documentation](#) on github.com.

Navigate to the file you wish to edit on github.com. All the technical documentation is in reStructuredText files under the /docs directory. Click on the pencil icon (“Edit the file in your fork of this project”) and start making changes.

Click the green “Propose file change” button. You will be directed to the page where you can start a pull request from the branch that was created for you. The branch is gets an automatic name patch-# where # is a number. Click on the green “Compare & pull request” to open the pull request.

In the comment for the pull request, include a description of the changes you have made and why. Github will automatically notify everyone on the commits@mynewt.apache.org mailing list about the newly opened pull requests. You can open a pull request even if you don’t think the code is ready for merging but want some discussion on the matter.

Upon receiving notification, one or more committers will review your work, ask for edits or clarifications, and merge when your proposed changes are ready.

If you want to withdraw the pull request simply go to your fork <https://github.com/<your github username>/mynewt-documentation> and click on “branches”. You should see your branch under “Your branches”. Click on the delete icon.

I would like to make some edits to the documentation but want to use an editor on my own laptop. What do I do?

You submit your proposed changes for your peers with committer status to review and merge.

Go to the [documentation mirror](#) on github.com. You need to create your own fork of the repo in github.com by clicking on the “Fork” button on the top right. Clone the forked repository into your laptop (using `git clone` from a terminal or using the download buttons on the github page)and create a local branch for the edits and switching to it (using `git checkout -b <new-branchname>` or GitHub Desktop).

Make your changes using the editor of your choice. Push that branch to your fork on github. Then submit a pull request from that branch on your github fork.

The review and merge process is the same as other pull requests described for earlier questions.

10.3 Mynewt FAQ - Bluetooth

- [*NimBLE on nRF52840*](#)
- [*Trigger Exactly One BLE Advertisement Immediately*](#)
- [*Supported Bluetooth Radio Transceivers*](#)
- [*NimBLE Operational Questions*](#)
- [*blemesh Forgets All Keys on Restart*](#)
- [*L2CAP Connection*](#)
- [*Bitbang, BLE Mesh, BLE Advertising*](#)
- [*Extended Advertising with btshell*](#)
- [*Configuring Maximum Number of Connections for blehci*](#)
- [*Disconnect/Crash While Writing Analog Value From Central Module*](#)
- [*Documentation for ble_gap_disc_params*](#)
- [*Multicast Messaging and Group Messaging*](#)
- [*Read the Value of a Characteristic of a Peripheral Device from a Central Device*](#)

10.3.1 NimBLE on nRF52840

Q: Is the nRF52840 supported by NimBLE?

A: The nRF52840 is supported, including Bluetooth 5 features.

10.3.2 Trigger Exactly One BLE Advertisement Immediately

Q: Is there a way to trigger exactly one BLE advertisement immediately? I basically want to use BLE as a means to advertise to a whole group of devices and it needs to be relatively time-precise. Hoping for about 1ms precision, but it's okay if there's a delay as long as it's deterministic.

A: With extended advertising you can enable an advertising instance for exactly 1 advertising event but there will be some delay before controller starts advertising which depends on the level of precision. The NimBLE controller always schedules the 1st advertising event with a constant 5ms delay. However, if there are other things scheduled in the controller it may need to schedule it later so it's not guaranteed to be deterministic. Periodic advertising is currently not supported.

10.3.3 Supported Bluetooth Radio Transceivers

Q: Are there any other Bluetooth radio transceivers that are supported already or anyone working on? If there is a Bluetooth radio that can be integrated using the HCI, we can use the NimBLE host? If so, I suppose we don't need detailed specs of the Bluetooth radio?

A: You should be able to run the NimBLE host against pretty much any controller using HCI. We have a BLE controller implementation for Nordic only, but host (in theory) should work with all. More likely, it will work with 5.0 controller. For 4.x controllers it may fail on initialization since we do not check controller features - but this can be fixed as for other radio transceivers. For example, there's NXP KW41Z which has BLE-compatible radio documented. However, NXP documentation is more or less a description of zillions of different registers so not as friendly as the one from Nordic which explains how to actually use it.

10.3.4 NimBLE Operational Questions

Q: Can I update firmware via BLE for multiple devices simultaneously using a single device/phone supporting Bluetooth?

A: Yes, it is possible to update several Nimble devices simultaneously from a single phone (e.g., central) as long as the central can handle all the simultaneous Bluetooth connections. You can also do it using your computer as a central with newtmgr. Just open two terminals and initiate a newtmgr image upload command in each, each to a different device. You will, however, get better overall throughput if you limit yourself to one upgrade at a time.

Q: I have the following doubts on NimBLE: The document says 32+ concurrent connections, multiple connections in simultaneous central and peripheral roles. Does that mean the “device running NimBLE” can connect to 32 different other devices like phones?

A: Yes, with one caveat: each of the 32 centrals needs to be sufficiently cooperative in choosing connection partners (<http://www.novelbits.io/ble-connection-intervals/>). Your app might need to request different connection parameters from the centrals (using `ble_gap_update_params`).

Of course, you will also need to build the Mynewt image device with a configuration suitable for 32 concurrent connections (e.g., `NIMBLE_MAX_CONNECTIONS=32`, etc.)

10.3.5 blemesh Forgets All Keys on Restart

Q: Is it expected that the blemesh example forgets all the keys on restart and needs to be provisioned again? If so, how can I implement key persistence myself? Is there any API to obtain / provide mesh keys before the mesh node is started? I found `bt_mesh_provision`, but the comment there seems to indicate that this is not the right way to use it.

A: Mesh implementations do not persist keys at the moment. There is a plan to add it but not sure about timeline. It probably needs to be implemented inside mesh implementation so there's no API, but if you'd like to hack something I suggest taking a look at `shell.c` and `testing.c` - there are some testing functions to add/display keys. Another area to look if you actually want to persist keys to flash is `net/nimble/host/store/config/src/ble_store_config.c`. This is the code that persists and restores security material for (non-mesh) Bluetooth.

10.3.6 L2CAP Connection

Q: I want to do an L2CAP connection, and am trying the auth-passkey command, but am not sure about the parameters `psm`, `action`, and `oob`. What is `psm`, and what is the value of that parameter in the btshell command `l2cap-connect`? How do I set the parameters `action` and `oob`?

A: `psm` stands for Protocol Service Multiplexer. You pass the `psm` value to either `l2-cap-connect` or `l2cap-create-server`. The parameters `action` and `oob` are just passing constant values as defined in the API.

10.3.7 Bitbang, BLE Mesh, BLE Advertising

Q: Is it possible to run bitbanging and BLE mesh at the same time? How about running BLE Mesh and BLE advertising at the same time?

A: It is possible to run bitbanging and BLE mesh at the same time, but the bitbanging UART takes a lot of CPU on Nordic. We've run it at 9600 which would probably be okay for lower rate devices, but for reliability it is recommended to run at 4800. If this is just for the console and your UART port is tied up, `rtt` is recommended. Take a look at [Segger RTT Console](#) for more information. However, bitbanger can be handy given limited UARTs.

You can certainly continue advertisements during connections, if you are using the GATT bearer for mesh. Mesh is also tied into the ext-adv bearer in Mynewt, which also allows for interleaving, even if you're transmitting mesh data on advertising channels.

10.3.8 Extended Advertising with btshell

Q: I am using `btshell` for advertising with nRF52. When I use 31 bytes, `mfg_data` accepts the data with extended advertising. But when I use more bytes than that, `mfg_data` doesn't accept it. Is 251 byte payload supported in extended advertising? How can I send more than a 251 byte payload on extended advertising?

A: You need to set the `BLE_EXT_ADV_MAX_SIZE` syscfg to your required value. For example:

```
newt target amend <your_target> syscfg=BLE_EXT_ADV=1:BLE_EXT_ADV_MAX_SIZE=1650
```

The default is 31 bytes, and the max is 1650.

Keep in mind that with extended advertising, you cannot set advertising data for an instance configured as scannable (e.g. `advertise-configure scannable=1`). Either set scan response data using `advertise-set-scan-rsp` command (parameters are the same as for `advertise-set-adv-data`) or configure the instance as non-scannable. For example, `advertise-configure` alone will configure the instance as non-connectable and non-scannable which means you can set advertising data. Also note that if you continue to use a scannable instance you will need to perform active scanning in order to get scan response data.

FYI, legacy advertising instances can accept both advertising and scan response data but since they use legacy PDUs the limit is still 31 bytes.

10.3.9 Configuring Maximum Number of Connections for blehci

Q: How do I set the maximum number of connections for the blehci *example*? I see there is a MYNEWT_VAL_BLE_MAX_CONNECTIONS, but I don't know how to set it.

A: You can set it in target settings:

```
newt target amend <target> syscfg=BLE_MAX_CONNECTIONS=2
```

and then rebuild using `newt build <target>`. MYNEWT_VAL_BLE_MAX_CONNECTIONS is just a symbol that is defined in `syscfg.h` which is autogenerated by newt tool and contains all the settings set at package/app/target level.

Q: What if I need to set 2 constants? What's the syntax?

A: You can set each setting in separate commands or separate key=value pairs with colon:

```
newt target amend <target> syscfg=FOO=1:BAR=2
```

Q: How do you know the constant is BLE_MAX_CONNECTIONS and not MYNEWT_VAL_BLE_MAX_CONNECTIONS? Is there a place I can see those names?

A: This is actually one of NimBLE's settings - you can find these settings available for different packages in the `syscfg.yml` files in the repository. You can also use `newt target config show <target>` to show all settings with their current values, and then change any of these settings accordingly. Each setting will get a symbol prefixed by MYNEWT_VAL_ in the autogenerated `syscfg.h` file so you can get the actual setting name from this symbol. For more info on System Configuration and Initialization, please visit the [Compile-Time Configuration and Initialization](#) page in the OS User Guide.

10.3.10 Disconnect/Crash While Writing Analog Value From Central Module

Q: I'm trying to write analog sensor data from my central module to my peripheral module. I can receive the values from the ADC callback perfectly, but I'm not able to write them to the peripheral module. The peripheral module disconnects right when the `ble_gattc_write_flat` command is called. What could be causing the issue?

A: First, check the reason for the disconnect. The gap event callback should indicate the reason in `disconnect.reason`. If the code never reaches the disconnect callback, then the code most likely crashed. If so, check whether `ble_gattc_write_flat` is called from an interrupt context. Calling into the BLE host from within an interrupt is a bad idea in general, and may cause a crash to occur because the Bluetooth host attempts to log to the console during the write procedure. Logging to the console uses quite a bit of stack space, so it is likely that the interrupt stack is overflowing.

Instead, you should send an event to the event queue and handle this in a task context. You'll need to associate the characteristic data with the event so that your event callback knows what payload to pass to the `ble_gattc_write_flat()` function. If you don't need to perform multiple writes in rapid succession, then you can just use a single global event and single global buffer. However, you will still need to make sure your buffer doesn't become corrupted by a subsequent ADC interrupt while you are in mid-write.

10.3.11 Documentation for ble_gap_disc_params

Q: Is there documentation somewhere on correct values for `ble_gap_disc_params`? I'm trying to do a passive discovery and getting `BLE_HS_EINVAL`.

A: Unfortunately, not at the moment. Here is a brief description of the fields:

- `itvl`: This is defined as the time interval from when the Controller started its last LE scan until it begins the subsequent LE scan. (units=0.625 msec)
- `window`: The duration of the LE scan. `LE_Scan_Window` shall be less than or equal to `LE_Scan_Interval` (units=0.625 msec)
- `filter_policy`: The only useful documentation is the table in the Bluetooth spec (section 2.E.7.8.10). This field controls which types of devices to listen for.
- `limited`: If set, only discover devices in limited discoverable mode.
- `passive`: If set, don't send scan requests to advertisers (i.e., don't request additional advertising data).
- `filter_duplicates`: If set, the controller ignores all but the first advertisement from each device.

10.3.12 Multicast Messaging and Group Messaging

Q: Is it possible to send a broadcast message by one of the devices present in the mesh (e.g. broadcast an event which happened)? Something like a push notification instead of continuously polling for it by a client.

A: It is possible to do so with a publish model. Group address or virtual address should help here, according to the Mesh spec. There is no real documentation on it but you can try it out on our `btermesh_shell` app. There is a `shell.c` file which exposes configuration client which you can use for testing (e.g. you can subscribe to virtual addresses). You can also trigger sending messages to devices. By playing with the `dst` command, you probably should be able to set destination to some group. However, since we do not support the provisioner role, there is a command provision which sets fixed keys so you can create a mesh network out of a couple of nodes without the actual provisioner.

10.3.13 Read the Value of a Characteristic of a Peripheral Device from a Central Device

Q: I want to read the value of a characteristic of a peripheral device from a central device which runs on Mynewt OS. How can I obtain the value using the following function?

```
int ble_gattc_read(uint16_t conn_handle, uint16_t attr_handle,  
                    ble_gatt_attr_fn *cb, void *cb_arg);
```

A: To see an example of this function being used, take a look at the `blecent` sample app. The data is in `attr->om`, which is an `mbuf` struct. There are dedicated APIs to access data in `mbufs` (see `os_mbuf.h`). `attr->om->om_data` is a raw pointer to access data in `mbuf` so you could also use it, but keep in mind that data in `mbuf` can be fragmented so you may not be able to access them easily like this. Thus, it is safer to use `mbuf` APIs instead. `os_mbuf_copydata` should be especially useful here since it can copy data from `mbuf` to flat buffer.

10.4 Mynewt FAQ - Bootloader and Firmware Upgrade

- *Firmware Upgrade Capability*
- *Bootloader Documentation*
- *MCUboot vs. Mynewt Bootloader*
- *boot_serial vs. bootutil*

10.4.1 Firmware Upgrade Capability

Q: I wanted to check if the stack provides firmware upgrade capability and if so, is there an example you can provide on how it is being done?

A: The `newtmgr` tool is used to upgrade Mynewt devices. `newtmgr` is a command line tool, but there are other client libraries available. There is some information listed under the “Image Upgrade” header in the [Split Image documentation](#).

10.4.2 Bootloader Documentation

Q: Is there any documentation on using the bootloader? It sounds like it has baked-in support for serial loading, but I can't find any details on serial protocol, or how to do a serial boot load. I assume we set a GPREGRET flag that tells the bootloader to expect to be flashed by serial, then it handles the rest. Is that true?

A: The serial bootloader would inspect a GPIO to see whether to wait for image upload commands or not. The protocol is the same `newtmgr` protocol we use for usual image uploads. For some the state reporting is simplified (omitted), and image upload goes to slot 0 instead of slot 1. The serial bootloading is built into `newtmgr`. For more information, refer to the documentation on the [Mynewt bootloader](#).

10.4.3 MCUboot vs. Mynewt Bootloader

Q: Is there any major difference between MCUBoot and the Mynewt bootloader?

A: They use different formats. The header is different as well, since you need to pass an extra flag (e.g. `-2` to `newt create-image` for MCUBoot). Visit the MCUBoot page for more documentation.

10.4.4 boot_serial vs. bootutil

Q: What is the difference between `boot_serial` and `bootutil`?

A: `boot_serial` is used only for downloading images over the serial port. If you are using `newtmgr` to upload image over serial, it is handled in `boot_serial`. All other bootloader code is in `bootutil`.

10.5 Mynewt FAQ - Drivers and Modules

- *Drivers in Mynewt*
- *Module Argument in Mynewt Logging Library*
- *Log Name vs. Module Number*

10.5.1 Drivers in Mynewt

Q: Is this a correct assumption about Mynewt, that if there exists no driver implementation for a specific SoC, in `hw/drivers/`, then it is not supported. For instance, there exists a flash driver for `at45db`, this implies that the Nordic nRF52 SoC is not supported at the moment?

A: `at45db` is SPI, and any SPI would work. You send SPI configuration info when initializing. SPI drivers are below the `hw/mcu/` tree. `hw/drivers/pwm` and `hw/drivers/adc` are SoC specific. In general, drivers are for peripherals that aren't universally supported. Features that all (or nearly all) MCUs support are implemented in the HAL. For example, internal flash support is a HAL feature. Visit the [HAL Documentation](#) for more information.

10.5.2 Module Argument in Mynewt Logging Library

Q: Can you tell me what the purpose of the module argument is in the Mynewt logging library? It looks like it just takes an `int`. Is this just to assign an integer ID for each module that logs?

A: It is just an integer which accompanies each log entry. It provides context for each log entry, and it allows a client to filter messages based on module (e.g. “give me all the file system log entries”).

10.5.3 Log Name vs. Module Number

Q: So, what is the conceptual difference between a log name, and a module number? It seems like a log type would be assigned the same name as the module that is using it, and that the module number is just a numerical ID for the module. Basically, I don't understand what the purpose of storing the name into the log type is, and passing the module number in as part of `LOG_<LEVEL>` macro.

A: A log just represents a medium or region of storage (e.g., “console”, or “flash circular buffer in 12kB of flash, starting at `0x0007d000`”). Many parts of the system can write to the same log, so you may end up with Bluetooth, file system, and kernel scheduler entries all in the same log. The module ID distinguishes these entries from one another. You can control level per module, so you can say, “give me all bluetooth warnings, but only give me system level errors”.

Q: Okay, so for something like console logging, we would likely register one log for the entire application, and give each module an ID?

A: I think the thought is that would be the debug log, and during development you could pipe that to console. In production, that might go in the spare image slot. I'm not sure if we support it yet, but we should make sure the log can write to multiple handlers at the same time.

10.6 Mynewt FAQ - File System

- *nffs Setup*

10.6.1 nffs Setup

Q: I'm struggling to find any examples for `nffs`, especially how do I setup the `nffs_area_desc` correctly. Where do I set it up in the BSP especially?

A: It's all taken care of in `nffs_pkg_init`. As long as the `nffs` package is included in the project, it should initialize itself. A few things you might find helpful:

1. The `NFFS_FLASH_AREA` syscfg setting specifies the flash area that contains the file system.
2. The BSP's `bsp.xml` file defines all the flash areas in the system, including the one specified in "1." above.

10.7 Mynewt FAQ - Hardware-Specific Questions

- *Nordic nRF52-DK*
 - *Inverted Pin Value on nRF52-DK*
 - *Time Precision on nRF52*
 - *Size Limit on Transaction Length for nRF52*
- *Redbear BLE Nano 2*
 - *View Logs on the Redbear BLE Nano2*

10.7.1 Nordic nRF52-DK

Inverted Pin Value on nRF52-DK

Q: I've been experiencing what seems to be some oddities with `hal_gpio_write`. It appears as though the LED pin value on the nRF52-DK is inverted (0 sets the pin high, 1 sets it low). I am checking the GPIO state by turning an LED on and off. Why is this the case?

A: LEDs on the nRF52-DK are connected to VDD and GPIO so you need to set GPIO to a low state in order to make it turn on.

Time Precision on nRF52

Q: Can OS_TICKS_PER_SEC be changed per app? I'm on the nRF52 and I need better time precision than 128Hz.

A: No, it isn't possible to change the ticks per second for a single app. That constant is defined to be most efficient for the particular MCU.

If you need precision, the OS tick timer is probably not the right thing to use. Take a look at [OS CPU Time](#) for timer documentation. `os_cputime` has 1MHz frequency by default, and is enabled by default. It is recommended to use this for higher precision applications.

Size Limit on Transaction Length for nRF52

Q: There appears to be a 256-byte size limit on the maximum transaction length that the nRF52xxx `hal_i2c` driver can support. When I try send transactions larger than this the transactions fail (even after playing with a larger timeout etc). Does anyone know if this limit is due to the way the driver is written or the nRF52's i2c peripheral itself?

A: There shouldn't be any limit here since TWI works basically by transferring data byte after byte. Check your slave device to see if it has some limit.

10.7.2 Redbear BLE Nano 2

View Logs on the Redbear BLE Nano2

Q: Is it possible to see debug statements / logs while using Mynewt on the Redbear BLE Nano2. If so, can someone please tell me how to do so?

A: The RedBear daplink board presents multiple USB devices, one is a serial port which you can connect to. The [Blinky example project](#) has debug log statements which you can look at for how to log.

10.8 Mynewt FAQ - Syntax, Semantics, Configuration

- [*Floating Point in Mynewt*](#)
- [*Ending the Delay of a Task Blocking a Call Early*](#)
- [*Random Function / Device*](#)
- [*Setting serial and mfghash*](#)
- [*Leading Zeros Format in printf*](#)
- [*Mynewt Equivalent of UNIX sleep\(3\)*](#)
- [*Alternatives to cmsis_nvic.c*](#)

10.8.1 Floating Point in Mynewt

Q: I am trying to print floating point in Mynewt using `console_printf("%f", floating_var)` but I am unable to do so. How do I resolve the issue? I'm aware Mynewt uses baselibc - does it even support floating point?

A: Baselibc does support floating point formatting, but it is not enabled by default. To enable it, set the following syscfg setting to 1 in your target: `FLOAT_USER`

However, baselibc's float printf support is a bit limited. In particular, it ignores precision specifiers and always prints three digits after the decimal point.

10.8.2 Ending the Delay of a Task Blocking a Call Early

Q: I have a task which is blocking on a call to `os_time_delay()`. What is the recommended way to end the delay early in an ISR (e.g. button press)?

A: The best way would be to use a semaphore. Initialize the semaphore with a count of 0 (`os_sem_init()`), then block on the semaphore with the maximum delay you want to wait for (`os_sem_pend()`). The button press event would wake the first task up early by calling `os_sem_release()`.

10.8.3 Random Function / Device

Q: Does Mynewt have a random function or random device?

A: baselibc has `rand()`, and crypto/tinycrypt has `hmac-prng`.

10.8.4 Setting serial and mfghash

Q: What is mfghash? How do I set serial and mfghash (currently blank in my app)?

A: `mfghash` is computed if you're using `newt mfg` to construct your flash image, and it identifies the build of your bootloader. `newt mfg` bundles together the bootloader, target image, and other data you'd want to bundle when creating an image to burn to flash. See the `newt mfg` [documentation](#) for the construction side of things and `apache-mynewt-core/sys/mfg/src/mfg.c` for the firmware side. `serial` was intended to be used if you want to have your own naming scheme per device when building products; i.e. you want something other than the mcu serial number, or if you don't have serial number available.

10.8.5 Leading Zeros Format in printf

Q: Is there a way to make printf and console_printf honor the leading zeros format? As in:

```
console_printf("%.2d", 5);
```

outputting "05" instead of as for me now: "2d" ?

```
A: console_printf("%02d", 5);
```

10.8.6 Mynewt Equivalent of UNIX sleep(3)

Q: Is there an equivalent to the UNIX sleep(3)?

A: `os_time_delay(OS_TICKS_PER_SEC * secs)`

10.8.7 Alternatives to `cmsis_nvic.c`

Q: What do I use instead of the full version of `cmsis_nvic.c` (i.e. for setting and getting IRQ priorities)?

A: Those functions are in the `core_cmx.h` files in `hw/cmsis-core`.

10.9 Mynewt FAQ - NFC

The NFC stack is work in progress.

10.10 Mynewt FAQ - Newt

- *newt size Command vs. Elf File Size*

10.10.1 newt size Command vs. Elf File Size

Q: I did a test build of blinky for nrf52 and got an elf-file of size 295424 bytes. If I use the newt size command for the application it says something like: 18764 bytes. What does this mean?

A: Elfs have a lot of extra information. newt size will show the are in flash that is used which better matches the `blinky.elf.bin` file. Try running `newt -ldebug build -v <your-target>` and you will see something like this:

```
arm-none-eabi-objcopy -R .bss -R .bss.core -R .bss.core.nz -O binary ...
```

10.11 Mynewt FAQ - Newt Manager

- *Connection Profile with newtmgr*
- *NMP*
- *Communicate with a Device based on peer_id*
- *Newt Manager with the Adafruit nRF52DK*

10.11.1 Connection Profile with newtmgr

Q: I'm trying to connect to an Adafruit nRF52 Feather Pro running Mynewt via the newtmgr tool on MacOS. I have the device powered via micro USB to my Mac. How do I find the "connection profile" of the device so I can connect to it? I want to communicate over BLE and not serial.

A: A connection profile tells newtmgr how to communicate with your device. You can create one using the `newtmgr conn add` command. Try talking to your device without a connection profile first. If that works, you can create a profile to make it easier to communicate with the device going forward.

For BLE, you can send an echo command to your device with something like this:

```
newtmgr --conntype ble --connstring peer_name=nimble-bleprph echo Hello
```

That `peer_name` string is correct if your device is running the `bleprph` app. You'll need to adjust it if your device has a different BLE name. The `--conntype ble --connstring peer_name=nimble-bleprph` part is what would go in a connection profile. If you create one, then you can just specify the profile's name rather than typing that long string each time you send a command.

10.11.2 NMP

Q: What does NMP stand for?

A: Newtmgr Management Protocol

10.11.3 Communicate with a Device based on peer_id

Q: How do I communicate with a device using the newtmgr tool based on `peer_id`?

A: The command to use is:

```
newtmgr -c myprofile --connstring peer_id=aa:bb:cc:dd:ee:ff image list
```

Where device address has to be lowercase. You can also create connection profile so you do not need to specify address each time:

```
newtmgr conn add mypeerprofile type=ble connstring="peer_id=aa:bb:cc:dd:ee:ff:"
```

Afterwards, you can then use:

```
newtmgr -c mypeerprofile image list
```

Just be aware that `peer_id` has an OS-specific meaning. In Linux, it is the peer's Bluetooth address. In macOS, it is the UUID that the OS chooses to assign to the peer.

If you prefer to forgo connection profiles and see everything on the command line here is an alternative method:

```
newtmgr --conntype ble --connstring peer_id=aa:bb:cc:dd:ee:ff image list
```

10.11.4 Newt Manager with the Adafruit nRF52DK

Q: I'm having issues using Newt Manager with the Adafruit nRF52DK. What do I do?

You can specify the reduced MTU by adding `mtu=128` to your connection string. The reason for this change is that MTU is the serial boot loader used to have a smaller receive buffer (128 bytes). The newtmgr tool sends larger image chunks by default, so specifying the MTU will reduce the image size.

A: There are two things you will need to do to fix any issues you encounter when working with the Adafruit nRF52DK and Newt Manager:

1. Specify a reduced MTU: You can specify the reduced MTU by adding `mtu=128` to your connection string. The reason for this change is that MTU is the serial boot loader used to have a smaller receive buffer (128 bytes). The newtmgr tool sends larger image chunks by default, so specifying the MTU will reduce the image size.

2. Indicate that the existing image should not be erased: This is accomplished with the `-e` command line option. Your command line should look similar to the following:

```
$ newtmgr --conntype serial --connextra 'dev=/dev/ttyUSB0,mtu=128' image upload -e  
→<image-path>
```

This change is needed because the serial boot loader doesn't support the standalone "erase image" command - as a result, it drops the request. The newtmgr image upload command starts by sending an erase command, then times out when it doesn't receive a response. The older version of newtmgr would use smaller chunk size for images, and it did not send the standalone erase command. When newtmgr was changed in versions 1.2 and 1.3, the serial boot loader changed along with it. The latest newtmgr is not compatible with an older version of the boot loader (which your board will probably ship with) without the above workarounds.

10.12 Mynewt FAQ - Porting Mynewt

- *Porting Mynewt to Core-M3 MCU*

10.12.1 Porting Mynewt to Core-M3 MCU

Q: I have a weird OS tick issue with a Core-M3 MCU port. The tick rate is set up identically to most ARM MCUs by setting up a hardware interrupt to trigger SysClock / `os_tick_per_sec`. SysClock is correct and `os_tick_per_sec` is set to 1000, but the tick rate seems to be significantly higher. What am I doing wrong?

A: Check whether the LED is actually staying on or it is flickering really fast by debugging through the loop. If it is staying on, you may be getting into an `assert()`. Otherwise, it is due to the fact that the OS timer wasn't created, which is done by `hal_bsp.c`. The OS timer needs a hardware timer to be running, so you will need to call `hal_timer_init` for timer 0 at one point.

10.13 Mynewt FAQ - Troubleshooting/Debugging

- *Concurrent debug sessions with two boards*
- *Error: Unsatisfied APIs detected*
- *Greyed out files on iOS 11*
- *arm-none-eabi-gcc Build Error for Project*
- *Issues Running Image on Boot*
- *Enable Trace in Mynewt*
- *Bug With Older Versions of gcc*

10.13.1 Concurrent debug sessions with two boards

Q: Can I run concurrent GDB connections to multiple devices? Is the GDB port hardcoded?

A: Yes, starting with release 1.5 Mynewt allows “newt debug” to have a session with JLink as well as openocd simultaneously open. And no, the GDB port is not hardcoded. You can specify it with the -port option. For example:

```
newt debug slinky_nrf52 --extrajtacmd "-port 5431 -select usb=682148664"
```

10.13.2 Error: Unsatisfied APIs detected

Q: I ran into the following error message:

```
Error: Unsatisfied APIs detected:  
* stats, required by: hw/drivers/sensors/bmp280, hw/drivers/sensors/ms5837, net/oic
```

A: How do I resolve this?

A: You may be missing some package dependencies in your `pkg.yml` file. In this case, you need to include `sys/stats` (either `sys/stats/full` or `sys/stats/stub`) to your `pkg.yml` file. You can add it to either your app’s or target’s `pkg.yml` file, but if you have a custom app it is recommended that you add it to your app’s `pkg.yml`. That way you can have multiple targets for the same app, without having to add it to every target. Moreover, if you share your app package, others won’t run into the same error when building it.

10.13.3 Greyed out files on iOS 11

Q: I’m trying to use the Adafruit Mynewt Manager to upload a custom image over BLE. Uploading one of the provided `bleuartx000.img` works fine and I can boot into them, confirm etc. However, when I try to upload a custom image I can’t even seem to add it to the app. Images stored in the iCloud drive just appear as disabled icons. Anyone with a clue as to how to get that working?

A: The new iOS version no longer allows files with unrecognized extensions to be selected. Try renaming the file to something more compatible (e.g. `.txt`).

10.13.4 arm-none-eabi-gcc Build Error for Project

Q: I am having this error when I try to build my project:

```
Building target targets/stm32l072czy6tr_boot
Error: exec: "arm-none-eabi-gcc": executable file not found in $PATH
```

How do I add it?

A: First, install the GNU Arm Embedded Toolchain if you haven't already. Then, depending on your OS, add the link to your arm-none-eabi-gcc executable path to your PATH environment variable.

10.13.5 Issues Running Image on Boot

Q: I was able to successfully create a BSP for my custom board (using nRF52 MCU), then build and run that image in the debugger. However, it does not run on boot. Any ideas to fix the issue?

A: A good process in general is to do a full flash erase, then flash the bootloader and the running image. Make sure to dump the contents of flash and see that it actually gets written there as well. If you experience the issue again after a reboot, you will also want to set MCU_DCDC_ENABLED:0 then redo the process of erase, rebuild, and reload.

10.13.6 Enable Trace in Mynewt

Q: I'm trying to use gdb with Trace, but do not know how to enable it. How do I do this in Mynewt?

A: To enable Trace, you can add cflags to pkg.yml in your target directory:

```
~/dev/mynewt $ cat targets/mytarget/pkg.yml
### Package: targets/mytarget
pkg.name: "targets/mytarget"
pkg.type: "target"
pkg.description:
pkg.author:
pkg.homepage:
pkg.cflags:
- -DENABLE_TRACE
```

10.13.7 Bug With Older Versions of gcc

Q: I got the following error using newt build. How do I fix it?

```
Error: repos/apache-mynewt-core/sys/log/modlog/src/modlog.c: In function 'modlog_alloc':
repos/apache-mynewt-core/sys/log/modlog/src/modlog.c:61:23: error: missing braces around
↳ initializer [-Werror=missing-braces]
    *mm = (struct modlog_mapping) { 0 };
               ^
repos/apache-mynewt-core/sys/log/modlog/src/modlog.c:61:23: error: (near initialization
↳ for '(anonymous).next') [-Werror=missing-braces]
cc1: all warnings being treated as errors
```

A: That is a bug in older versions of gcc (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=53119). The recommended gcc version is 7.x.

11.1 Contributing to Newt or Newtmgr Tools

Newt and Newtmgr are written in Go (golang). This guide shows you how to install Go and setup your environment to update and build the tools if you want to:

- Build the tools with latest updates from the master branch on Linux or Windows platforms. **Note:** For Mac OS, you can use the `brew install mynewt-newt -HEAD` and the `brew install mynewt-newtmgr --HEAD` commands.
- Contribute to newt or newtmgr features or fix bugs.

This guide shows you how to perform the following:

1. Install Mac OS X, Linux, Windows. (Tasks that are specific to each platform are called out.)
2. Setup the Go environment.
3. Download the source, build, and install the newt or newtmgr tools.
4. Update and rebuild the tools.

- *Step 1: Installing Go*
 - *Installing Go on Mac OS X*
 - *Installing Go and git on Linux*
 - *Installing Go on and git Windows*
- *Step 2: Setting Up Your Go Environment*
- *Step 3: Downloading the Source and Installing the Tools*
 - *Downloading and Installing the Newt Tool*
 - *Downloading and Installing the Newtmgr Tool*
- *Step 4: Updating and Rebuilding the Tools:*

Note: You will also need to read and follow the instructions from the [Mynewt FAQ](#) to set up your git repos to submit changes.

11.1.1 Step 1: Installing Go

The latest master branch of newt and newtmgr requires GO version 1.7.6 or higher. You can skip this step and proceed to Step 2 if you already have Go version 1.7.6 or higher installed.

Installing Go on Mac OS X

If you do not have Homebrew installed, run the following command. You will be prompted for your sudo password.

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

You can also extract (or `git clone`) Homebrew and install it to /usr/local.

Use brew to install Go:

```
$ brew install go
==>
...
==> Summary
/usr/local/Cellar/go/1.8.3: 7,035 files, 282.0MB
```

You can also download the Go package directly from (<https://golang.org/dl/>) and install it in /usr/local/bin instead of brewing it.

Installing Go and git on Linux

```
sudo apt-get install golang git
```

Installing Go on and git Windows

In msys2 terminal type

```
pacman -S mingw-w64-x64_64-go git
```

Make sure to restart msys2 terminal after installation.

11.1.2 Step 2: Setting Up Your Go Environment

This section describes the Go environment and how to setup a Go workspace. If you already have a Go workspace for your other Go projects, you can skip this step and proceed to Step 3.

Go provides an environment to compile Go code, construct Go packages, and import Go code. You will use Go commands to import the newt or newtmgr package repository into your local Go environment. The Go language environment dictates a specific directory structure, or workspace in Go parlance. It must contain three sibling directories with the names **src**, **pkg** and **bin**:

- **src** contains Go source files organized into packages (one package per directory)
- **pkg** contains package objects
- **bin** contains the Go application executables that Go builds and installs.

The **GOPATH** environment variable specifies the location of your workspace. To setup this workspace environment, create a **dev** directory and then a **go** directory under it. Set the **GOPATH** environment variable to this directory where you will clone the newt and newtmgr repositories.

```
$ cd $HOME
$ mkdir -p dev/go
$ cd dev/go
$ export GOPATH=`pwd`
```

Add the following export statements to your `~/.bash_profile` file and source the file:

```
export GOPATH=$HOME/dev/go
export PATH=$GOPATH/bin:$PATH
```

11.1.3 Step 3: Downloading the Source and Installing the Tools

Newt and newtmgr are individual Go packages and have their own git repositories. You can download the source and install one or both tools.

We use the `go get` command to download the source, build, and install the binary in the **\$GOPATH/bin** directory.

Downloading and Installing the Newt Tool

The newt Go package is mynewt.apache.org/newt/newt and is stored in the Apache Mynewt newt tool repository mirrored on github.

Download the newt package source and install the tool:

```
$ cd $GOPATH
$ go get mynewt.apache.org/newt/newt
$ cd $GOPATH/src/mynewt.apache.org/newt
$ ls
DISCLAIMER      RELEASE_NOTES.md    util
INSTALLING.md   build.sh        viper
LICENSE         newt           yaml
NOTICE          newtmgr
README.md       newtvn
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Note: The source code under the **newtmgr** directory is no longer used or updated. The current **newtmgr** source has its own Git repository.

Check that the newt binary is installed and you are using the one from **\$GOPATH/bin**:

```
$ ls $GOPATH/bin/newt
~/dev/go/bin/newt
$ which newt
~/dev/go/bin/newt
$ newt version
Apache Newt version: 1.1.0-dev
```

Downloading and Installing the Newtmgr Tool

The newtmgr Go package is mynewt.apache.org/newtmgr/newtmgr. It is stored in the Apache Mynewt newtmgr tool repository mirrored on [github](#).

Download the newtmgr package and install the tool:

```
$ cd $GOPATH
$ go get mynewt.apache.org/newtmgr/newtmgr
$ cd $GOPATH/src/mynewt.apache.org/newtmgr
$ ls
LICENSE      NOTICE      README.md    newtmgr      nmxact
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Check that the newtmgr binary is installed and you are using the one from **\$GOPATH/bin**:

```
$ ls $GOPATH/bin/newtmgr
~/dev/go/bin/newtmgr
$ which newtmgr
~/dev/go/bin/newtmgr
```

11.1.4 Step 4: Updating and Rebuilding the Tools:

This section shows you how to rebuild the newt and newtmgr tools with the latest updates from the master branch or after you have made changes in your branch.

Here is the general procedure to rebuild either the newt or newtmgr tool. The only difference is the directory where you will be executing the commands from. You will need to repeat the procedure to rebuild both tools.

1. Change to the directory where the local Git repository for the tool is installed.
2. Pull the latest changes from the master branch. If you made changes you will need to rebase with **origin master** (See [Mynewt FAQ](#)).
3. Build and install the tool.

Change to the directory where the source for the tool is installed.

For the **newt** tool:

```
$ cd $GOPATH/src/mynewt.apache.org/newt/newt
```

For the **newtmgr** tool:

```
$ cd $GOPATH/src/mynewt.apache.org/newtmgr/newtmgr
```

After you change to the specific tool directory, get the latest updates from the master branch. If you made changes and need to rebase with the origin, add the `--rebase origin master` arguments to the `git pull` command:

```
$ git pull
```

Build and install the tool. The updated binary will be installed in the **\$GOPATH/bin** directory:

```
$ go install
```

You can run the `ls -l` command to check the modification time for the binary to ensure the new version is installed.

11.2 Developing Mynewt Applications with Visual Studio Code

This guide shows you how to set up Visual Studio Code to develop and debug Mynewt applications. Visual Studio Code is supported on Mac OS, Linux, and Windows. This guide shows you how to:

1. Install Visual Studio Code.
2. Install the C/C++ and debugger extensions.
3. Define task configurations to build Mynewt applications.
4. Define debugger configurations to debug Mynewt applications.
5. Launch the debugger.

- *Installing Visual Studio Code*
- *Installing the C/C++ and Debugger Extensions*
- *Defining Tasks for Mynewt Projects*
 - *Associating a Mynewt Project to a Workspace*
 - *Defining Visual Studio Code Tasks to Build and Debug Mynewt Applications*
 - *Running a Task*
 - *Defining Tasks for Other Newt Commands*
- *Defining Debugger Configurations*
- *Debugging Your Application*
- *Working with Multiple Mynewt Applications*

Prerequisites:

- Have Internet connectivity to fetch remote Mynewt components.
- Have a computer to build a Mynewt application.
- Perform *native installation* for the Mynewt tools and toolchains. **Note:** For Windows platforms, ensure that the MinGW bash you install is added to your Windows Path. In addition, if you are using Windows 10 WSL, you must have the MinGW bash before the Windows 10 WSL bash in your Windows Path.
- Read the Mynewt OS Concepts section.
- Create a project space (directory structure) and populate it with the core code repository (`apache-mynewt-core`) or know how to as explained in Creating Your First Project.
- Complete one of the *Blinky Tutorials*.

Notes:

- This guide is not a tutorial for Visual Studio Code. It assumes you are familiar with Visual Studio Code. If this is your first time using Visual Studio Code, we recommend that you read the Visual Studio Code [documentation and tutorials](#) and evaluate whether you would like to use it to develop Mynewt applications.

- This guide uses Visual Studio Code on Windows. Visual Studio Code is supported on Linux and Mac OS but may have some variations in the keyboard shortcuts and command names for these platforms.
- You can also use the Eclipse IDE to develop Mynewt applications. See [hacking-mynewt-in-eclipse](#) for more details. On Windows platforms, you must also ensure the MinGW bash is set in your Windows Path as described in the prerequisites.

11.2.1 Installing Visual Studio Code

Download and install Visual Studio Code from <https://code.visualstudio.com/>.

11.2.2 Installing the C/C++ and Debugger Extensions

You need to install two extensions:

1. The C/C++ extension from Microsoft. This extension provides language support such as symbol searching, signature help, go to definition, and go to declaration.
2. The Native Debug extension from webfreak. This extension provides GDB support.

To install the C/C++ extension:

1. Press **Ctrl-P** to open the search box.
2. Type `ext install cpptools` in the search box and press Enter. You should see the extension at the top of the list.
3. Click **Install** to install the extension.

To install the Native Debugger:

1. Press **Ctrl-P** to open the search box.
2. Type `ext install webfreak.debug` in the search box and press Enter. You should see the Native Debug extension at the top of the list.
3. Click **Install** to install the extension.

11.2.3 Defining Tasks for Mynewt Projects

Two main concepts in Visual Studio Code are workspaces and tasks. A workspace represents a folder that is open. You can open multiple workspaces and switch between workspaces.

Tasks allow you to integrate the external tools and operations that are used to build or test your project into Visual Studio Code. Tasks are run from and the task results can be analyzed in Visual Studio Code. Tasks are defined within the scope of a workspace. This means that the tasks you define for a workspace only apply to the given workspace.

Associating a Mynewt Project to a Workspace

For your Mynewt project, your Visual Studio Code workspace is the Mynewt project base directory. For example, if you create a project named `myproj` under the `~/dev` directory, then you open the `~/dev/myproj` folder for your workspace.

Select **File > Open Folder**, and select the `myproj` folder from the **Select Folder** dialog box to open the folder.

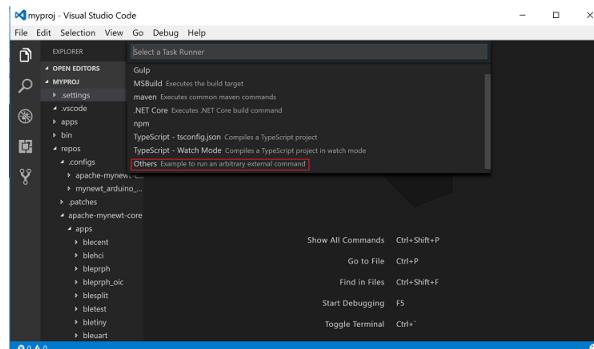
Defining Visual Studio Code Tasks to Build and Debug Mynewt Applications

You define Visual Studio Code tasks to build and debug your Mynewt targets in Visual Studio Code. We use the Blinky application for the Arduino Zero board from the [Blinky On Arduino Zero Tutorial](#) to illustrate how to define the tasks to build and debug the Arduino blinky bootloader and application targets.

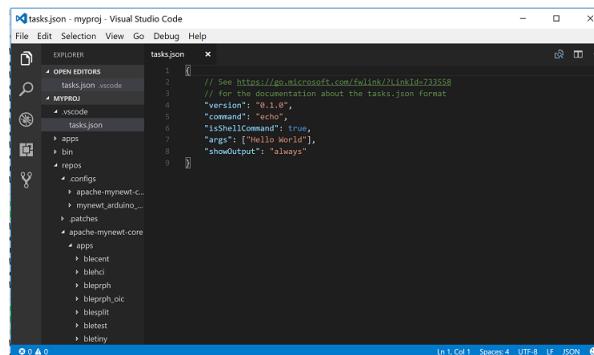
Perform the following steps to create the tasks to build and debug the Arduino blinky bootloader and application targets:

Step 1: Press **Ctrl+Shift+P**, type **task**, and select **Tasks:Configure Task Runner** from the search results.

Step 2: Select **Others** (scroll down to the bottom of the list) to create a task runner for external commands.



Tasks are defined in the `tasks.json` file. You should see the `.vscode` folder created in the `MYPROJ` folder and a `tasks.json` file created in the `.vscode` folder. The `tasks.json` file has the following default values.



The sample `tasks.json` file defines a simple task that runs the echo command with “Hello World” as the argument.

Step 3: Delete the content from the `tasks.json` file, add the following definitions, and press **Ctrl+S** to save the file.

```
{
  "version": "0.1.0",
  "command": "newt",
  "echoCommand": true,
```

(continues on next page)

(continued from previous page)

```

"isShellCommand": true,

"tasks": [
{
  "taskName": "build_arduino_boot",
  "args": ["build", "arduino_boot"],
  "suppressTaskName": true
},
{
  "taskName": "build_arduino_blinky",
  "args": ["build", "arduino_blinky"],
  "isBuildCommand": true,
  "suppressTaskName": true
},
{
  "taskName": "create_arduino_blinky",
  "args": ["create-image", "arduino_blinky", "1.0.0"],
  "suppressTaskName": true
},
{
  "taskName": "debug_arduino_blinky",
  "args": ["debug", "arduino_blinky", "-n"],
  "suppressTaskName": true
}
]
}

```

The `tasks.json` file specifies the tasks that are run to build and debug the Arduino blinky targets. Each task runs a `newt` command. The `newt` command to run and the arguments for the `newt` command are passed in the `args` property for each task.

The following tasks are defined in this example:

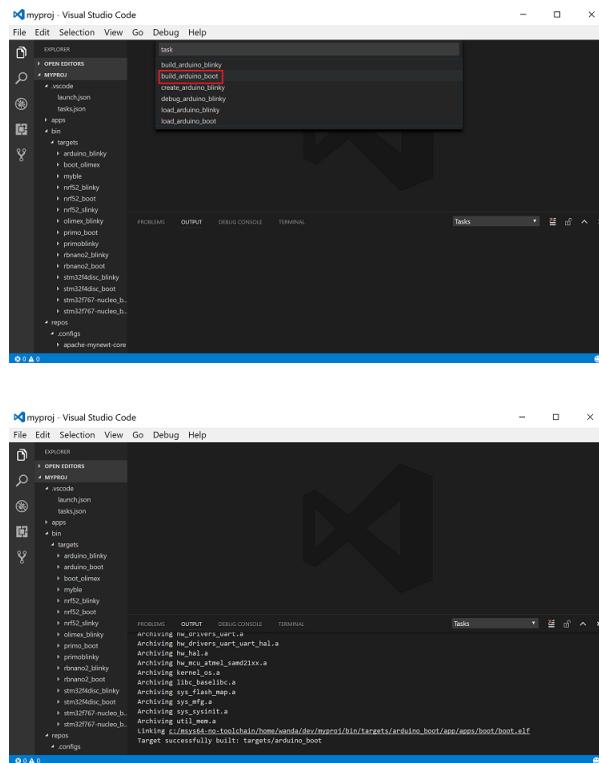
1. **build_arduino_boot**: Runs the `newt build arduino_boot` command to build the `arduino_boot` target.
 2. **build_arduino_blinky**: Runs the `newt build arduino_blinky` command to build the `arduino_blinky` target.
- Note:** This task sets the `isBuildCommand` property to `true`. This is an optional property that, when set to `true`, allows you to run the **Tasks: Run Build Task**(`Ctrl-Shift-B`) command to start the task.
3. **create_arduino_blinky**: Runs the `newt create-image arduino_blinky` command to create the image file.
 4. **debug_arduino_blinky**: Runs the `newt build arduino_blinky -n` command to debug the `arduino_blinky` target. The `-n` flag is specified to start only the GDB server and not the GDB client. We will launch the GDB client from Visual Studio Code.

For more information on tasks and all supported properties, see the [Visual Studio Code Task documentation](#).

Running a Task

To run a task, press **Ctrl-Shift-P**, type **task** on the search box, and select **Tasks: Run Task**. The tasks that you define in the `tasks.json` file are listed. Select the task to run.

The following is an example of running the `build_arduino_boot` task:



Note: To run the `build_arduino_blinky` task, you can use the keyboard shortcut **Ctrl-Shift-B** because the task has the property `isBuildCommand` set to true.

Defining Tasks for Other Newt Commands

Other newt commands, such as the `newt load` command, do not need to run from within Visual Studio Code. You can define a task for each command as a convenience and run the command as a task, or you can run the `newt` command on the command line from the Visual Studio Code integrated terminal or an external terminal.

To create the tasks for the `newt load arduino_boot` and `newt load arduino_blinky` commands, add the following definitions to the `tasks.json` file:

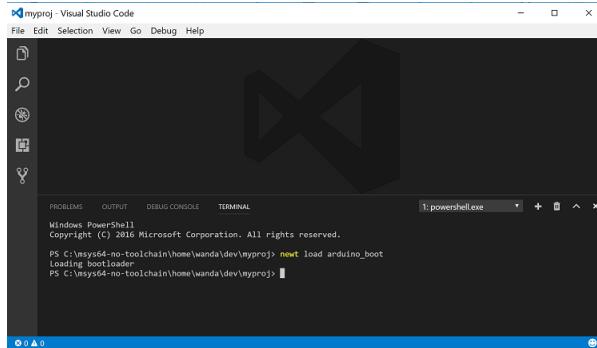
```
[  
  {  
    "taskName": "load_arduino_boot",  
    "args": ["load", "arduino_boot"],  
    "suppressTaskName":true  
  },  
  {  
    "taskName": "load_arduino_blinky",  
    "args": ["load", "arduino_blinky"],  
    "suppressTaskName":true  
  }]
```

(continues on next page)

(continued from previous page)

```
    },
]
```

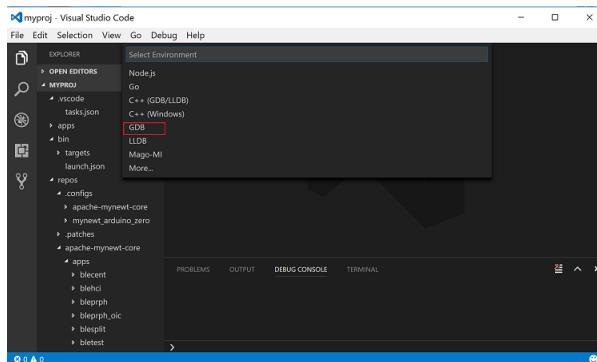
To run a command from the Visual Studio integrated terminal, instead of starting a task, press **Ctrl-`** to launch the integrated terminal and enter the command on the prompt:



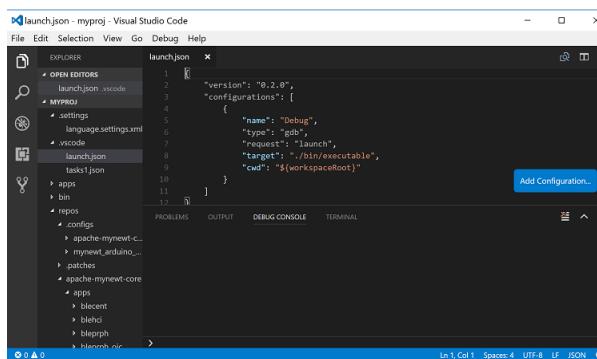
11.2.4 Defining Debugger Configurations

You need to define a debugger configuration to launch the GDB debugger from within Visual Studio Code:

Step 1: Select **Debug > Open Configuration**, and select the **GDB** environment.



You should see a default `launch.json` file created in the `.vscode` folder.



Step 2: Delete the content from the `launch.json` file, add the following definitions, and press ‘**Ctrl-S**’ to save the file.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "gdb_arduino_blinky",
      "type": "gdb",
      "request": "attach",
      "executable": "${workspaceRoot}\\bin\\targets\\arduino_blinky\\app\\apps\\
      ↵blinky\\blinky.elf",
      "target": ":3333",
      "cwd": "${workspaceRoot}",
      "gdbpath": "C:\\Program Files (x86)\\GNU Tools ARM Embedded\\4.9 2015q2\\bin\\
      ↵\\arm-none-eabi-gdb.exe",
      "remote": true
    }
  ]
}
```

This defines a `gdb_arduino_blinky` debugger configuration. It specifies:

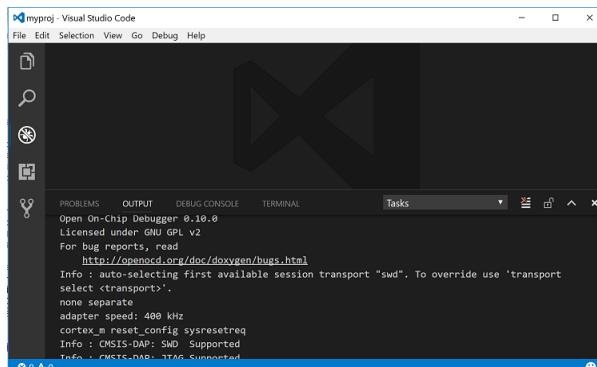
- The debugger is type **gdb**.
- To use the `blinky.elf` file for the executable.
- To use port 3333 to connect with the remote target.
- To use `arm-none-eabi-gdb` for the GDB program.

11.2.5 Debugging Your Application

To debug your application, start the GDB server and launch the GDB session from Visual Studio Code. For the arduino blinky example, perform the following:

Step 1: Run the `debug_arduino_blinky` task to start the GDB server. Perform the following:

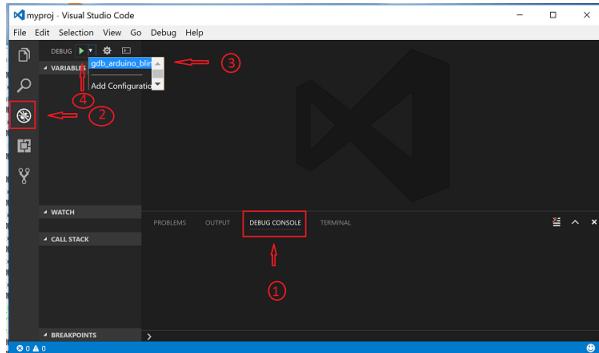
1. Press **Ctrl-Shift-P** and type **task** in the search box.
2. Select **Tasks:Run Task > debug_arduino_blinky**.
3. Press **Ctrl-Shift-U** to open the Output Panel and see the OpenOCD GDB Server output.



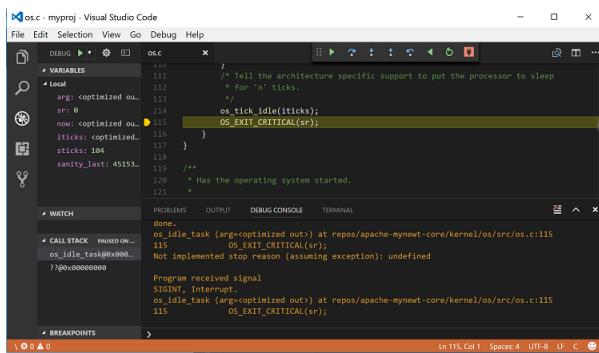
Step 2: Start the GDB session. Perform the following:

1. Press **Ctrl-Shift-Y** to view the Debug Console.

2. Press the Debugging icon on the activity bar (Ctrl-Shift-D) to bring up the Debug Side Bar.
3. Select gdb_arduino_blinky from the DEBUG drop down menu.
4. Press the green play button to start the gdb session.



Step 3: Debug your application. You should see a debug session similar to the one shown below:



For more information on how to use the Visual Studio Code Debugger, see the [Visual Studio Code debugging documentation](#).

11.2.6 Working with Multiple Mynewt Applications

As mentioned previously, each mynewt project corresponds to a Visual Studio Code workspace. If you have multiple Mynewt application targets defined in same project, you will need to define build and debug tasks for each target in the `tasks.json` file and debugger configurations for the targets in the `launch.json` file for the workspace. If you have a different Mynewt project for each mynewt application, you will need to define build and debug tasks in the `tasks.json` file and the debugger configuration in the `launch.json` file for each workspace.

INDEX

Symbols

_sbrk (*C function*), 427

A

area_id (*C var*), 575

ARG10 (*C macro*), 560

ASSERT_IF_TEST (*C macro*), 560

B

ble_gap_add_dev_to_periodic_adv_list (*C function*), 602

ble_gap_adv_active (*C function*), 596

BLE_GAP_DFLT_CHANNEL_MAP (*C macro*), 613

ble_gap_adv_get_free_instance (*C function*), 599

ble_gap_adv_params (*C struct*), 613

ble_gap_adv_params.channel_map (*C var*), 614

ble_gap_adv_params.conn_mode (*C var*), 614

ble_gap_adv_params.disc_mode (*C var*), 614

ble_gap_adv_params.filter_policy (*C var*), 614

ble_gap_adv_params.high_duty_cycle (*C var*), 614

ble_gap_adv_params.itvl_max (*C var*), 614

ble_gap_adv_params.itvl_min (*C var*), 614

ble_gap_adv_rsp_set_data (*C function*), 596

ble_gap_adv_rsp_set_fields (*C function*), 597

ble_gap_adv_set_data (*C function*), 596

ble_gap_adv_set_fields (*C function*), 597

ble_gap_adv_start (*C function*), 595

ble_gap_adv_stop (*C function*), 596

ble_gap_clear_periodic_adv_list (*C function*), 602

ble_gap_conn_active (*C function*), 605

ble_gap_conn_cancel (*C function*), 605

ble_gap_conn_desc (*C struct*), 614

ble_gap_conn_desc.conn_handle (*C var*), 615

ble_gap_conn_desc.conn_itvl (*C var*), 615

ble_gap_conn_desc.conn_latency (*C var*), 615

ble_gap_conn_desc.master_clock_accuracy (*var*), 615

ble_gap_conn_desc.our_id_addr (*C var*), 614

ble_gap_conn_desc.our_ota_addr (*C var*), 615

ble_gap_conn_desc.peer_id_addr (*C var*), 614

ble_gap_conn_desc.peer_ota_addr (*C var*), 615

ble_gap_conn_desc.role (*C var*), 615

ble_gap_conn_desc.sec_state (*C var*), 614

ble_gap_conn_desc.supervision_timeout (*C var*), 615

ble_gap_conn_find (*C function*), 595

ble_gap_conn_find_by_addr (*C function*), 595

ble_gap_conn_find_handle_by_addr (*C function*), 611

ble_gap_conn_FOREACH_HANDLE (*C function*), 611

ble_gap_conn_FOREACH_HANDLE_fn (*C type*), 595

ble_gap_conn_params (*C struct*), 615

ble_gap_conn_params.itvl_max (*C var*), 615

ble_gap_conn_params.itvl_min (*C var*), 615

ble_gap_conn_params.latency (*C var*), 615

ble_gap_conn_params.max_ce_len (*C var*), 616

ble_gap_conn_params.min_ce_len (*C var*), 616

ble_gap_conn_params.scan_itvl (*C var*), 615

ble_gap_conn_params.scan_window (*C var*), 615

ble_gap_conn_params.supervision_timeout (*C var*), 616

ble_gap_conn_rssi (*C function*), 608

ble_gap_connect (*C function*), 604

ble_gap_disc (*C function*), 602

ble_gap_disc_active (*C function*), 604

ble_gap_disc_cancel (*C function*), 604

ble_gap_disc_desc (*C struct*), 619

ble_gap_disc_desc.addr (*C var*), 620

ble_gap_disc_desc.data (*C var*), 620

ble_gap_disc_desc.direct_addr (*C var*), 620

ble_gap_disc_desc.event_type (*C var*), 620

ble_gap_disc_desc.length_data (*C var*), 620

ble_gap_disc_desc.rssi (*C var*), 620

ble_gap_disc_params (*C struct*), 616

ble_gap_disc_params.filter_duplicates (*C var*), 617

ble_gap_disc_params.filter_policy (*C var*), 616

ble_gap_disc_params.itvl (*C var*), 616

ble_gap_disc_params.limited (*C var*), 616

ble_gap_disc_params.passive (*C var*), 616

ble_gap_disc_params.window (*C var*), 616

ble_gap_encryption_initiate (*C function*), 607

ble_gap_enh_read_transmit_power_level (*C func-*

tion), 612
ble_gap_event (*C struct*), 621
ble_gap_event.type (*C var*), 621
ble_gap_event_fn (*C type*), 595
ble_gap_event_listener (*C struct*), 624
ble_gap_event_listener.arg (*C var*), 625
ble_gap_event_listener.fn (*C var*), 625
ble_gap_event_listener_register (*C function*), 611
ble_gap_event_listener_unregister (*C function*), 611
ble_gap_ext_adv_active (*C function*), 599
ble_gap_ext_adv_clear (*C function*), 598
ble_gap_ext_adv_configure (*C function*), 597
BLE_GAP_EXT_ADV_DATA_STATUS_COMPLETE (*C macro*), 613
BLE_GAP_EXT_ADV_DATA_STATUS_INCOMPLETE (*C macro*), 613
BLE_GAP_EXT_ADV_DATA_STATUS_TRUNCATED (*C macro*), 613
ble_gap_ext_adv_params (*C struct*), 621
ble_gap_ext_adv_params.anonymous (*C var*), 622
ble_gap_ext_adv_params.channel_map (*C var*), 622
ble_gap_ext_adv_params.connectable (*C var*), 622
ble_gap_ext_adv_params.directed (*C var*), 622
ble_gap_ext_adv_params.filter_policy (*C var*), 623
ble_gap_ext_adv_params.high_duty_directed (*C var*), 622
ble_gap_ext_adv_params.include_tx_power (*C var*), 622
ble_gap_ext_adv_params.itvl_max (*C var*), 622
ble_gap_ext_adv_params.itvl_min (*C var*), 622
ble_gap_ext_adv_params.legacy_pdu (*C var*), 622
ble_gap_ext_adv_params.own_addr_type (*C var*), 622
ble_gap_ext_adv_params.peer (*C var*), 623
ble_gap_ext_adv_params.primary_phy (*C var*), 623
ble_gap_ext_adv_params.primary_phy_opt (*C var*), 623
ble_gap_ext_adv_params.scan_req_notif (*C var*), 622
ble_gap_ext_adv_params.scannable (*C var*), 622
ble_gap_ext_adv_params.secondary_phy (*C var*), 623
ble_gap_ext_adv_params.secondary_phy_opt (*C var*), 623
ble_gap_ext_adv_params.sid (*C var*), 623
ble_gap_ext_adv_params.tx_power (*C var*), 623
ble_gap_ext_adv_remove (*C function*), 598
ble_gap_ext_adv_rsp_set_data (*C function*), 598
ble_gap_ext_adv_set_addr (*C function*), 597
ble_gap_ext_adv_set_data (*C function*), 598
ble_gap_ext_adv_start (*C function*), 598
ble_gap_ext_adv_stop (*C function*), 598
ble_gap_ext_connect (*C function*), 604
ble_gap_ext_disc (*C function*), 603
ble_gap_ext_disc_desc (*C struct*), 618
ble_gap_ext_disc_desc.addr (*C var*), 618
ble_gap_ext_disc_desc.data (*C var*), 619
ble_gap_ext_disc_desc.data_status (*C var*), 618
ble_gap_ext_disc_desc.direct_addr (*C var*), 619
ble_gap_ext_disc_desc.legacy_event_type (*C var*), 618
ble_gap_ext_disc_desc.length_data (*C var*), 619
ble_gap_ext_disc_desc.periodic_adv_itvl (*C var*), 619
ble_gap_ext_disc_desc.prim_phy (*C var*), 619
ble_gap_ext_disc_desc.props (*C var*), 618
ble_gap_ext_disc_desc.rssi (*C var*), 618
ble_gap_ext_disc_desc.sec_phy (*C var*), 619
ble_gap_ext_disc_desc.sid (*C var*), 619
ble_gap_ext_disc_desc.tx_power (*C var*), 618
ble_gap_ext_disc_params (*C struct*), 616
ble_gap_ext_disc_params.itvl (*C var*), 616
ble_gap_ext_disc_params.passive (*C var*), 616
ble_gap_ext_disc_params.window (*C var*), 616
ble_gap_pair_initiate (*C function*), 607
ble_gap_passkey_params (*C struct*), 617
ble_gap_passkey_params.action (*C var*), 617
ble_gap_passkey_params.numcmp (*C var*), 617
ble_gap_periodic_adv_configure (*C function*), 599
ble_gap_periodic_adv_params (*C struct*), 623
ble_gap_periodic_adv_params.include_tx_power (*C var*), 624
ble_gap_periodic_adv_params.itvl_max (*C var*), 624
ble_gap_periodic_adv_params.itvl_min (*C var*), 624
ble_gap_periodic_adv_set_data (*C function*), 599
ble_gap_periodic_adv_set_data_params (*C struct*), 624
ble_gap_periodic_adv_start (*C function*), 599
ble_gap_periodic_adv_start_params (*C struct*), 624
ble_gap_periodic_adv_stop (*C function*), 599
ble_gap_periodic_adv_sync_create (*C function*), 600
ble_gap_periodic_adv_sync_create_cancel (*C function*), 600
ble_gap_periodic_adv_sync_receive (*C function*), 601
ble_gap_periodic_adv_sync_reporting (*C function*), 600
ble_gap_periodic_adv_sync_reporting_params (*C struct*), 624
ble_gap_periodic_adv_sync_set_info (*C function*), 601

ble_gap_periodic_adv_sync_terminate (*C function*), 600
 ble_gap_periodic_adv_sync_transfer (*C function*), 601
 ble_gap_periodic_sync_params (*C struct*), 624
 ble_gap_periodic_sync_params.reports_disabled (*C var*), 624
 ble_gap_periodic_sync_params.skip (*C var*), 624
 ble_gap_periodic_sync_params.sync_timeout (*C var*), 624
 ble_gap_read_le_phy (*C function*), 609
 ble_gap_read_periodic_adv_list_size (*C function*), 602
 ble_gap_read_remote_transmit_power_level (*C function*), 612
 ble_gap_read_sugg_def_data_len (*C function*), 606
 ble_gap_rem_dev_from_periodic_adv_list (*C function*), 602
 ble_gap_repeat_pairing (*C struct*), 620
 ble_gap_repeat_pairing.conn_handle (*C var*), 620
 ble_gap_repeat_pairing.cur_authenticated (*C var*), 620
 ble_gap_repeat_pairing.cur_key_size (*C var*), 620
 ble_gap_repeat_pairing.cur_sc (*C var*), 621
 ble_gap_repeat_pairing.new_authenticated (*C var*), 621
 ble_gap_repeat_pairing.new_bonding (*C var*), 621
 ble_gap_repeat_pairing.new_key_size (*C var*), 621
 ble_gap_repeat_pairing.new_sc (*C var*), 621
 ble_gap_sec_state (*C struct*), 613
 ble_gap_sec_state.authenticated (*C var*), 613
 ble_gap_sec_state.bonded (*C var*), 613
 ble_gap_sec_state.encrypted (*C var*), 613
 ble_gap_sec_state.key_size (*C var*), 613
 ble_gap_security_initiate (*C function*), 607
 ble_gap_set_data_len (*C function*), 606
 ble_gap_set_default_subrate (*C function*), 610
 ble_gap_set_event_cb (*C function*), 595
 ble_gap_set_path_loss_reporting_enable (*C function*), 612
 ble_gap_set_path_loss_reporting_param (*C function*), 613
 ble_gap_set_preferred_default_le_phy (*C function*), 609
 ble_gap_set_preferred_le_phy (*C function*), 610
 ble_gap_set_priv_mode (*C function*), 609
 ble_gap_set_transmit_power_reporting_enable (*C function*), 612
 ble_gap_subrate_req (*C function*), 611
 ble_gap_terminate (*C function*), 605
 ble_gap_unpair (*C function*), 608
 ble_gap_unpair_oldest_except (*C function*), 608
 ble_gap_unpair_oldest_peer (*C function*), 608
 ble_gap_upd_params (*C struct*), 617
 ble_gap_upd_params.itvl_max (*C var*), 617
 ble_gap_upd_params.itvl_min (*C var*), 617
 ble_gap_upd_params.latency (*C var*), 617
 ble_gap_upd_params.max_ce_len (*C var*), 617
 ble_gap_upd_params.min_ce_len (*C var*), 617
 ble_gap_upd_params.supervision_timeout (*C var*), 617
 ble_gap_update_params (*C function*), 606
 ble_gap_wl_read_size (*C function*), 606
 ble_gap_wl_set (*C function*), 606
 ble_gap_write_sugg_def_data_len (*C function*), 607
 ble_gatt_access_ctxt (*C struct*), 639, 655
 ble_gatt_access_ctxt.om (*C var*), 640, 656
 ble_gatt_access_ctxt.op (*C var*), 640, 656
 ble_gatt_access_fn (*C type*), 626, 641
 ble_gatt_attr (*C struct*), 637, 652
 ble_gatt_attr.handle (*C var*), 637, 653
 ble_gatt_attr.offset (*C var*), 637, 653
 ble_gatt_attr.om (*C var*), 637, 653
 ble_gatt_attr_fn (*C type*), 625, 641
 ble_gatt_attr_mult_fn (*C type*), 625, 641
 ble_gatt_chr (*C struct*), 637, 653
 ble_gatt_chr.def_handle (*C var*), 637, 653
 ble_gatt_chr.properties (*C var*), 637, 653
 ble_gatt_chr.uuid (*C var*), 637, 653
 ble_gatt_chr.val_handle (*C var*), 637, 653
 ble_gatt_chr_def (*C struct*), 638, 654
 ble_gatt_chr_def.access_cb (*C var*), 638, 654
 ble_gatt_chr_def.arg (*C var*), 638, 654
 ble_gatt_chr_def.descriptors (*C var*), 638, 654
 ble_gatt_chr_def.flags (*C var*), 638, 654
 ble_gatt_chr_def.min_key_size (*C var*), 638, 654
 ble_gatt_chr_def.uuid (*C var*), 638, 654
 ble_gatt_chr_def.val_handle (*C var*), 638, 654
 ble_gatt_chr_flags (*C type*), 626, 641
 ble_gatt_chr_fn (*C type*), 625, 641
 ble_gatt_disc_svc_fn (*C type*), 625, 641
 ble_gatt_dsc (*C struct*), 637, 653
 ble_gatt_dsc.handle (*C var*), 638, 653
 ble_gatt_dsc.uuid (*C var*), 638, 653
 ble_gatt_dsc_def (*C struct*), 639, 655
 ble_gatt_dsc_def.access_cb (*C var*), 639, 655
 ble_gatt_dsc_def.arg (*C var*), 639, 655
 ble_gatt_dsc_def.att_flags (*C var*), 639, 655
 ble_gatt_dsc_def.min_key_size (*C var*), 639, 655
 ble_gatt_dsc_def.uuid (*C var*), 639, 655
 ble_gatt_dsc_fn (*C type*), 625, 641
 ble_gatt_error (*C struct*), 636, 652
 ble_gatt_error.att_handle (*C var*), 636, 652
 ble_gatt_error.status (*C var*), 636, 652
 ble_gatt_mtu_fn (*C type*), 625, 641

ble_gatt_notif (*C struct*), 638, 653
ble_gatt_notif.handle (*C var*), 638, 654
ble_gatt_notif.value (*C var*), 638, 654
ble_gatt_register_ctxt (*C struct*), 640, 656
ble_gatt_register_ctxt.op (*C var*), 640, 656
ble_gatt_register_fn (*C type*), 626, 641
ble_gatt_reliable_attr_fn (*C type*), 625, 641
ble_gatt_svc (*C struct*), 636, 652
ble_gatt_svc.end_handle (*C var*), 637, 652
ble_gatt_svc.start_handle (*C var*), 637, 652
ble_gatt_svc.uuid (*C var*), 637, 652
ble_gatt_svc_def (*C struct*), 639, 654
ble_gatt_svc_def.characteristics (*C var*), 639, 655
ble_gatt_svc_def.includes (*C var*), 639, 655
ble_gatt_svc_def.type (*C var*), 639, 655
ble_gatt_svc_def.uuid (*C var*), 639, 655
ble_gatt_svc_foreach_fn (*C type*), 626, 641
ble_gattc_disc_all_chrs (*C function*), 627, 643
ble_gattc_disc_all_dscs (*C function*), 627, 643
ble_gattc_disc_all_svcs (*C function*), 626, 642
ble_gattc_disc_chrs_by_uuid (*C function*), 627, 643
ble_gattc_disc_svc_by_uuid (*C function*), 626, 642
ble_gattc_exchange_mtu (*C function*), 626, 642
ble_gattc_find_inc_svcs (*C function*), 626, 642
ble_gattc_indicate (*C function*), 633, 649
ble_gattc_indicate_custom (*C function*), 633, 649
ble_gattc_init (*C function*), 634, 649
ble_gattc_notify (*C function*), 632, 648
ble_gattc_notify_custom (*C function*), 631, 647
ble_gattc_read (*C function*), 628, 643
ble_gattc_read_by_uuid (*C function*), 628, 644
ble_gattc_read_long (*C function*), 628, 644
ble_gattc_read_mult (*C function*), 628, 644
ble_gattc_read_mult_var (*C function*), 629, 645
ble_gattc_write (*C function*), 630, 645
ble_gattc_write_flat (*C function*), 630, 646
ble_gattc_write_long (*C function*), 630, 646
ble_gattc_write_no_rsp (*C function*), 629, 645
ble_gattc_write_no_rsp_flat (*C function*), 629, 645
ble_gattc_write_reliable (*C function*), 631, 646
ble_gatts_add_svcs (*C function*), 634, 649
ble_gatts_chr_updated (*C function*), 634, 650
ble_gatts_count_cfg (*C function*), 634, 650
ble_gatts_find_chr (*C function*), 635, 650
ble_gatts_find_dsc (*C function*), 635, 651
ble_gatts_find_svc (*C function*), 634, 650
ble_gatts_indicate (*C function*), 633, 649
ble_gatts_indicate_custom (*C function*), 633, 648
ble_gatts_notify (*C function*), 632, 647
ble_gatts_notify_custom (*C function*), 631, 647
ble_gatts_notify_multiple (*C function*), 632, 648
ble_gatts_notify_multiple_custom (*C function*), 631, 647

ble_gatts_peer_cl_sup_feat_get (*C function*), 636, 651
ble_gatts_reset (*C function*), 635, 651
ble_gatts_show_local (*C function*), 635, 651
ble_gatts_start (*C function*), 636, 651
ble_gatts_svc_set_visibility (*C function*), 634, 650
BLE_HS_CONN_HANDLE_NONE (*C macro*), 660
ble_hs_evq_set (*C function*), 659
BLE_HS_FOREVER (*C macro*), 660
ble_hs_id_copy_addr (*C function*), 658
ble_hs_id_gen_rnd (*C function*), 657
ble_hs_id_infer_auto (*C function*), 658
ble_hs_id_set_rnd (*C function*), 657
ble_hs_init (*C function*), 659
ble_hs_is_enabled (*C function*), 659
ble_hs_sched_reset (*C function*), 659
ble_hs_sched_start (*C function*), 659
ble_hs_shutdown (*C function*), 660
ble_hs_start (*C function*), 659
ble_hs_synced (*C function*), 659

C

completion_cb (*C type*), 391
conf_bytes_from_str (*C function*), 359
conf_commit (*C function*), 359
conf_commit_handler_ext_t (*C type*), 356
conf_commit_handler_t (*C type*), 356
conf_dst_register (*C function*), 360
conf_ensure_loaded (*C function*), 357
conf_export (*C function*), 358
conf_export_func_t (*C type*), 356
conf_export_handler_ext_t (*C type*), 357
conf_export_handler_t (*C type*), 357
conf_export_tgt (*C enum*), 355
conf_export_tgt.CONF_EXPORT_PERSIST (*C enumerator*), 355
conf_export_tgt.CONF_EXPORT_SHOW (*C enumerator*), 355
conf_export_tgt_t (*C type*), 355
conf_get_handler_ext_t (*C type*), 356
conf_get_handler_t (*C type*), 355
conf_get_stored_value (*C function*), 359
conf_get_value (*C function*), 358
conf_handler (*C struct*), 361
conf_handler.ch_arg (*C var*), 361
conf_handler.ch_ext (*C var*), 361
conf_handler.ch_name (*C var*), 361
conf_init (*C function*), 357
conf_load (*C function*), 357
conf_load_one (*C function*), 357
conf_lock (*C function*), 360
conf_register (*C function*), 357
conf_save (*C function*), 358

conf_save_one (*C function*), 358
conf_save_tree (*C function*), 358
conf_set_from_storage (*C function*), 358
conf_set_handler_ext_t (*C type*), 356
conf_set_handler_t (*C type*), 356
conf_set_value (*C function*), 358
conf_src_register (*C function*), 360
conf_store (*C struct*), 361
conf_store_init (*C function*), 357
conf_store_itf (*C struct*), 361
conf_store_load_cb (*C type*), 357
conf_str_from_bytes (*C function*), 360
CONF_STR_FROM_BYTES_LEN (*C macro*), 360
conf_str_from_value (*C function*), 360
conf_type (*C enum*), 355
conf_unlock (*C function*), 360
conf_value_from_str (*C function*), 359
CONF_VALUE_SET (*C macro*), 361
console_append_char_cb (*C type*), 391
console_blocking_mode (*C function*), 392
console_deinit (*C function*), 392
console_echo (*C function*), 392
console_handle_char (*C function*), 392
console_ignore_non_nlip (*C function*), 392
console_init (*C function*), 392
console_input (*C struct*), 393
console_input.line (*C var*), 393
console_is_init (*C function*), 392
console_is_midline (*C var*), 393
console_line_event_put (*C function*), 392
console_line_queue_set (*C function*), 392
console_lock (*C function*), 392
console_non_blocking_mode (*C function*), 392
console_out (*C function*), 392
console_prompt_set (*C function*), 393
console_read (*C function*), 392
console_reinit (*C function*), 392
console_rx_cb (*C type*), 391
console_rx_restart (*C function*), 392
console_set_completion_cb (*C function*), 392
console_silence (*C function*), 392
console_silence_non_nlip (*C function*), 392
console_unlock (*C function*), 393
console_vprintf (*C function*), 392
console_write (*C function*), 392

D

device_id (*C var*), 575

F

fcb (*C struct*), 514
fcb.f_active_sector_entry_count (*C var*), 514
fcb_append (*C function*), 513
fcb_append_finish (*C function*), 513
fcb_append_to_scratch (*C function*), 513
fcb_area_info (*C function*), 513
fcb_cache_free (*C function*), 512
fcb_cache_init (*C function*), 512
fcb_clear (*C function*), 513
fcb_entry (*C struct*), 514
fcb_entry_cache (*C struct*), 514
FCB_ERR_ARGS (*C macro*), 514
FCB_ERR_CRC (*C macro*), 514
FCB_ERR_FLASH (*C macro*), 514
FCB_ERR_MAGIC (*C macro*), 514
FCB_ERR_NEXT_SECT (*C macro*), 514
FCB_ERR_NOMEM (*C macro*), 514
FCB_ERR_NOSPACE (*C macro*), 514
FCB_ERR_NOVAR (*C macro*), 514
FCB_ERR_VERSION (*C macro*), 514
fcb_free_sector_cnt (*C function*), 513
fcb_getnext (*C function*), 513
fcb_init (*C function*), 513
fcb_is_empty (*C function*), 513
FCB_MAX_LEN (*C macro*), 513
fcb_offset_last_n (*C function*), 513
FCB_OK (*C macro*), 513
fcb_rotate (*C function*), 513
fcb_walk (*C function*), 513
fcb_walk_cb (*C type*), 512
fcb_write (*C function*), 513
FIRST (*C macro*), 560
FIRST_AUX (*C macro*), 560
fops_container (*C struct*), 510
fops_container.fops (*C var*), 510
FS_ACCESS_APPEND (*C macro*), 501
FS_ACCESS_READ (*C macro*), 501
FS_ACCESS_TRUNCATE (*C macro*), 501
FS_ACCESS_WRITE (*C macro*), 501
fs_close (*C function*), 502
fs_closedir (*C function*), 502
fs_dirent_is_dir (*C function*), 502
fs_dirent_name (*C function*), 502
FS_EACCESS (*C macro*), 502
FS_ECORRUPT (*C macro*), 501
FS_EMPTY (*C macro*), 501
FS_EEXIST (*C macro*), 502
FS_EFULL (*C macro*), 501
FS_EHW (*C macro*), 501
FS EINVAL (*C macro*), 501
FS_ENOENT (*C macro*), 501
FS_ENOMEM (*C macro*), 501
FS_EOFFSET (*C macro*), 501
FS_EOK (*C macro*), 501
FS_EOS (*C macro*), 501
FS_EUNEXP (*C macro*), 501
FS_EUNINIT (*C macro*), 502
fs_filelen (*C function*), 502

fs_flush (*C function*), 502
fs_getpos (*C function*), 502
FS_MGMT_ID_FILE (*C macro*), 502
FS_MGMT_MAX_NAME (*C macro*), 502
fs_mkdir (*C function*), 502
fs_open (*C function*), 502
fs_opendir (*C function*), 502
fs_ops (*C struct*), 508
fs_ops.f_close (*C var*), 509
fs_ops.f_closedir (*C var*), 509
fs_ops.f dirent_is_dir (*C var*), 509
fs_ops.f dirent_name (*C var*), 509
fs_ops.f filelen (*C var*), 509
fs_ops.f flush (*C var*), 509
fs_ops.f getpos (*C var*), 509
fs_ops.f mkdir (*C var*), 509
fs_ops.f name (*C var*), 509
fs_ops.f open (*C var*), 509
fs_ops.f opendir (*C var*), 509
fs_ops.f read (*C var*), 509
fs_ops.f readdir (*C var*), 509
fs_ops.f rename (*C var*), 509
fs_ops.f seek (*C var*), 509
fs_ops.f unlink (*C var*), 509
fs_ops.f write (*C var*), 509
fs_ops_for (*C function*), 508
fs_ops_from_container (*C function*), 508
fs_ops_try_unique (*C function*), 508
fs_read (*C function*), 502
fs_readdir (*C function*), 502
fs_register (*C function*), 508
fs_rename (*C function*), 502
fs_seek (*C function*), 502
fs_unlink (*C function*), 502
fs_write (*C function*), 502
fsutil_read_file (*C function*), 503
fsutil_write_file (*C function*), 503

G

g_console_ignore_non_nlip (*C var*), 393
g_console_silence (*C var*), 393
g_console_silence_non_nlip (*C var*), 393
g_stats_registry (*C var*), 384
g_ts_suites (*C var*), 558

H

hal_bsp_core_dump (*C function*), 427
hal_bsp_deinit (*C function*), 427
hal_bsp_flash_dev (*C function*), 427
hal_bsp_get_nvic_priority (*C function*), 428
hal_bsp_hw_id (*C function*), 427
hal_bsp_hw_id_len (*C function*), 427
hal_bsp_init (*C function*), 427
HAL_BSP_MAX_ID_LEN (*C macro*), 428

hal_bsp_mem_dump (*C struct*), 428
HAL_BSP_POWER_DEEP_SLEEP (*C macro*), 428
HAL_BSP_POWER_OFF (*C macro*), 428
HAL_BSP_POWER_ON (*C macro*), 428
HAL_BSP_POWER_PERUSER (*C macro*), 428
HAL_BSP_POWER_SLEEP (*C macro*), 428
hal_bsp_power_state (*C function*), 427
HAL_BSP_POWER_WFI (*C macro*), 428
hal_debugger_connected (*C function*), 425
hal_flash_align (*C function*), 423
hal_flash_erase (*C function*), 423
hal_flash_erase_sector (*C function*), 422
hal_flash_erased_val (*C function*), 424
hal_flash_init (*C function*), 424
hal_flash_ioctl (*C function*), 422
hal_flash_isempty (*C function*), 423
hal_flash_isempty_no_buf (*C function*), 423
hal_flash_read (*C function*), 422
hal_flash_sector_info (*C function*), 422
hal_flash_write (*C function*), 422
hal_flash_write_protect (*C function*), 424
hal_gpio_deinit (*C function*), 407
hal_gpio_init_in (*C function*), 406
hal_gpio_init_out (*C function*), 407
hal_gpio_irq_disable (*C function*), 408
hal_gpio_irq_enable (*C function*), 408
hal_gpio_irq_handler_t (*C type*), 406
hal_gpio_irq_init (*C function*), 407
hal_gpio_irq_release (*C function*), 408
hal_gpio_irq_trig_t (*C type*), 406
hal_gpio_irq_trigger (*C enum*), 406
hal_gpio_irq_trigger.HAL_GPIO_TRIGGER_HAL_GPIO_TRIGGER_BOTH (*C enumerator*), 406
hal_gpio_irq_trigger.HAL_GPIO_TRIGGER_FALLING (*C enumerator*), 406
hal_gpio_irq_trigger.HAL_GPIO_TRIGGER_HIGH (*C enumerator*), 406
hal_gpio_irq_trigger.HAL_GPIO_TRIGGER_LOW (*C enumerator*), 406
hal_gpio_irq_trigger.HAL_GPIO_TRIGGER_NONE (*C enumerator*), 406
hal_gpio_irq_trigger.HAL_GPIO_TRIGGER_RISING (*C enumerator*), 406
hal_gpio_mode_e (*C enum*), 405
hal_gpio_mode_e.HAL_GPIO_MODE_IN (*C enumerator*), 405
hal_gpio_mode_e.HAL_GPIO_MODE_NC (*C enumerator*), 405
hal_gpio_mode_e.HAL_GPIO_MODE_OUT (*C enumerator*), 405
hal_gpio_mode_t (*C type*), 406
hal_gpio_pull (*C enum*), 406
hal_gpio_pull.HAL_GPIO_PULL_DOWN (*C enumerator*), 406

hal_gpio_pull.HAL_GPIO_PULL_NONE (*C enumerator*), 406
 hal_gpio_pull.HAL_GPIO_PULL_UP (*C enumerator*), 406
 hal_gpio_pull_t (*C type*), 406
 hal_gpio_read (*C function*), 407
 hal_gpio_toggle (*C function*), 407
 hal_gpio_write (*C function*), 407
 hal_i2c_config (*C function*), 418
 hal_i2c_disable (*C function*), 418
 hal_i2c_enable (*C function*), 418
 HAL_I2C_ERR_ADDR_NACK (*C macro*), 420
 HAL_I2C_ERR_DATA_NACK (*C macro*), 420
 HAL_I2C_ERR_INVAL (*C macro*), 420
 HAL_I2C_ERR_TIMEOUT (*C macro*), 420
 HAL_I2C_ERR_UNKNOWN (*C macro*), 419
 hal_i2c_hw_settings (*C struct*), 420
 hal_i2c_init (*C function*), 418
 hal_i2c_init_hw (*C function*), 418
 hal_i2c_master_data (*C struct*), 420
 hal_i2c_master_data.address (*C var*), 421
 hal_i2c_master_data.buffer (*C var*), 421
 hal_i2c_master_data.len (*C var*), 421
 hal_i2c_master_probe (*C function*), 419
 hal_i2c_master_read (*C function*), 419
 hal_i2c_master_write (*C function*), 419
 hal_i2c_settings (*C struct*), 420
 hal_i2c_settings.frequency (*C var*), 420
 hal_reset_cause (*C function*), 425
 hal_reset_cause_str (*C function*), 425
 hal_reset_reason (*C enum*), 424
 hal_reset_reason.HAL_RESET_BROWNOUT (*C enumerator*), 425
 hal_reset_reason.HAL_RESET_DFU (*C enumerator*), 425
 hal_reset_reason.HAL_RESET_OTHER (*C enumerator*), 425
 hal_reset_reason.HAL_RESET_PIN (*C enumerator*), 424
 hal_reset_reason.HAL_RESET_POR (*C enumerator*), 424
 hal_reset_reason.HAL_RESET_REQUESTED (*C enumerator*), 425
 hal_reset_reason.HAL_RESET_SOFT (*C enumerator*), 425
 hal_reset_reason.HAL_RESET_SYS_OFF_INT (*C enumerator*), 425
 hal_reset_reason.HAL_RESET_WATCHDOG (*C enumerator*), 425
 hal_spi_abort (*C function*), 415
 hal_spi_config (*C function*), 412
 hal_spi_data_mode_breakout (*C function*), 415
 hal_spi_disable (*C function*), 413
 hal_spi_enable (*C function*), 413
 hal_spi_hw_settings (*C struct*), 416
 hal_spi_init (*C function*), 412
 hal_spi_init_hw (*C function*), 412
 HAL_SPI_LSB_FIRST (*C macro*), 416
 HAL_SPI_MODE0 (*C macro*), 415
 HAL_SPI_MODE1 (*C macro*), 415
 HAL_SPI_MODE2 (*C macro*), 415
 HAL_SPI_MODE3 (*C macro*), 415
 HAL_SPI_MSB_FIRST (*C macro*), 416
 hal_spi_set_txrx_cb (*C function*), 412
 hal_spi_settings (*C struct*), 416
 hal_spi_settings.baudrate (*C var*), 416
 hal_spi_settings.data_mode (*C var*), 416
 hal_spi_settings.data_order (*C var*), 416
 hal_spi_settings.word_size (*C var*), 416
 hal_spi_slave_set_def_tx_val (*C function*), 415
 hal_spi_tx_val (*C function*), 413
 hal_spi_txrx (*C function*), 413
 hal_spi_txrx_cb (*C type*), 412
 hal_spi_txrx_noblock (*C function*), 414
 HAL_SPI_TYPE_MASTER (*C macro*), 415
 HAL_SPI_TYPE_SLAVE (*C macro*), 415
 HAL_SPI_WORD_SIZE_8BIT (*C macro*), 416
 HAL_SPI_WORD_SIZE_9BIT (*C macro*), 416
 hal_system_clock_start (*C function*), 425
 hal_system_reset_cb (*C function*), 425
 hal_timer (*C struct*), 404
 hal_timer.bsp_timer (*C var*), 404
 hal_timer.cb_arg (*C var*), 404
 hal_timer.cb_func (*C var*), 404
 hal_timer.expiry (*C var*), 404
 hal_timer_cb (*C type*), 402
 hal_timer_config (*C function*), 402
 hal_timer_deinit (*C function*), 402
 hal_timer_delay (*C function*), 403
 hal_timer_get_resolution (*C function*), 403
 hal_timer_init (*C function*), 402
 hal_timer_read (*C function*), 403
 hal_timer_set_cb (*C function*), 403
 hal_timer_start (*C function*), 403
 hal_timer_start_at (*C function*), 404
 hal_timer_stop (*C function*), 404
 hal_uart_blocking_tx (*C function*), 411
 hal_uart_close (*C function*), 410
 hal_uart_config (*C function*), 410
 hal_uart_flow_ctrl (*C enum*), 409
 hal_uart_flow_ctrl.HAL_UART_FLOW_CTL_NONE (*C enumerator*), 409
 hal_uart_flow_ctrl.HAL_UART_FLOW_CTL_RTS_CTS (*C enumerator*), 409
 hal_uart_init (*C function*), 410
 hal_uart_init_cbs (*C function*), 410
 hal_uart_parity (*C enum*), 409

hal_uart_parity.HAL_UART_PARITY_EVEN (*C enumerator*), 409
hal_uart_parity.HAL_UART_PARITY_NONE (*C enumerator*), 409
hal_uart_parity.HAL_UART_PARITY_ODD (*C enumerator*), 409
hal_uart_rx_char (*C type*), 409
hal_uart_start_rx (*C function*), 410
hal_uart_start_tx (*C function*), 410
hal_uart_tx_char (*C type*), 409
hal_uart_tx_done (*C type*), 409
hal_watchdog_enable (*C function*), 426
hal_watchdog_init (*C function*), 426
hal_watchdog_tickle (*C function*), 426

|

INT32_MAX (*C macro*), 346

J

json_array_t (*C struct*), 567
json_array_t.arr (*C union*), 568
json_array_t.arr (*C var*), 567
json_array_t.arr.booleans (*C struct*), 570
json_array_t.arr.booleans (*C var*), 569
json_array_t.arr.booleans.store (*C var*), 570
json_array_t.arr.integers (*C struct*), 569
json_array_t.arr.integers (*C var*), 569
json_array_t.arr.integers.store (*C var*), 570
json_array_t.arr.objects (*C struct*), 569
json_array_t.arr.objects (*C var*), 569
json_array_t.arr.objects.base (*C var*), 569
json_array_t.arr.objects.stride (*C var*), 569
json_array_t.arr.objects.subtype (*C var*), 569
json_array_t.arr.reals (*C struct*), 570
json_array_t.arr.reals (*C var*), 569
json_array_t.arr.reals.store (*C var*), 570
json_array_t.arr.strings (*C struct*), 569
json_array_t.arr.strings (*C var*), 569
json_array_t.arr.strings.ptrs (*C var*), 569
json_array_t.arr.strings.store (*C var*), 569
json_array_t.arr.strings.storelen (*C var*), 569
json_array_t.arr.uintegers (*C struct*), 570
json_array_t.arr.uintegers (*C var*), 569
json_array_t.arr.uintegers.store (*C var*), 570
json_array_t.count (*C var*), 567
json_array_t.element_type (*C var*), 567
json_array_t maxlen (*C var*), 567
JSON_ATTR_MAX (*C macro*), 563
json_attr_t (*C struct*), 567
json_attr_t.addr (*C union*), 570
json_attr_t.addr (*C var*), 567
json_attr_t.addr.array (*C var*), 570
json_attr_t.addr.boolean (*C var*), 570
json_attr_t.addr.character (*C var*), 570
json_attr_t.addr.integer (*C var*), 570
json_attr_t.addr.offset (*C var*), 570
json_attr_t.addr.real (*C var*), 570
json_attr_t.addr.string (*C var*), 570
json_attr_t.addr.uinteger (*C var*), 570
json_attr_t.attribute (*C var*), 567
json_attr_t.dflt (*C union*), 571
json_attr_t.dflt (*C var*), 567
json_attr_t.dflt.boolean (*C var*), 571
json_attr_t.dflt.character (*C var*), 571
json_attr_t.dflt.check (*C var*), 571
json_attr_t.dflt.integer (*C var*), 571
json_attr_t.dflt.real (*C var*), 571
json_attr_t.dflt.uinteger (*C var*), 571
json_attr_t.len (*C var*), 567
json_attr_t.map (*C var*), 567
json_attr_t.nodefault (*C var*), 567
json_attr_t.type (*C var*), 567
json_buffer (*C struct*), 568
json_buffer.jb_read_next (*C var*), 568
json_buffer.jb_read_prev (*C var*), 568
json_buffer.jb_readn (*C var*), 568
json_buffer.read_next_byte_t (*C type*), 565
json_buffer.read_prev_byte_t (*C type*), 565
json_buffer.readn_t (*C type*), 565
json_encode_array_finish (*C function*), 566
json_encode_array_name (*C function*), 566
json_encode_array_start (*C function*), 566
json_encode_array_value (*C function*), 566
json_encode_object_entry (*C function*), 566
json_encode_object_finish (*C function*), 566
json_encode_object_key (*C function*), 566
json_encode_object_start (*C function*), 566
json_encoder (*C struct*), 566
json_encoder.je_arg (*C var*), 566
json_encoder.je_encode_buf (*C var*), 566
json_encoder.je_wr_commas (*C var*), 566
json_encoder.je_write (*C var*), 566
json_enum_t (*C struct*), 567
json_enum_t.name (*C var*), 567
json_enum_t.value (*C var*), 567
JSON_ERR_ARRAYSTART (*C macro*), 564
JSON_ERR_ATTRLEN (*C macro*), 564
JSON_ERR_ATTRSTART (*C macro*), 564
JSON_ERR_BADATTR (*C macro*), 564
JSON_ERR_BADENUM (*C macro*), 564
JSON_ERR_BADNUM (*C macro*), 564
JSON_ERR_BADSTRING (*C macro*), 564
JSON_ERR_BADSUBTRAIL (*C macro*), 564
JSON_ERR_BADTRAIL (*C macro*), 564
JSON_ERR_CHECKFAIL (*C macro*), 564
JSON_ERR_MISC (*C macro*), 564
JSON_ERR_NOARRAY (*C macro*), 564
JSON_ERR_NOBRAK (*C macro*), 564

JSON_ERR_NONQSTRING (*C macro*), 564
 JSON_ERR_NOPARSTR (*C macro*), 564
 JSON_ERR_NULLPTR (*C macro*), 565
 JSON_ERR_OBJARR (*C macro*), 564
 JSON_ERR_OBSTART (*C macro*), 563
 JSON_ERR_QNONSTRING (*C macro*), 564
 JSON_ERR_STRLONG (*C macro*), 564
 JSON_ERR_SUBTOOLONG (*C macro*), 564
 JSON_ERR_SUBTYPE (*C macro*), 564
 JSON_ERR_TOKLONG (*C macro*), 564
 JSON_NITEMS (*C macro*), 563
 json_read_array (*C function*), 566
 json_read_object (*C function*), 566
 JSON_STRUCT_ARRAY (*C macro*), 565
 JSON_STRUCT_OBJECT (*C macro*), 565
 json_type (*C enum*), 565
 json_type.t_array (*C enumerator*), 565
 json_type.t_boolean (*C enumerator*), 565
 json_type.t_character (*C enumerator*), 565
 json_type.t_check (*C enumerator*), 565
 json_type.t_ignore (*C enumerator*), 565
 json_type.t_integer (*C enumerator*), 565
 json_type.t_object (*C enumerator*), 565
 json_type.t_real (*C enumerator*), 565
 json_type.t_string (*C enumerator*), 565
 json_type.t_structobject (*C enumerator*), 565
 json_type.t_uinteger (*C enumerator*), 565
 JSON_VAL_MAX (*C macro*), 563
 json_value (*C struct*), 566
 json_value.jv_len (*C var*), 566
 json_value.jv_pad1 (*C var*), 566
 json_value.jv_type (*C var*), 566
 json_value.jv_val (*C union*), 568
 json_value.jv_val (*C var*), 566
 json_value.jv_val.composite (*C struct*), 568
 json_value.jv_val.composite (*C var*), 568
 json_value.jv_val.composite.keys (*C var*), 568
 json_value.jv_val.composite.values (*C var*), 568
 json_value.jv_val.fl (*C var*), 568
 json_value.jv_val.str (*C var*), 568
 json_value.jv_val.u (*C var*), 568
 JSON_VALUE_BOOL (*C macro*), 563
 JSON_VALUE_INT (*C macro*), 563
 JSON_VALUE_STRING (*C macro*), 563
 JSON_VALUE_STRINGN (*C macro*), 563
 JSON_VALUE_TYPE_ARRAY (*C macro*), 563
 JSON_VALUE_TYPE_BOOL (*C macro*), 563
 JSON_VALUE_TYPE_INT64 (*C macro*), 563
 JSON_VALUE_TYPE_OBJECT (*C macro*), 563
 JSON_VALUE_TYPE_STRING (*C macro*), 563
 JSON_VALUE_TYPE_UINT64 (*C macro*), 563
 JSON_VALUE_UINT (*C macro*), 563
 json_write_func_t (*C type*), 565

L

lh_append_body_func_t (*C type*), 366
 lh_append_func_t (*C type*), 366
 lh_append_mbuf_body_func_t (*C type*), 366
 lh_append_mbuf_func_t (*C type*), 366
 lh_flush_func_t (*C type*), 366
 lh_read_func_t (*C type*), 366
 lh_read_mbuf_func_t (*C type*), 366
 lh_registered_func_t (*C type*), 366
 lh_set_watermark_func_t (*C type*), 366
 lh_storage_info_func_t (*C type*), 366
 lh_walk_func_t (*C type*), 366
 log (*C struct*), 375
 log.l_append_cb (*C var*), 376
 log.l_arg (*C var*), 376
 log.l_level (*C var*), 376
 log.l_log (*C var*), 376
 log.l_max_entry_len (*C var*), 376
 log.l_name (*C var*), 376
 log.l_rotate_notify_cb (*C var*), 376
 log_append (*C function*), 369
 log_append_body (*C function*), 368
 log_append_mbuf (*C function*), 370
 log_append_mbuf_body (*C function*), 369
 log_append_mbuf_body_no_free (*C function*), 368
 log_append_mbuf_no_free (*C function*), 370
 log_append_mbuf_typed (*C function*), 368
 log_append_mbuf_typed_no_free (*C function*), 367
 log_append_typed (*C function*), 367
 LOG_BASE_ENTRY_HDR_SIZE (*C macro*), 365
 log_cbmem_handler (*C var*), 374
 log_console_get (*C function*), 369
 log_console_handler (*C var*), 374
 log_console_init (*C function*), 369
 LOG_CRITICAL (*C macro*), 365
 LOG_DEBUG (*C macro*), 365
 LOG_ERROR (*C macro*), 365
 log_fcb_handler (*C var*), 374
 log_fill_current_img_hash (*C function*), 374
 log_find (*C function*), 367
 LOG_FLAGS_IMG_HASH (*C macro*), 365
 log_flush (*C function*), 372
 log_get_last_index (*C function*), 373
 log_get_level (*C function*), 373
 log_handler (*C struct*), 375
 log_handler.log_append (*C var*), 375
 log_handler.log_append_body (*C var*), 375
 log_handler.log_append_mbuf (*C var*), 375
 log_handler.log_append_mbuf_body (*C var*), 375
 log_handler.log_flush (*C var*), 375
 log_handler.log_read (*C var*), 375
 log_handler.log_read_mbuf (*C var*), 375
 log_handler.log_registered (*C var*), 375
 log_handler.log_set_watermark (*C var*), 375

log_handler.log_storage_info (*C var*), 375
log_handler.log_type (*C var*), 375
log_handler.log_walk (*C var*), 375
log_handler.log_walk_sector (*C var*), 375
log_hdr_len (*C function*), 371
LOG_HDR_SIZE (*C macro*), 365
LOG_IMG_HASHLEN (*C macro*), 365
LOG_INFO (*C macro*), 365
log_init (*C function*), 366
log_level_get (*C function*), 372
log_level_set (*C function*), 372
log_list_get_next (*C function*), 366
log_module_get_name (*C function*), 366
log_module_register (*C function*), 366
LOG_MODULE_STR (*C macro*), 365
log_offset (*C struct*), 374
log_offset.lo_arg (*C var*), 374
log_offset.lo_data_len (*C var*), 374
log_offset.lo_index (*C var*), 374
log_offset.lo_ts (*C var*), 374
log_offset.lo_walk_backward (*C var*), 374
log_printf (*C function*), 370
log_read (*C function*), 370
log_read_body (*C function*), 371
log_read_hdr (*C function*), 371
log_read_mbuf (*C function*), 371
log_read_mbuf_body (*C function*), 371
log_register (*C function*), 366
log_set_append_cb (*C function*), 366
log_set_level (*C function*), 372
log_set_max_entry_len (*C function*), 373
log_set_rotate_notify_cb (*C function*), 373
log_set_watermark (*C function*), 373
LOG_STATS_INC (*C macro*), 365
LOG_STATS_INCN (*C macro*), 365
log_storage_info (*C function*), 373
log_storage_info (*C struct*), 374
log_storage_info.size (*C var*), 375
log_storage_info.used (*C var*), 375
log_storage_info.used_unread (*C var*), 375
log_walk (*C function*), 371
log_walk_body (*C function*), 372
log_walk_body_func_t (*C type*), 366
log_walk_body_section (*C function*), 372
log_walk_func_t (*C type*), 366
LOG_WARN (*C macro*), 365

M

MFG_HASH_SZ (*C macro*), 573
mfg_init (*C function*), 574
mfg_meta_flash_area (*C struct*), 575
mfg_meta_flash_area.area_id (*C var*), 575
mfg_meta_flash_area.device_id (*C var*), 575
mfg_meta_flash_area.offset (*C var*), 575

mfg_meta_flash_area.size (*C var*), 575
mfg_meta_flash_traits (*C struct*), 575
mfg_meta_flash_traits.device_id (*C var*), 576
mfg_meta_flash_traits.min_write_sz (*C var*), 576
mfg_meta_mmr_ref (*C struct*), 576
mfg_meta_mmr_ref.area_id (*C var*), 576
mfg_meta_tlv (*C struct*), 575
mfg_meta_tlv.size (*C var*), 575
mfg_meta_tlv.type (*C var*), 575
MFG_META_TLV_TYPE_FLASH_AREA (*C macro*), 573
MFG_META_TLV_TYPE_FLASH_TRAITS (*C macro*), 573
MFG_META_TLV_TYPE_HASH (*C macro*), 573
MFG_META_TLV_TYPE_MMR_REF (*C macro*), 573
mfg_open (*C function*), 573
mfg_read_tlv_flash_area (*C function*), 574
mfg_read_tlv_hash (*C function*), 574
mfg_read_tlv_mmr_ref (*C function*), 574
mfg_reader (*C struct*), 576
mfg_reader.cur_tlv (*C var*), 576
mfg_reader.mmr_idx (*C var*), 576
mfg_reader.offset (*C var*), 576
mfg_seek_next (*C function*), 573
mfg_seek_next_with_type (*C function*), 574
min_write_sz (*C var*), 575

N

nffs_area_desc (*C struct*), 505
nffs_area_desc.nad_flash_id (*C var*), 506
nffs_area_desc.nad_length (*C var*), 506
nffs_area_desc.nad_offset (*C var*), 506
nffs_config (*C struct*), 505
nffs_config (*C var*), 505
nffs_config.nc_num_blocks (*C var*), 505
nffs_config.nc_num_cache_blocks (*C var*), 505
nffs_config.nc_num_cache_inodes (*C var*), 505
nffs_config.nc_num_dirs (*C var*), 505
nffs_config.nc_num_files (*C var*), 505
nffs_config.nc_num_inodes (*C var*), 505
nffs_detect (*C function*), 505
NFFS_FILENAME_MAX_LEN (*C macro*), 504
nffs_format (*C function*), 505
nffs_init (*C function*), 505
NFFS_MAX AREAS (*C macro*), 504
nffs_misc_desc_from_flash_area (*C function*), 505
NUM (*C macro*), 560

O

offset (*C var*), 575
os_callout (*C struct*), 313
os_callout.c_ev (*C var*), 314
os_callout.c_evq (*C var*), 314
os_callout.c_ticks (*C var*), 314
os_callout_init (*C function*), 313
os_callout_queued (*C function*), 313

os_callout_remaining_ticks (*C function*), 313
 os_callout_reset (*C function*), 313
 os_callout_stop (*C function*), 313
 os_cputime_delay_nsecs (*C function*), 340
 os_cputime_delay_ticks (*C function*), 340
 os_cputime_delay_usecs (*C function*), 340
 os_cputime_get32 (*C function*), 339
 os_cputime_init (*C function*), 339
 os_cputime_nsecs_to_ticks (*C function*), 339
 os_cputime_ticks_to_nsecs (*C function*), 339
 os_cputime_ticks_to_usecs (*C function*), 340
 os_cputime_timer_init (*C function*), 340
 os_cputime_timer_relative (*C function*), 341
 os_cputime_timer_start (*C function*), 341
 os_cputime_timer_stop (*C function*), 341
 os_cputime_usecs_to_ticks (*C function*), 340
 os_event (*C struct*), 311
 os_event.ev_arg (*C var*), 311
 os_event.ev_cb (*C var*), 311
 os_event.ev_queued (*C var*), 311
 os_event_fn (*C type*), 309
 OS_EVENT_QUEUED (*C macro*), 311
 os_eventq (*C struct*), 312
 os_eventq.evq_owner (*C var*), 312
 os_eventq.evq_prev (*C var*), 312
 os_eventq.evq_task (*C var*), 312
 os_eventq_dflt_get (*C function*), 311
 os_eventq_get (*C function*), 310
 os_eventq_get_no_wait (*C function*), 310
 os_eventq_init (*C function*), 309
 os_eventq_initiated (*C function*), 309
 os_eventq_mon (*C struct*), 312
 os_eventq_mon_start (*C function*), 311
 os_eventq_mon_stop (*C function*), 311
 os_eventq_poll (*C function*), 310
 os_eventq_put (*C function*), 310
 os_eventq_remove (*C function*), 310
 os_eventq_run (*C function*), 310
 os_free (*C function*), 314
 os_get_uptime (*C function*), 345
 os_get_uptime_usec (*C function*), 345
 os_gettimeofday (*C function*), 344
 os_malloc (*C function*), 314
 os_mbuf (*C struct*), 338
 os_mbuf.om_data (*C var*), 338
 os_mbuf.om_databuf (*C var*), 338
 os_mbuf.om_flags (*C var*), 338
 os_mbuf.om_len (*C var*), 338
 os_mbuf.om_omp (*C var*), 338
 os_mbuf.om_pkthdr_len (*C var*), 338
 os_mbuf_adj (*C function*), 333
 os_mbuf_append (*C function*), 332
 os_mbuf_appendfrom (*C function*), 332
 os_mbuf_cmpf (*C function*), 333
 os_mbuf_cmpm (*C function*), 333
 os_mbuf_concat (*C function*), 334
 os_mbuf_copydata (*C function*), 332
 os_mbuf_copyinto (*C function*), 334
 OS_MBUF_DATA (*C macro*), 336
 os_mbuf_dup (*C function*), 331
 os_mbuf_extend (*C function*), 335
 OS_MBUF_F_MASK (*C macro*), 336
 os_mbuf_free (*C function*), 333
 os_mbuf_free_chain (*C function*), 333
 os_mbuf_get (*C function*), 331
 os_mbuf_get_pkthdr (*C function*), 331
 OS_MBUF_IS_PKTHDR (*C macro*), 336
 OS_MBUF.LEADINGSPACE (*C macro*), 337
 os_mbuf_len (*C function*), 332
 os_mbuf_off (*C function*), 331
 os_mbuf_pack_chains (*C function*), 336
 OS_MBUF_PKTHDR (*C macro*), 336
 os_mbuf_pkthdr (*C struct*), 337
 os_mbuf_pkthdr.omp_flags (*C var*), 338
 os_mbuf_pkthdr.omp_len (*C var*), 338
 OS_MBUF_PKTHDR_TO_MBUF (*C macro*), 336
 OS_MBUF_PKTLEN (*C macro*), 336
 os_mbuf_pool (*C struct*), 337
 os_mbuf_pool.omp_databuf_len (*C var*), 337
 os_mbuf_pool.omp_pool (*C var*), 337
 os_mbuf_pool_init (*C function*), 331
 os_mbuf_prepend (*C function*), 334
 os_mbuf_prepend_pullup (*C function*), 334
 os_mbuf_pullup (*C function*), 335
 OS_MBUF.TRAILINGSPACE (*C macro*), 337
 os_mbuf_trim_front (*C function*), 335
 OS_MBUF_USRHDR (*C macro*), 336
 OS_MBUF_USRHDR_LEN (*C macro*), 336
 os_mbuf_widen (*C function*), 335
 os_memblock (*C struct*), 319
 os_memblock_from (*C function*), 317
 os_memblock_get (*C function*), 318
 os_memblock_put (*C function*), 318
 os_memblock_put_from_cb (*C function*), 318
 os_mempool (*C struct*), 319
 os_mempool.mp_block_size (*C var*), 319
 os_mempool.mp_flags (*C var*), 319
 os_mempool.mp_mbuf_addr (*C var*), 319
 os_mempool.mp_min_free (*C var*), 319
 os_mempool.mp_num_blocks (*C var*), 319
 os_mempool.mp_num_free (*C var*), 319
 os_mempool.name (*C var*), 319
 OS_MEMPOOL_BLOCK_SZ (*C macro*), 318
 OS_MEMPOOL_BYTES (*C macro*), 318
 os_mempool_clear (*C function*), 317
 os_mempool_ext (*C struct*), 320
 os_mempool_ext.mpe_mp (*C var*), 320
 os_mempool_ext.mpe_put_arg (*C var*), 320

os_mempool_ext.mpe_put_cb (*C var*), 320
os_mempool_ext_init (*C function*), 317
OS_MEMPOOL_F_EXT (*C macro*), 318
os_mempool_get (*C function*), 316
os_mempool_info (*C struct*), 320
os_mempool_info.omi_block_size (*C var*), 320
os_mempool_info.omi_min_free (*C var*), 320
os_mempool_info.omi_name (*C var*), 320
os_mempool_info.omi_num_blocks (*C var*), 320
os_mempool_info.omi_num_free (*C var*), 320
os_mempool_info_get_next (*C function*), 316
OS_MEMPOOL_INFO_NAME_LEN (*C macro*), 318
os_mempool_init (*C function*), 316
os_mempool_is_sane (*C function*), 317
os_mempool_put_fn (*C type*), 316
OS_MEMPOOL_SIZE (*C macro*), 318
os_mempool_unregister (*C function*), 317
os_mqueue (*C struct*), 338
os_mqueue.mq_ev (*C var*), 339
os_mqueue_get (*C function*), 329
os_mqueue_init (*C function*), 329
os_mqueue_put (*C function*), 329
os_msyst_count (*C function*), 330
os_msyst_get (*C function*), 330
os_msyst_get_pkthdr (*C function*), 330
os_msyst_num_free (*C function*), 330
os_msyst_register (*C function*), 330
os_msyst_reset (*C function*), 330
os_mutex (*C struct*), 306
os_mutex._pad (*C var*), 306
os_mutex.mu_level (*C var*), 306
os_mutex.mu_owner (*C var*), 306
os_mutex.mu_prio (*C var*), 306
os_mutex_get_level (*C function*), 305
os_mutex_init (*C function*), 305
os_mutex_pend (*C function*), 305
os_mutex_release (*C function*), 305
os_realloc (*C function*), 314
os_sanity_check (*C struct*), 352
os_sanity_check.sc_arg (*C var*), 352
os_sanity_check.sc_checkin_itvl (*C var*), 352
os_sanity_check.sc_checkin_last (*C var*), 352
os_sanity_check.sc_func (*C var*), 352
os_sanity_check_func_t (*C type*), 351
os_sanity_check_init (*C function*), 351
os_sanity_check_register (*C function*), 351
os_sanity_check_reset (*C function*), 351
OS_SANITY_CHECK_SETFUNC (*C macro*), 352
os_sanity_task_checkin (*C function*), 351
os_sched (*C function*), 296
os_sched_get_current_task (*C function*), 295
os_sched_next_task (*C function*), 296
os_sched_resume (*C function*), 296
os_sched_set_current_task (*C function*), 295
os_sched_suspend (*C function*), 296
os_sem (*C struct*), 308
os_sem._pad (*C var*), 308
os_sem.sem_tokens (*C var*), 308
os_sem_get_count (*C function*), 308
os_sem_init (*C function*), 307
os_sem_pend (*C function*), 308
os_sem_release (*C function*), 307
os_settimeofday (*C function*), 344
OS_STIME_MAX (*C macro*), 346
os_stime_t (*C type*), 344
os_task (*C struct*), 302
os_task.t_arg (*C var*), 303
os_task.t_ctx_sw_cnt (*C var*), 303
os_task.t_flags (*C var*), 303
os_task.t_func (*C var*), 303
os_task.t_lockcnt (*C var*), 303
os_task.t_name (*C var*), 303
os_task.t_next_wakeup (*C var*), 303
os_task.t_pad (*C var*), 303
os_task.t_prio (*C var*), 303
os_task.t_run_time (*C var*), 303
os_task.t_sanity_check (*C var*), 303
os_task.t_stackbottom (*C var*), 302
os_task.t_stackptr (*C var*), 302
os_task.t_stacksize (*C var*), 302
os_task.t_state (*C var*), 303
os_task.t_taskid (*C var*), 302
os_task_count (*C function*), 300
OS_TASK_FLAG_EVQ_WAIT (*C macro*), 302
OS_TASK_FLAG_MUTEX_WAIT (*C macro*), 301
OS_TASK_FLAG_NO_TIMEOUT (*C macro*), 301
OS_TASK_FLAG_SEM_WAIT (*C macro*), 301
os_task_func_t (*C type*), 299
os_task_info (*C struct*), 303
os_task_info.oti_cswcnt (*C var*), 304
os_task_info.oti_last_checkin (*C var*), 304
os_task_info.oti_name (*C var*), 304
os_task_info.oti_next_checkin (*C var*), 304
os_task_info.oti_prio (*C var*), 304
os_task_info.oti_runtime (*C var*), 304
os_task_info.oti_state (*C var*), 304
os_task_info.oti_stksize (*C var*), 304
os_task_info.oti_stkusage (*C var*), 304
os_task_info.oti_taskid (*C var*), 304
os_task_info_get (*C function*), 300
os_task_info_get_next (*C function*), 300
os_task_init (*C function*), 299
OS_TASK_MAX_NAME_LEN (*C macro*), 302
os_task_obj (*C struct*), 302
OS_TASK_PRI_HIGHEST (*C macro*), 301
OS_TASK_PRI_LOWEST (*C macro*), 301
os_task_remove (*C function*), 299
OS_TASK_STACK_DEFINE (*C macro*), 301

OS_TASK_STACK_DEFINE_NOSTATIC (*C macro*), 301
 os_task_stacktop_get (*C function*), 300
 os_task_state (*C enum*), 299
 os_task_state.OS_TASK_READY (*C enumerator*), 299
 os_task_state.OS_TASK_SLEEP (*C enumerator*), 299
 os_task_state_t (*C type*), 299
 os_tick_idle (*C function*), 429
 os_tick_init (*C function*), 429
 OS_TICKS_PER_SEC (*C macro*), 346
 os_time_advance (*C function*), 344
 os_time_change_fn (*C type*), 344
 os_time_change_info (*C struct*), 347
 os_time_change_info.tci_cur_tv (*C var*), 347
 os_time_change_info.tci_cur_tz (*C var*), 348
 os_time_change_info.tci_newly_synced (*C var*), 348
 os_time_change_info.tci_prev_tv (*C var*), 347
 os_time_change_info.tci_prev_tz (*C var*), 347
 os_time_change_listen (*C function*), 346
 os_time_change_listener (*C struct*), 348
 os_time_change_listener.tcl_arg (*C var*), 348
 os_time_change_listener.tcl_fn (*C var*), 348
 os_time_change_remove (*C function*), 346
 os_time_delay (*C function*), 344
 os_time_get (*C function*), 344
 os_time_is_set (*C function*), 345
 OS_TIMEOUT_MAX (*C macro*), 346
 os_time_ms_to_ticks (*C function*), 345
 os_time_ms_to_ticks32 (*C function*), 345
 os_time_t (*C type*), 344
 os_time_ticks_to_ms (*C function*), 345
 os_time_ticks_to_ms32 (*C function*), 345
 OS_TIMEOUT_NEVER (*C macro*), 346
 os_timeradd (*C macro*), 347
 os_timersub (*C macro*), 347
 os_timeval (*C struct*), 347
 os_timeval.tv_sec (*C var*), 347
 os_timeval.tv_usec (*C var*), 347
 os_timezone (*C struct*), 347
 os_timezone.tz_dsttime (*C var*), 347
 os_timezone.tz_minuteswest (*C var*), 347

P

periodic_adv_set_default_sync_params (*C function*), 601

R

REST_OR_0 (*C macro*), 560
 REST_OR_0_AUX (*C macro*), 560
 REST_OR_0_AUX_1 (*C macro*), 560
 REST_OR_0_AUX_INNER (*C macro*), 560
 REST_OR_0_AUX_N (*C macro*), 560

S

sensor (*C struct*), 537
 sensor.s_poll_rate (*C var*), 537
 sensor_cfg (*C struct*), 535
 sensor_check_type (*C function*), 534
 sensor_clear_high_thresh (*C function*), 541
 sensor_clear_low_thresh (*C function*), 541
 sensor_clear_trigger_thresh_t (*C type*), 531
 SENSOR_DATA_CMP_GT (*C macro*), 535
 SENSOR_DATA_CMP_LT (*C macro*), 535
 sensor_data_func_t (*C type*), 529
 sensor_data_t (*C union*), 535
 sensor_data_t.sad (*C var*), 535
 sensor_data_t.satd (*C var*), 536
 sensor_data_t.scd (*C var*), 536
 sensor_data_t.sed (*C var*), 535
 sensor_data_t.sgd (*C var*), 536
 sensor_data_t.sgrd (*C var*), 535
 sensor_data_t.slad (*C var*), 535
 sensor_data_t.sld (*C var*), 536
 sensor_data_t.smd (*C var*), 535
 sensor_data_t.spd (*C var*), 536
 sensor_data_t.sqd (*C var*), 535
 sensor_data_t.srhd (*C var*), 536
 sensor_data_t.std (*C var*), 536
 sensor_driver (*C struct*), 536
 sensor_error_func_t (*C type*), 530
 sensor_event_type_t (*C enum*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_DOUBLE_TAP
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_FREE_FALL
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_CHANGE
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_X_CHANGE
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_X_H_CHANGE
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_X_L_CHANGE
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_Y_CHANGE
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_Y_H_CHANGE
 (*C enumerator*), 529
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_Y_L_CHANGE
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_Z_CHANGE
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_Z_H_CHANGE
 (*C enumerator*), 529
 sensor_event_type_t.SENSOR_EVENT_TYPE_ORIENT_Z_L_CHANGE
 (*C enumerator*), 528
 sensor_event_type_t.SENSOR_EVENT_TYPE_SINGLE_TAP
 (*C enumerator*), 528

sensor_event_type_t.SENSOR_EVENT_TYPE_SLEEP
 (*C enumerator*), 528
sensor_event_type_t.SENSOR_EVENT_TYPE_SLEEP_CHANGE
 (*C enumerator*), 528
sensor_event_type_t.SENSOR_EVENT_TYPE_TILT_CHANGE
 (*C enumerator*), 529
sensor_event_type_t.SENSOR_EVENT_TYPE_TILT_NEG
 (*C enumerator*), 529
sensor_event_type_t.SENSOR_EVENT_TYPE_TILT_POSS
 (*C enumerator*), 529
sensor_event_type_t.SENSOR_EVENT_TYPE_WAKEUP
 (*C enumerator*), 528
sensor_ftostr (*C function*), 541
sensor_get_config (*C function*), 534
sensor_get_config_func_t (*C type*), 530
SENSOR_GET_DEVICE (*C macro*), 535
SENSOR_GET_ITF (*C macro*), 535
sensor_get_type_traits_byname (*C function*), 540
sensor_get_type_traits_bytype (*C function*), 540
sensor_handle_interrupt_t (*C type*), 532
SENSOR_IGN_LISTENER (*C macro*), 535
sensor_init (*C function*), 532
sensor_int (*C struct*), 536
sensor_itf (*C struct*), 536
SENSOR_ITF_I2C (*C macro*), 535
sensor_itf_lock (*C function*), 532
SENSOR_ITF_SPI (*C macro*), 534
SENSOR_ITF_UART (*C macro*), 535
sensor_itf_unlock (*C function*), 532
sensor_listener (*C struct*), 536
sensor_lock (*C function*), 533
sensor_mgr_compare_func_t (*C type*), 538
sensor_mgr_evq_get (*C function*), 538
sensor_mgr_find_next (*C function*), 538
sensor_mgr_find_next_bydevname (*C function*), 539
sensor_mgr_find_next_bytype (*C function*), 539
sensor_mgr_lock (*C function*), 538
sensor_mgr_match_bytype (*C function*), 539
sensor_mgr_put_interrupt_evt (*C function*), 541
sensor_mgr_put_notify_evt (*C function*), 541
sensor_mgr_put_read_evt (*C function*), 541
sensor_mgr_register (*C function*), 538
sensor_mgr_unlock (*C function*), 538
sensor_notifier (*C struct*), 536
sensor_notifier_func_t (*C type*), 530
sensor_notify_ev_ctx (*C struct*), 536
sensor_notify_os_ev (*C struct*), 536
sensor_oic_init (*C function*), 542
sensor_oic_tx_trigger (*C function*), 540
sensor_pkg_init (*C function*), 532
sensor_read (*C function*), 533
sensor_read_ctx (*C struct*), 537
sensor_read_ev_ctx (*C struct*), 536
sensor_read_func_t (*C type*), 530
sensor_register_err_func (*C function*), 543
sensor_register_listener (*C function*), 543
sensor_register_notifier (*C function*), 544
sensor_reset (*C function*), 541
sensor_reset_t (*C type*), 532
sensor_set_config_func_t (*C type*), 531
sensor_set_driver (*C function*), 533
sensor_set_interface (*C function*), 534
sensor_set_n_poll_rate (*C function*), 540
sensor_set_notification_t (*C type*), 531
sensor_set_poll_rate_ms (*C function*), 539
sensor_set_thresh (*C function*), 540
sensor_set_trigger_thresh_t (*C type*), 531
sensor_set_type_mask (*C function*), 533
sensor_shell_register (*C function*), 542
SENSOR_THRESH_ALGO_USERDEF (*C macro*), 535
SENSOR_THRESH_ALGO_WATERMARK (*C macro*), 535
SENSOR_THRESH_ALGO_WINDOW (*C macro*), 535
sensor_timestamp (*C struct*), 536
sensor_trigger_cmp_func_t (*C type*), 529
sensor_trigger_init (*C function*), 540
sensor_trigger_notify_func_t (*C type*), 529
sensor_type_t (*C enum*), 527
sensor_type_t.SENSOR_TYPE_ACCELEROMETER
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_ALL
 (*C enumerator*), 528
sensor_type_t.SENSOR_TYPE_ALTITUDE
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_AMBIENT_TEMPERATURE
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_COLOR
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_CURRENT
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_EULER
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_GRAVITY
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_GYROSCOPE
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_LIGHT
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_LINEAR_ACCEL
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_MAGNETIC_FIELD
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_NONE
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_PRESSURE
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_PROXIMITY
 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_RELATIVE_HUMIDITY

(*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_ROTATION_VECTOR (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_TEMPERATURE (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_USER_DEFINED_1 (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_USER_DEFINED_2 (*C enumerator*), 528
sensor_type_t.SENSOR_TYPE_USER_DEFINED_3 (*C enumerator*), 528
sensor_type_t.SENSOR_TYPE_USER_DEFINED_4 (*C enumerator*), 528
sensor_type_t.SENSOR_TYPE_USER_DEFINED_5 (*C enumerator*), 528
sensor_type_t.SENSOR_TYPE_USER_DEFINED_6 (*C enumerator*), 528
sensor_type_t.SENSOR_TYPE_VOLTAGE (*C enumerator*), 527
sensor_type_t.SENSOR_TYPE_WEIGHT (*C enumerator*), 527
sensor_type_traits (*C struct*), 536
sensor_unlock (*C function*), 533
sensor_unregister_listener (*C function*), 543
sensor_unregister_notifier (*C function*), 544
sensor_unset_notification_t (*C type*), 532
SENSOR_VALUE_TYPE_FLOAT (*C macro*), 534
SENSOR_VALUE_TYPE_FLOAT_TRIPLET (*C macro*), 534
SENSOR_VALUE_TYPE_INT32 (*C macro*), 534
SENSOR_VALUE_TYPE_INT32_TRIPLET (*C macro*), 534
SENSOR_VALUE_TYPE_OPAQUE (*C macro*), 534
SHELL_CMD (*C macro*), 397
shell_cmd (*C struct*), 400
shell_cmd.help (*C var*), 400
shell_cmd.sc_cmd (*C var*), 400
shell_cmd.sc_ext (*C var*), 400
SHELL_CMD_EXT (*C macro*), 397
shell_cmd_ext_func_t (*C type*), 398
shell_cmd_func_t (*C type*), 398
shell_cmd_help (*C struct*), 399
shell_cmd_help.params (*C var*), 400
shell_cmd_help.summary (*C var*), 400
shell_cmd_help.usage (*C var*), 400
shell_cmd_register (*C function*), 399
shell_eqv_set (*C function*), 399
shell_exec (*C function*), 399
SHELL_HELP_ (*C macro*), 397
shell_module (*C struct*), 400
shell_module.commands (*C var*), 400
shell_module.name (*C var*), 400
shell_nlip_input_func_t (*C type*), 398
shell_nlip_input_register (*C function*), 399
shell_nlip_output (*C function*), 399
SHELL_NMGR_OP_EXEC (*C macro*), 397
shell_param (*C struct*), 399
shell_param.help (*C var*), 399
shell_param.param_name (*C var*), 399
shell_prompt_function_t (*C type*), 398
shell_register (*C function*), 398
shell_register_app_cmd_handler (*C function*), 398
shell_register_default_module (*C function*), 399
shell_register_prompt_handler (*C function*), 399
size (*C var*), 575
snm_name (*C var*), 384
snm_off (*C var*), 384
STANDARD_ACCEL_GRAVITY (*C macro*), 535
STATS_CLEAR (*C macro*), 382
STATS_GET (*C macro*), 381
stats_group_find (*C function*), 383
stats_group_walk (*C function*), 383
stats_group_walk_func_t (*C type*), 383
STATS_HDR (*C macro*), 381
stats_hdr (*C struct*), 385
stats_hdr.s_cnt (*C var*), 385
stats_hdr.s_flags (*C var*), 385
stats_hdr.s_map (*C var*), 385
stats_hdr.s_map_cnt (*C var*), 385
stats_hdr.s_name (*C var*), 385
stats_hdr.s_size (*C var*), 385
STATS_HDR_F_PERSIST (*C macro*), 381
STATS_INC (*C macro*), 382
STATS_INC_RAW (*C macro*), 382
STATS_INCN (*C macro*), 382
STATS_INCN_RAW (*C macro*), 381
stats_init (*C function*), 383
stats_init_and_reg (*C function*), 383
stats_mgmt_register_group (*C function*), 383
STATS_NAME (*C macro*), 382
STATS_NAME_END (*C macro*), 382
STATS_NAME_INIT_PARMS (*C macro*), 382
stats_name_map (*C struct*), 384
stats_name_map.snm_name (*C var*), 385
stats_name_map.snm_off (*C var*), 385
STATS_NAME_MAP_NAME (*C macro*), 382
STATS_NAME_START (*C macro*), 382
stats_persist_flush (*C function*), 383
stats_persist_init (*C function*), 383
stats_persist_sched (*C function*), 383
STATS_PERSIST_SCHED (*C macro*), 383
STATS_PERSISTED_HDR (*C macro*), 383
stats_persisted_hdr (*C struct*), 385
stats_persisted_hdr.sp_hdr (*C var*), 385
stats_persisted_hdr.sp_persist_delay (*C var*), 385
stats_persisted_hdr.sp_persist_timer (*C var*), 385
STATS_PERSISTED_SECT_START (*C macro*), 382
stats_register (*C function*), 383

stats_reset (*C function*), 383
STATS_RESET (*C macro*), 381
STATS_SECT_DECL (*C macro*), 381
STATS_SECT_END (*C macro*), 381
STATS_SECT_ENTRY (*C macro*), 381
STATS_SECT_ENTRY16 (*C macro*), 381
STATS_SECT_ENTRY32 (*C macro*), 381
STATS_SECT_ENTRY64 (*C macro*), 381
STATS_SECT_START (*C macro*), 381
STATS_SECT_VAR (*C macro*), 381
STATS_SET (*C macro*), 381
STATS_SET_RAW (*C macro*), 381
stats_shell_register (*C function*), 383
STATS_SIZE_16 (*C macro*), 381
STATS_SIZE_32 (*C macro*), 381
STATS_SIZE_64 (*C macro*), 381
STATS_SIZE_INIT_PARMS (*C macro*), 381
stats_walk (*C function*), 383
stats_walk_func_t (*C type*), 383
STR (*C macro*), 560

tu_config (*C var*), 558
tu_post_test_fn_t (*C type*), 558
tu_pre_test_fn_t (*C type*), 558
tu_restart (*C function*), 558
tu_set_fail_cb (*C function*), 558
tu_set_pass_cb (*C function*), 558
tu_start_os (*C function*), 559
tu_suite_complete (*C function*), 558
tu_suite_failed (*C var*), 558
tu_suite_init (*C function*), 558
tu_suite_name (*C var*), 558
tu_suite_pre_test (*C function*), 558
tu_suite_register (*C function*), 558
tu_suite_set_pre_test_cb (*C function*), 559
tu_testsuite_fn_t (*C type*), 558
type (*C var*), 575

X

XSTR (*C macro*), 560

T

TEST_ASSERT (*C macro*), 560
TEST_ASSERT_FATAL (*C macro*), 560
TEST_ASSERT_FULL (*C macro*), 560
TEST_CASE (*C macro*), 559
TEST_CASE_DECL (*C macro*), 559
TEST_CASE_DEFN (*C macro*), 559
TEST_CASE_SELF (*C macro*), 559
TEST_CASE_SELF_EMIT_ (*C macro*), 559
TEST_CASE_TASK (*C macro*), 559
TEST_CASE_TASK_EMIT_ (*C macro*), 559
TEST_PASS (*C macro*), 560
TEST_SUITE (*C macro*), 559
TEST_SUITE_DECL (*C macro*), 559
TEST_SUITE_REGISTER (*C macro*), 559
ts_suite (*C struct*), 560
tu_any_failed (*C var*), 558
tu_case_complete (*C function*), 559
tu_case_fail (*C function*), 559
tu_case_fail_assert (*C function*), 559
tu_case_failed (*C var*), 558
tu_case_idx (*C var*), 558
tu_case_init (*C function*), 559
tu_case_jb (*C var*), 558
tu_case_name (*C var*), 558
tu_case_pass (*C function*), 559
tu_case_pass_manual (*C function*), 559
tu_case_post_test (*C function*), 559
tu_case_report_fn_t (*C type*), 558
tu_case_reported (*C var*), 558
tu_case_set_post_test_cb (*C function*), 559
tu_case_write_pass_auto (*C function*), 559
tu_config (*C struct*), 560