

An OS to build, deploy and securely manage billions of devices

Latest News:

Apache Mynewt 1.10.0, Apache NimBLE 1.5.0 (/download) released (May 6, 2022)

Docs (/documentation/) / Newt Tool Guide (index.html) / Theory of Operations

C Edit on GitHub (https://github.com/apache/mynewt-newt/edit/master/docs/newt_operation.rst)

Search documentation

Version: latest

Introduction (../index.html)

Setup & Get Started (../get_started/index.html)

Concepts (../concepts.html)

Tutorials (../tutorials/tutorials.html)

Third-party Resources (../external_links.html)

OS User Guide (../os/os_user_guide.html)

BLE User Guide (../network/index.html)

Newt Tool Guide (index.html)

Theory of Operations

Command Structure (newt_ops.html)

Install (install/index.html)

Newt Manager Guide (../newtmgr/index.html)

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.

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
 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.

```
$ 1s 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_oi
c testbench
       blemesh_light bleprph_oic
                                                                lorashell
blehci
                                   bleuart
                                             bsnprph iptest
                                                                               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 <code>pkg.yml</code> file. For example, blinky's <code>pkg.yml</code> 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 <code>syscfg.yml</code> file (stored in the targets/<target-name> directory). You can use the <code>newt target config show</code> command to see the configuration settings and values for your target, and use the <code>newt target set</code> command to set the <code>syscfg</code> variable and override the configuration setting values. You can also use an editor to create your target's <code>syscfg.yml</code> file and add the setting values to the file. See Compile-Time Configuration (../os/modules/sysinitconfig/sysinitconfig.html) for more information on system configuration settings.

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/
\sqsubseteq os
    — arch
        ├─ cortex m0
           ∟ os
               └─ os_arch.h
          - cortex_m4
           ∟ os
               └─ os_arch.h
         - mips
               └─ os_arch.h
         — sim
           ∟ os
               └─ os_arch.h
        └─ sim-mips
           ∟ os
               └─ os_arch.h
    — 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
    L— 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 <code>syscfg.yml</code> files. It generates a <code>syscfg.h</code> 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 Compile-Time Configuration (../os/modules/sysinitconfig/sysinitconfig.html) 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!

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/
\sqsubseteq targets
      my_blinky_sim
          — арр
              apps
                └─ blinky
                        apps
                        └─ blinky
                            L_ src
                                 ├─ main.d
                                  - main.o
                                └─ main.o.cmd
                      — apps_blinky.a
                      - apps_blinky.a.cmd
                      - blinky.elf
                      — blinky.elf.cmd
                      blinky.elf.dSYM
                        └─ Contents
                            ├─ Info.plist
                            └─ Resources
                                L- DWARF
                    ├─ blinky.elf.lst
                    └── manifest.json
               hw
                  bsp
                    └── native
                        — hw_bsp_native.a
                        — hw_bsp_native.a.cmd
                        └─ repos

    □ apache-mynewt-core

                                L-- hw
<snip>
```

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.

Download/Debug Support

Once a target has been built, there are a number of helper functions that work on the target. These are:

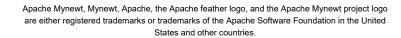
- load Download built target to board
- debug Open debugger session to target
- size Size of target components
- create-image Add image header to target binary
- run The equivalent of build, create-image, load, and debug on specified target
- target Create, delete, configure, and query a target

load and debug handles driving GDB and the system debugger. These commands call out to scripts that are defined by the BSP.

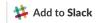
The idea is that every BSP will add support for the debugger environment for that board. That way common tools can be used across various development boards and kits.

Previous: Newt Tool Guide (index.html)

Next: Command Structure **②** (newt_ops.html)



APACHE SOFTWARE FOUNDATION http://www.apache.org/



()