NuttX Companion

Release 0.1.6

Jul 25, 2020

Contents:

Release v0.1.6. (*Quickstart*)

Apache NuttX is a very capable, configurable, fast, POSIX-compatible, internet-connected real-time operating system.

This book is meant to be a companion to the Apache NuttX Documentation. Hopefully it will provide some more help for people interested in learning about Apache NuttX, getting it running on their embedded hardware, and developing applications on it.

CHAPTER 1

User Guide

How to get NuttX, configure it, compile it, and install it on your embedded hardware

1.1 Introduction

This is the Apache NuttX Companion, a warm and friendly guide to all things Apache NuttX– how to install it, configure it, develop on it, debug it, improve it, get help from its community, and generally be have fun with a very capable, configurable, fast, POSIX-compatible, internet-connected real-time operating system.

Hopefully this guide will complement the Apache NuttX documentation, filling in gaps when needed, and providing help for people wanting to get started with Apache NuttX.

If you have improvements, corrections, additions, or things you'd like to see covered here, let me know! Contributions to this Companion are also welcome and encouraged! See the section *Contributing* for more info.

Adam Feuer
adam@adamfeuer.com

1.1.1 License

1.2 Quickstart

Here's the quick version of getting things going. This is a bare-bones outline for experienced developers– if it's going too quickly, dive into the other sections of the Companion. This Quickstart guide assumes you're on a Linux computer, you're using an ARM processor on your embedded board, and you're familiar with using the command line.

1. Install a Cross-Compiler Toolchain

   With NuttX, you compile the operating system and your application on your desktop or laptop computer, then install the binary file on your embedded computer. This guide assumes your computer is an ARM CPU. If it isn't, you'll need a different tool chain.

   Download the right flavor of the ARM Embedded Gnu Toolchain for your embedded processor's CPU.

   Unpack it into `/opt/gcc` and add the bin directory to your path. For instance:

```
$ sudo mkdir /opt/gcc
$ sudo chgrp -R users /opt/gcc
$ cd /opt/gcc
$ wget https://developer.arm.com/-/media/Files/downloads/gnu-rm/9-2019q4/gcc-arm-
↪none-eabi-9-2019-q4-major-x86_64-linux.tar.bz2?revision=108bd959-44bd-4619-9c19-
↪26187abf5225&la=en&hash=E788CE92E5DFD64B2A8C246BBA91A249CB8E2D2D
$ tar xf gcc-arm-none-eabi-9-2019-q4-major-x86_64-linux.tar.bz2
$ # add the toolchain bin/ dir to your path...
$ # you can edit your shell's rc files if you don't use bash
$ echo "export PATH=/opt/gcc/gcc-arm-none-eabi-9-2019-q4-major/bin:$PATH" >> ~/.
↪bashrc
```

2. Download Apache NuttX

```
$ mkdir nuttx
$ cd nuttx
$ git clone https://github.com/apache/incubator-nuttx.git nuttx
$ git clone https://github.com/apache/incubator-nuttx-apps apps
$ git clone https://starcat-io@bitbucket.org/nuttx/tools.git tools
```

3. Install the `kconfig-frontends` package

   This is necessary to use the menuconfig system to configure NuttX, and includes the `kconfig-tweak` utility that can be used to quickly change debug settings.

```
$ cd tools/kconfig-frontends
$ # on MacOS do the following:
$ patch < ../kconfig-macos.diff -p 1
$ ./configure --enable-mconf --disable-shared --enable-static --disable-gconf --
↪disable-qconf --disable-nconf
$ # on Linux do the following:
$  ./configure --enable-mconf --disable-nconf --disable-gconf --disable-qconf
$ make
$ make install
```

4. List Possible Apache NuttX Base Configurations

   Find your hardware and a good starting application in the list of base configurations. In the application list, `nsh` is the Apache NuttX Shell, an interactive commandline that's a good starting place if you're new.

```
$ cd nuttx
$ ./tools/configure.sh list | less
```

5. Initialize Configuration

   Pick one of the board:application base configuration pairs from the list, and feed it to the configuration script. The `-l` tells us that we're on Linux. macOS and Windows builds are possible, this Companion doesn't cover them yet.

```
$ cd nuttx
$ ./tools/configure.sh -l <board-name>:<config-dir>
```

6. Customize Your Configuration (Optional)

   This step is optional. Right now, this is mainly to get familiar with how it works– you don't need to change any of the options now, but knowing how to do this will come in handy later.

   There are a lot of options. We'll cover a few of them here. Don't worry about the complexity– you don't have to use most of the options.

```
$ make menuconfig
```

7. Exit the menuconfig System

   Use the left or right arrow keys to select `<Exit>`, and save the configuration.

8. Compile Apache NuttX

```
$ make clean; make
```

9. Install the Executable Program on Your Board

   This step is a bit more complicated, depending on your board. It's covered in the section *Running Apache NuttX*.

1.3 Installing

To start developing on Apache NuttX, we need to get the source code, configure it, compile it, and get it uploaded onto an embedded computing board. These instructions are for Ubuntu Linux and macOS Catalina. If you're using a different version, you may need to change some of the commands.

1.3.1 Prerequisites

Install prerequisites for building and using Apache NuttX (Linux)

1. Install system packages

```
$ sudo apt install \
    bison flex gettext texinfo libncurses5-dev libncursesw5-dev \
    gperf automake libtool pkg-config build-essential gperf \
    libgmp-dev libmpc-dev libmpfr-dev libisl-dev binutils-dev libelf-dev \
    libexpat-dev gcc-multilib g++-multilib picocom u-boot-tools util-linux
```

1. Give yourself access to the serial console device

   This is done by adding your Linux user to the `dialout` group:

```
$ sudo usermod -a -G dialout $USER
$ # now get a login shell that knows we're in the dialout group:
$ su - $USER
```

Install prerequisites for building and using Apache NuttX (macOS)

```
$ brew tap discoteq/discoteq
$ brew install flock
$ brew install x86_64-elf-gcc  # Used by simulator
$ brew install u-boot-tools  # Some platform integrate with u-boot
```

Install prerequisites for building and using Apache NuttX (Windows – WSL)

If you are are building Apache NuttX on windows and using WSL follow that installation guide for Linux. This has been verified against the Ubunutu 18.04 version.

There may be complications interacting with programming tools over USB. Recently support for USBIP was added to WSL 2 which has been used with the STM32 platform, but it is not trivial to configure: https://github.com/rpasek/usbip-wsl2-instructions

Install prerequisites for building and using Apache NuttX (Windows – Cygwin)

Download and install Cygwin using the minimal installation in addition to these packages:

```
make              bison             libmpc-devel
gcc-core          byacc             automake-1.15
gcc-g++           gperf             libncurses-devel
flex              gdb               libmpfr-devel
git               unzip             zlib-devel
```

Install Required Tools (All Platforms)

There are a collection of required tools that need to be built to build most Apache NuttX configurations:

```
$ mkdir buildtools
$ export NUTTXTOOLS=`pwd`/buildtools
$ export NUTTXTOOLS_PATH=$NUTTXTOOLS/bin:$NUTTXTOOLS/usr/bin
$ git clone https://bitbucket.org/nuttx/tools.git tools
```

NOTE: You will need to add `NUTTXTOOLS_PATH` to your environment `PATH`

1. Install kconfig-frontends tool

This is necessary to run the `./nuttx/tools/configure.sh` script as well as using the ncurses-based menuconfig system.

```
$ cd tools/
$ cd kconfig-frontends
$ # on MacOS do the following:
$ patch < ../kconfig-macos.diff -p 1
$ ./configure --prefix=$NUTTXTOOLS --enable-mconf --disable-shared --enable-
↪static --disable-gconf --disable-qconf --disable-nconf
$ # on Linux do the following:
$ ./configure --prefix=$NUTTXTOOLS --enable-mconf --disable-gconf --disable-
↪qconf
$ touch aclocal.m4 Makefile.in
$ make
$ make install
```

1. Install gperf tool (optional)

```
$ cd tools/
$ wget http://ftp.gnu.org/pub/gnu/gperf/gperf-3.1.tar.gz
$ tar zxf gperf-3.1.tar.gz
$ cd gperf-3.1
$ ./configure --prefix=$NUTTXTOOLS
$ make
$ make install
```

1. Install gen-romfs (optional)

```
$ cd tools/
$ tar zxf genromfs-0.5.2.tar.gz
$ cd genromfs-0.5.2
$ make install PREFIX=$NUTTXTOOLS
```

1.3.2 Get Source Code (Stable)

Apache NuttX releases are published on the project Downloads page and distributed by the Apache mirrors. Be sure to download both the nuttx and apps tarballs.

1.3.3 Get Source Code (Development)

Apache NuttX is actively developed on GitHub. If you want to use it, modify it or help develop it, you'll need the source code.

You can either clone the public repositories:

```
$ mkdir nuttx
$ cd nuttx
$ git clone https://github.com/apache/incubator-nuttx.git nuttx
$ git clone https://github.com/apache/incubator-nuttx-apps apps
```

Or, download the tarball:

```
$ curl -OL https://github.com/apache/incubator-nuttx/tarball/master
$ curl -OL https://github.com/apache/incubator-nuttx-apps/tarball/master
# optionally, zipball is also available (for Windows users).
```

Later if we want to make modifications (for instance, to improve NuttX and save them in our own branch, or submit them back to the project), we can do that easily. It's covered in the section *Making Changes*.

1.3.4 Install a Cross-Compiler Toolchain

With Apache NuttX, you compile the operating system and your application on your desktop or laptop computer, then install the binary file on your embedded computer. This guide assumes your computer is an ARM CPU. If it isn't, you'll need a different tool chain.

> There are hints on how to get the latest tool chains for most supported architectures in the Apache NuttX CI helper script and Docker container

Download the right flavor of the ARM Embedded Gnu Toolchain for your embedded processor's CPU.

Unpack it into `/opt/gcc` and add the bin directory to your path. For instance:

```
$ usermod -a -G users $USER
$ # get a login shell that knows we're in this group:
$ su - $USER
$ sudo mkdir /opt/gcc
$ sudo chgrp -R users /opt/gcc
$ sudo chmod -R u+rw /opt/gcc
$ cd /opt/gcc
$ HOST_PLATFORM=x86_64-linux   # use "mac" for macOS.
$ # For windows there is a zip instead (gcc-arm-none-eabi-9-2019-q4-major-
↪win32.zip)
$ curl -L -o gcc-arm-none-eabi-9-2019-q4-major-${HOST_PLATFORM}.tar.bz2␣
↪https://developer.arm.com/-/media/Files/downloads/gnu-rm/9-2019q4/gcc-arm-
↪none-eabi-9-2019-q4-major-${HOST_PLATFORM}.tar.bz2
$ tar xf gcc-arm-none-eabi-9-2019-q4-major-${HOST_PLATFORM}.tar.bz2
$ # add the toolchain bin/ dir to your path...
$ # you can edit your shell's rc files if you don't use bash
$ echo "export PATH=/opt/gcc/gcc-arm-none-eabi-9-2019-q4-major/bin:$PATH" >>␣
↪~/.bashrc
```

1.4 Compiling

Now that we've installed Apache NuttX prerequisites and downloaded the source code, we are ready to compile the source code into an executable binary file that can be run on the embedded board.

1. List Possible Apache NuttX Base Configurations

   Find your hardware and a good starting application in the list of base configurations. In the application list, `nsh` is the Apache NuttX Shell, an interactive commandline that's a good starting place if you're new.

   ```
   $ cd nuttx
   $ ./tools/configure.sh list | less
   ```

2. Initialize Configuration

   Pick one of the board:application base configuration pairs from the list, and feed it to the configuration script. The `-l` tells use that we're on Linux. macOS and Windows builds are possible, this Companion doesn't cover them yet.

   ```
   $ cd nuttx
   $ # this is the basic layout of the command:
   $ # ./tools/configure.sh -l <board-name>:<config-dir>
   $ # for example:
   $ ./tools/configure.sh -l sama5d3-xplained:nsh
   ```

3. Customize Your Configuration (Optional)

   This step is optional. Right now, this is mainly to get familiar with how it works– you don't need to change any of the options now, but knowing how to do this will come in handy later.

   There are a lot of options. We'll cover a few of them here. Don't worry about the complexity– you don't have to use most of the options.

   ```
   $ make menuconfig
   ```

4. Compile NuttX

```
$ make clean; make
```

5. Install the Executable Program on Your Board

   This step is a bit more complicated, depending on your board. It's covered in the section *Running Apache NuttX*.

1.5 Running

Embedded boards have different ways to get your program onto them and get them running. This guide assumes your board has a JTAG connector, and you have a JTAG hardware debugger like a Segger J-Link. JTAG is a set of standards that let you attach a hardware device to your embedded board, and then remotely control the CPU. You can load code, start, stop, step through the program, and examine variables and memory.

1. Attach the Debugger Cables
2. Start the Debugger

   Refer to your JTAG debugger's documentation for information on how to start a GDB Server process that gdb can communicate with to load code and start, stop, and step the embedded board's CPU. Your command line may be different from this one.

   ```
   $ JLinkGDBServer -device ATSAMA5D27 -if JTAG -speed 1000 -JTAGConf -1,-1
   ```

3. Launch the Gnu Debugger

   In another terminal window, launch the GDB for your platform. In the case of this guide, this came with the ARM Embedded Gnu Toolchain we downloaded in the Install step.

   ```
   $ arm-none-eabi-gdb
   ```

4. Connect to the board's serial console

   Usually you connect a USB-to-serial adapter to the board's serial console so you can see debug logging or execute Apache NuttX Shell (nsh) commands. You can access the serial console from Linux with the `picocom` terminal program. From another terminal, do this:

   ```
   $ picocom -b 115200 /dev/ttyUSB0
   ```

5. Set gdb to talk with the J-Link

   ```
   (gdb) target extended-remote :2331
   ```

6. Reset the board

   ```
   (gdb) mon reset
   ```

7. You may need to switch to the serial console to hit a key to stop the board from booting from its boot monitor (U-Boot, in the case of the SAMA5 boards from Microchip).
8. Halt the board

   ```
   (gdb) mon halt
   ```

9. Load nuttx

```
(gdb) file nuttx
(gdb) load nuttx
`/home/adamf/src/nuttx-sama5d36-xplained/nuttx/nuttx' has changed; re-
↪reading symbols.
Loading section .text, size 0x9eae4 lma 0x20008000
Loading section .ARM.exidx, size 0x8 lma 0x200a6ae4
Loading section .data, size 0x125c lma 0x200a6aec
Start address 0x20008040, load size 654664
Transfer rate: 75 KB/sec, 15587 bytes/write.
(gdb)
```

10. Set a breakpoint

```
(gdb) breakpoint nsh_main
```

11. Start nuttx

```
(gdb) continue
Continuing.

Breakpoint 1, nsh_main (argc=1, argv=0x200ddfac) at nsh_main.c:208
208         sched_getparam(0, &param);
(gdb) continue
Continuing.
```

1.5.1 Debugging Shortcuts

Note that you can abbreviate `gdb` commands, `info b` is a shortcut for `information breakpoints`; `c` works the same as `continue`, etc.

1.6 Configuring

Apache NuttX is a very configurable operating system. Nearly all features can be configured in or out of the system. This makes it possible to compile a build tailored for your hardware and application. It also makes configuring the system complex at times.

There is a configuration system that can be used on the commandline or in a GUI. I've found the easiest way to configured Apache NuttX is to use the `menuconfig` system. This is used via a terminal program and allows quick access to all of Apache NuttX's features via a system of menus.

The Apache NuttX configuration system uses Linux's kconfig system adapted for use with Apache NuttX. Here's info on Linux's kconfig menuconfig system.

After you've configured your board (see *Compiling*), you can use the menuconfig system to change the configuration. Once you've configured, you can compile to make a build that has your configuration options selected.

1. Initialize Board Configuration

   Here we'll use the simulator since that's the simplest to explain. You can do this with any board and base configuration. Note here you should be supplying *configure.sh* the correct flag for your build environment:

```
-l selects the Linux (l) host environment.
-m selects the macOS (m) host environment.
-c selects the Windows host and Cygwin (c) environment.
```

(continues on next page)

(continued from previous page)

```
-u selects the Windows host and Ubuntu under Windows 10 (u) environment.
-g selects the Windows host and MinGW/MSYS environment.
-n selects the Windows host and Windows native (n) environment.
```

Select the simulator configuration for a Linux host:

```
$ cd nuttx
$ make distclean  # make a clean start, clearing out old configurations
$ ./tools/configure.sh -l sim:nsh
  Copy files
  Select CONFIG_HOST_LINUX=y
  Refreshing...
```

2. Make

```
$ make clean; make
$ ./nuttx
login:
```

From another terminal window, kill the simulator:

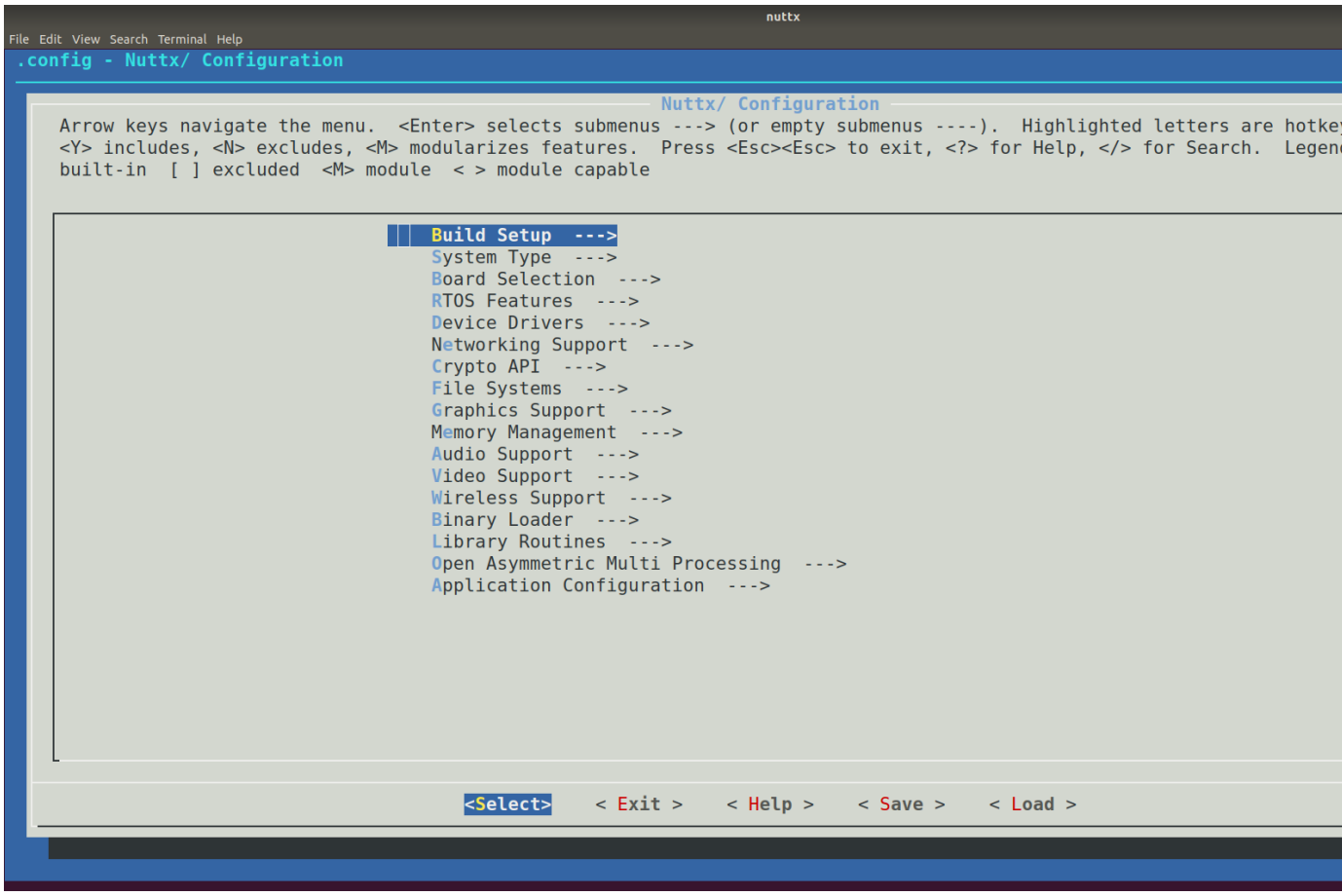
```
$ pkill nuttx
```

3. Menu Configuration

Showing that `login:` is annyoing. Let's use the `menuconfig` system to turn it off.

```
$ make menuconfig
```

Here's what you should see:



4. Application Configuration

   The NSH Login setting is under `Application Configuration > NSH Library`. Use the up and down arrows to navigate to `Application Configuration`; hit `<return>` to select it. Now you're in the `Application Configuration` menu. Use the arrows to go down to `NSH Library` and select that. Now navigate down to `Console Login` and use the spacebar to uncheck that setting (so that it has a blank space instead of a star in it).

   Now let's save. Use the right and left arrow keys to select the `Exit` menu item at the bottom of the screen. Hit `<return>` to select it, hit `<return>` again, and again, finally hitting `<return>` in the `Save Configuration` dialog box.

5. Make the New Configuration

   ```
   $ make clean; make
   ```

6. Run

   ```
   $ ./nuttx
   NuttShell (NSH) NuttX-8.2
   MOTD: username=admin password=Administrator
   ```

   Success!

   If you find that message of the day (MOTD) annoying and want to turn that off, it's configured in `Application Configuration > NSH Library >> Message of the Day (MOTD)`.

1.7 Debugging

Finding and fixing bugs is an important part of the hardware and software development process. Sometimes you also need to use debugging techniques to understand how the system works. Two tools that are helpful are debug logging and debugging using the Gnu Debugger (gdb).

1.7.1 Debug Logging

NuttX has a powerful logging facility with `info`, `warn`, and `error` levels. You can enable debugging for your build for the `net` feature (TCP/IP stack) by putting the following lines in your `.config` file:

```
CONFIG_DEBUG_ALERT=y
CONFIG_DEBUG_FEATURES=y
CONFIG_DEBUG_ERROR=y
CONFIG_DEBUG_WARN=y
CONFIG_DEBUG_INFO=y
CONFIG_DEBUG_ASSERTIONS=y
CONFIG_DEBUG_NET=y
CONFIG_DEBUG_NET_ERROR=y
CONFIG_DEBUG_NET_WARN=y
CONFIG_DEBUG_NET_INFO=y
CONFIG_DEBUG_SYMBOLS=y
CONFIG_DEBUG_NOOPT=y
CONFIG_SYSLOG_TIMESTAMP=y
```

Note that turning all these to `y` will produce an incredible amount of logging output. Set the level you want and the area you're interested in to `y`, and the rest to `n`, and then recompile. You can see the full list of debug feature areas in the file debug.h.

Timestamps can be enabled by setting `CONFIG_SYSLOG_TIMESTAMP=y`.

You may need to do a little bit of experimenting to find the combination of logging settings that work for the problem you're trying to solve. See the file debug.h for available debug settings that are available. This can also be configured via the `menuconfig` system.

There are also subsystems that enable USB trace debugging, and you can log to memory too, if you need the logging to be faster than what the console can output.

1.7.2 Changing Debug Settings Quickly

You can use the `kconfig-tweak` script that comes with the `kconfig-frontends` tools to quickly change debug settings, for instance turning them on or off before doing a build:

```
$ kconfig-tweak --disable CONFIG_DEBUG_NET
$ kconfig-tweak --enable CONFIG_DEBUG_NET
```

You can put a bunch of these into a simple script to configure the logging the way you want:

```
#!/bin/bash

$ kconfig-tweak --disable CONFIG_DEBUG_ALERT
$ kconfig-tweak --disable CONFIG_DEBUG_FEATURES
$ kconfig-tweak --disable CONFIG_DEBUG_ERROR
$ kconfig-tweak --disable CONFIG_DEBUG_WARN
$ kconfig-tweak --disable CONFIG_DEBUG_INFO
```

(continues on next page)

(continued from previous page)

```
$ kconfig-tweak --disable CONFIG_DEBUG_ASSERTIONS
$ kconfig-tweak --disable CONFIG_DEBUG_NET
$ kconfig-tweak --disable CONFIG_DEBUG_NET_ERROR
$ kconfig-tweak --disable CONFIG_DEBUG_NET_WARN
$ kconfig-tweak --disable CONFIG_DEBUG_NET_INFO
$ kconfig-tweak --disable CONFIG_DEBUG_SYMBOLS
$ kconfig-tweak --disable CONFIG_DEBUG_NOOPT
$ kconfig-tweak --disable CONFIG_SYSLOG_TIMESTAMP
```

1.7.3 Custom Debug Logging

Sometimes you need to see debug logs specific to your feature, and you don't want the rest of the built-in logs because they're either not relevant or have too much information. Debugging using logs is surprisingly powerful.

You can add your own custom debug logging by adding the following lines to debug.h:

```
/* after the CONFIG_DEBUG_WATCHDOG_INFO block near line 721 */
#ifdef CONFIG_DEBUG_CUSTOM_INFO
#  define custinfo    _info
#else
#  define custinfo    _none
#endif
```

You need to add the following line to your `.config` file:

```
CONFIG_DEBUG_CUSTOM_INFO=y
```

You would use it like this:

```
/* Custom debug logging */
custinfo("This is a custom log message.");
custinfo("Custom log data: %d", my_integer_variable);
```

1.7.4 JTAG Debugging

JTAG is a set of standards that specify a way to attach a hardware device to your embedded board, and then remotely control the CPU. You can load code, start, stop, step through the program, and examine variables and memory.

This guide assumes your board has a JTAG connector, and you have a JTAG hardware debugger like a Segger J-Link or OpenOCD.

The NuttX operating system uses threads, so you need a thread-aware debugger to do more than load code, start, and stop it. A thread-aware debugger will allow you to switch threads to the one that is running the code you're interested in, for instance your application, or an operating system network thread. So far, OpenOCD is the only supported NuttX thread-aware debugger.

You will need an OpenOCD-compatible hardware adapter, ideally a fast one (USB 2.0 High Speed). This guide assumes you are using the Olimex ARM USB TINY H. (Olimex ARM USB TINY H on Amazon.) Other adapters may work, follow the OpenOCD instructions and the instructions that came with your adapter.

You'll need to use the Sony fork of OpenOCD. Download and install it according to the OpenOCD instructions.

See this article for more info: Debugging a Apache NuttX target with GDB and OpenOCD.

See the section *Running* for a brief tutorial on how to use GDB.

1.8 Simulator

Apache NuttX has a simulator that can run as a regular program on Linux, Mac, and Windows computers. It's useful for debugging operating system features that aren't associated with particular device drivers— for instance the TCP/IP stack itself, a web interface or API for your application, or other communication protocols. It's also handy for trying out Apache NuttX without having a piece of embedded hardware.

This guide assumes you're on Linux. It works on Windows and Mac too— if you know how, submit a PR the NuttX Companion to update this guide!

1.8.1 Compiling

1. Configure the Simulator

   There are a lot of simulator configurations available that set you up to test various operating system features.

   Here we'll use the `sim:tcpblaster` configuration because it comes with networking that is ready to use.

   ```
   $ cd nuttx
   $ ./tools/configure.sh sim:tcpblaster
   ```

2. Compile

   ```
   $ make clean; make
   ```

1.8.2 Running

1. Give the Simulator Privileges

   On recent Linux distributions, you need to give the `nuttx` program the capabilities (similar to permissions) to access the network:

   ```
   $ sudo setcap cap_net_admin+ep ./nuttx
   ```

2. Run the simulator:

   ```
   $ ./nuttx
   ```

3. Bring Up the Network Interfaces

   On Apache NuttX:

   ```
   nsh> ifup eth0
   ```

   On Linux, first you need to find your main network interface— this will usually either be an ethernet or wireless network adapter. Do this:

   ```
   $ ifconfig
   lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
           inet 127.0.0.1  netmask 255.0.0.0
           inet6 ::1  prefixlen 128  scopeid 0x10<host>
           loop  txqueuelen 1000  (Local Loopback)
           RX packets 5846  bytes 614351 (614.3 KB)
           RX errors 0  dropped 0  overruns 0  frame 0
           TX packets 5846  bytes 614351 (614.3 KB)
   ```

(continues on next page)

(continued from previous page)

```
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlp0s20f3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.209  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::1161:c26b:af05:d784  prefixlen 64  scopeid 0x20<link>
        ether 24:41:8c:a8:30:d1  txqueuelen 1000  (Ethernet)
        RX packets 219369  bytes 176416490 (176.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 108399  bytes 27213617 (27.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

`lo0` is the Loopback Interface, so `wlp0s20f3` is the wireless interface. Note that it has an IP address on the local net. There may be other interfaces listed, you'll need to pick the one that's right for your system.

Then, on Linux do this to set up the tap network interface and route that will let the Apache Nuttx simulator access the network:

```
$ sudo ./tools/simhostroute.sh wlp0s20f3 on
$ ping -c 1 10.0.1.2  # nuttx system
PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.
64 bytes from 10.0.1.2: icmp_seq=1 ttl=64 time=7.52 ms

--- 10.0.1.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.529/7.529/7.529/0.000 m
```

4. Test that Apache NuttX can access the Internet

   First let's ping the network interface of our Linux host to prove we can see the gateway to the Internet:

   ```
   nsh> ping -c 1 10.0.1.1
   nsh> ping -c 1 10.0.1.1
   PING 10.0.1.1 56 bytes of data
   56 bytes from 10.0.1.1: icmp_seq=0 time=0 ms
   1 packets transmitted, 1 received, 0% packet loss, time 1010 ms
   ```

   Now let's ping one of Google's DNS servers to prove we can access the rest of the Internet:

   ```
   nsh> ping -c 1 8.8.8.8
   PING 8.8.8.8 56 bytes of data
   56 bytes from 8.8.8.8: icmp_seq=0 time=10 ms
   1 packets transmitted, 1 received, 0% packet loss, time 1010 ms
   ```

   Success!

1.8.3 Stopping

The only really effective way to stop the simulator is kill it from another terminal:

```
$ pkill nuttx
```

1.8.4 Debugging

You can debug the simulator like any regular Linux program.

1.9 Drivers

Some NuttX boards don't have full support for all the on-chip peripherals. If you need support for this hardware, you will either need to port a driver from another chip, or write one yourself. This section discusses how to do that.

1.9.1 Porting a Driver

Often support for on-chip peripherals exists in a closely related chip, or even a different family or from a different manufacturer. Many chips are made up of different Intellectual Property (IP) blocks that are licensed from vendors like Cadence, Synopsys, and others. The IP blocks may be similar enough to use another chip's driver with little modification.

- Find a similar driver in NuttX source code:
  - Look at register names listed in the datasheet for the peripheral.
  - Search the NuttX codebase for the register names (try several different ones).
  - Note that you'll have to compare the datasheet to the header and code files to see if there are differences; there will usually be some differences between architectures, and they can be significant.
- Find a similar driver in U-Boot source code:
  - Only for inspiration, you can't copy code because of license incompatibility.
  - But you can debug to see how the driver works.
  - U-Boot drivers are often easier to understand than BSD Unix drivers because U-Boot is simpler.
- Find a similar driver in Zephyr or BSD Unix (OpenBSD, FreeBSD, NetBSD):
  - If you find one, you can borrow code directly (Apache 2.0 and BSD licenses are compatible).
- Understanding how the driver works

  Here are a couple of techniques that helped me.

  - printf debugging
    - Sprinkle `custinfo()` logging statements through your code to see execution paths and look at variables while the code is running. The reason to use `custinfo()` as opposed to the other logging shortcuts (`mcinfo()`, etc.) is that you can turn on and off other other logging and still see your custom debug logging. Sometimes it's useful to quiet the flood of logging that comes from a particular debug logging shortcut.
    - Note that printing info to the console will affect timing.
    - Keep a file with just your debug settings in it, like this (`debugsettings`):

      ```
      CONFIG_DEBUG_CUSTOM_INFO=y
      (etc..)
      ```

    - Add the settings to the end of your `.config` file after running `make menuconfig` (that will reorder the file, making it harder to see and change the debug settings if you need to).

      ```
      $ cat .config debugsettings > .config1 ; mv .config1 .config
      ```

    - If you are using interrupts and threads (many things in NuttX run in different threads as a response to interrupts), you can use printf debugging to see overlapping execution.
      - Interrupts - here's how to inspect the C stack frame to see what execution environment is currently running:

```
uint32_t frame = 0;  /* MUST be the very first thing in the function */
uint32_t p_frame;
frame++;             /* make sure that frame doesn't get optimized out␣
↪*/
p_frame = (uint32_t)(&frame);
custinfo("p_frame: %08x\n", p_frame);
```

· Threads - here's how to get the thread identifier to see which thread is currently executing:

```
pthread_t thread_id = pthread_self();
custinfo("pthread_id: %08x\n", thread_id);
```

* stack frame printf
* thread printf

– GDB — the Gnu Debugger

GDB is a great tool. In this guide we've already used it to load our program and run it. But it can do a lot more. You can single-step through code, examine variables and memory, set breakpoints, and more. I generally use it from the commandline, but have also used it from an IDE like JetBrains' Clion, where it's easier to see the code context.

One add-on that I found to be essential is the ability to examine blocks of memory, like buffers that NuttX uses for reading and writing to storage media or network adapters. This Stack Overflow question on using GDB to examine memory includes a GDB command that is very handy. Add this to your `.gdbinit` file, and then use the `xxd` command to dump memory in an easy-to-read format:

Here's a short GDB session that shows what this looks like in practice. Note that the memory location being examined (`0x200aa9e0` here) is a buffer being passed to `mmcsd_readsingle`:

1.9.2 NuttX Drivers as a Reference

If you're not porting a NuttX driver from another architecture, it still helps to look at other similar NuttX drivers, if there are any. For instance, when implementing an Ethernet driver, look at other NuttX Ethernet drivers; for an SD Card driver, look at other NuttX Ethernet drivers. Even if the chip-specific code won't be the same, the structure to interface with NuttX can be used.

1.9.3 Using Chip Datasheets

To port or write a driver, you'll have to be familiar with the information in the chip datasheet. Definitely find the datasheet for your chip, and read the sections relevant to the peripheral you're working with. Doing so ahead of time will save a lot of time later.

Another thing that's often helpful is to refer to sample code provided by the manufacturer, or driver code from another operating system (like U-Boot, Zephyr, or FreeBSD) while referring to the datasheet — seeing how working code implements the necessary algorithms often helps one understand how the driver needs to work.

- How to use a datasheet

  Key pieces of information in System-on-a-Chip (SoC) datasheets are usually:

  – Chip Architecture Diagram — shows how the subsections of the chip (CPU, system bus, peripherals, I/O, etc.) connect to each other.

  – Memory Map — showing the location of peripheral registers in memory. This info usually goes into a header file.

- DMA Engine — if Direct Memory Access (DMA) is used, this may have info on how to use it.
- Peripheral — the datasheet usually has a section on how the peripheral works. Key parts of this include:
  * Registers List — name and offset from the base memory address of the peripheral. This needs to go into a header file.
  * Register Map — what is the size of each register, and what do the bits mean? You will need to create `#defines` in a header file that your code will use to operate on the registers. Refer to other driver header files for examples.

1.9.4 Logic analyzers

For drivers that involve input and output (I/O), especially that involve complex protocols like SD Cards, SPI, I2C, etc., actually seeing the waveform that goes in and out the chip's pins is extremely helpful. Logic analyzers can capture that information and display it graphically, allowing you to see if the driver is doing the right thing on the wire.

1.9.5 DMA Debugging

- Dump registers before, during, and after transfer. Some NuttX drivers (`sam_sdmmc.c` or `imxrt_sdmmc.c` for example) have built-in code for debugging register states, and can sample registers before, during, and immediately after a DMA transfer, as well as code that can dump the peripheral registers in a nicely-formatted way onto the console device (which can be a serial console, a network console, or memory). Consider using something like this to see what's happening inside the chip if you're trying to debug DMA transfer code.
- Compare register settings to expected settings determined from the datasheet or from dumping registers from working code in another operating system (U-Boot, Zephyr, FreeBSD, etc.).
- Use the `xxd` GDB tool mentioned above to dump NuttX memory buffers before, during, and after a transfer to see if data is being transferred correctly, if there are over- or under-runs, or to diagnose data being stored in incorrect locations.
- printf debugging register states can also help here.
- Remember that logging can change the timing of any algorithms you might be using, so things may start or stop working when logging is added or removed. Definitely test with logging disabled.

1.10 Making Changes

If you want to make changes to NuttX, for your own personal use, or to submit them back to project to improve NuttX, that's easy. For the purposes of this guide, you'll need a GitHub account, since the Apache NuttX team uses GitHub. (You could also use git locally, or save your changes to other sites like GitLab or BitBucket, but that's beyond the scope of this guide).

Here's how to do it:

1. Set your git user name and email

```
$ cd nuttx/
$ git config --global user.name "Your Name"
$ git config --global user.email "yourname@somedomaincom"
```

2. Sign in to GitHub

   If you don't have a GitHub account, it's free to sign up.

3. Fork the Project

   Visit both these links and hit the Fork button in the upper right of the page:

   - NuttX
   - NuttX Apps

4. Change the Git Remotes

   The git repositories in your project are currently connected to the official NuttX repositories, but you don't have permission to push software there. But you can push them to your forks, and from there create Pull Requests if you want to send them to the NuttX project.

   First, remove the current remote, `origin` (we'll add it back later):

   ```
   $ cd nuttx/
   $ # display the remote
   $ git remote -v
   ```

   You should see something like this:

   ```
   origin   https://github.com/apache/incubator-nuttx.git
   ```

   Now, on the GitHub web page for your forked `incubator-nuttx` project, copy the clone url – get it by hitting the green `Clone or Download` button in the upper right. Then do this:

   ```
   $ git remote rm origin
   $ git remote add origin <your forked incubator-nuttx project clone url>
   $ git remote add upstream https://github.com/apache/incubator-nuttx.git
   ```

   Do the same for your forked `incubator-nuttx-apps` project:

   ```
   $ cd ../apps
   $ git remote rm origin
   $ git remote add origin <your forked incubator-nuttx-apps project clone␣
   ↪url>
   $ git remote add upstream https://github.com/apache/incubator-nuttx-apps.
   ↪git
   ```

5. Create a Local Git Branch

   Now you can create local git branches and push them to GitHub:

   ```
   $ git checkout -b test/my-new-branch
   $ git push
   ```

1.10.1 Git Workflow With an Upstream Repository

The main NuttX git repository is called an "upstream" repository - this is because it's the main source of truth, and its changes flow downstream to people who've forked that repository, like us.

Working with an upstream repo is a bit more complex, but it's worth it since you can submit fixes and features to the main NuttX repos. One of the things you need to do regularly is keep your local repo in sync with the upstream. I work with a local branch, make changes, pull new software from the upstream and merge it in, maybe doing that several times. Then when everything works, I get my branch ready to do a Pull Request. Here's how:

1. Fetch upstream changes and merge into my local master:

```
$ git checkout master
$ git fetch upstream
$ git merge upstream/master
$ git push
```

2. Merge my local master with my local branch:

```
$ git checkout my-local-branch
$ git merge master
$ git push
```

3. Make changes and push them to my fork

```
$ vim my-file.c
$ git add my-file.c
$ git commit my-file.c
$ git push
```

4. Repeat 1-3 as necessary

5. Run the `tools/checkpatch.sh` script on your files

   When your code runs, then you're almost ready to submit it. But first you need to check the code to ensure that it conforms to the NuttX Coding Standard. The `tools/checkpatch.sh` script will do that. Here's the usage info:

```
$ ./tools/checkpatch.sh -h
USAGE: ./tools/checkpatch.sh [options] [list|-]

Options:
-h
-c spell check with codespell(install with: pip install codespell
-r range check only (used with -p and -g)
-p <patch list> (default)
-g <commit list>
-f <file list>
-  read standard input mainly used by git pre-commit hook as below:
   git diff --cached | ./tools/checkpatch.sh -
```

   Run it against your files and correct all the errors in the code you added, so that `tools/checkpatch.sh` reports no errors. Then commit the result. For example:

```
$ ./tools/checkpatch.sh -f my-file.c
arch/arm/src/sama5/hardware/my-file.c:876:82: warning: Long line found
$ # fix errors
$ vim my-file.c
$ # run again
$ ./tools/checkpatch.sh -f my-file.c
```

   If you have made a lot of changes, you can also use this bash commandline to see the errors for all the changed C files in your branch (assumes you are currently on the branch that has the changed files):

```
$ git diff --name-only master | egrep "\.c|\.h" | xargs echo | xargs ./
↪tools/checkpatch.sh -f | less
```

   Note that there are some bugs in the `nxstyle` program that `checkpatch.sh` uses, so it may report a few errors that are not actually errors. The committers will help you find these. (Or view the nxstyle Issues.)

6. Commit the fixed files

```
$ git add my-file.c
$ git commit my-file.c
$ git push
```

1.10.2 Submitting Your Changes to NuttX

> Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.
>
> (from GitHub's About pull requests page)

Before you do a Pull Request, the NuttX team will usually want all the changes you made in your branch "squashed" into a single commit, so that when they review your changes, there's a clean view of the history. If there are changes after Pull Request review feedback, they can be separate commits. Here's the easiest way I found to do that initial squash before submitting the Pull Request:

1. Check out my branch

   ```
   $ git checkout my-branch
   ```

2. Rename it to `my-branch-old` to save it, because we're going to create new clean branch to push:

   ```
   $ git branch -m my-branch-old
   ```

3. Create a new clean branch with the same name you were using before the last step:

   ```
   $ git checkout master
   $ git checkout -b my-branch
   ```

4. Merge your saved old branch into the new one, telling git to "squash" all your commits into one (note this will not commit the result; the changed files will be in your staging area, ready to be committed):

   ```
   $ git merge --squash my-branch-old
   ```

5. Commit the result

   ```
   $ git commit
   ```

6. Force-push your new clean branch to the remote— this will overwrite all your previous changes in that branch:

   ```
   $ git push -f --set-upstream origin my-branch
   ```

7. Create a GitHub Pull Request

   A Pull Request is how you ask your upstream to review and merge your changes.

   Here's GitHub's instructions for creating a Pull Request.

8. Get Pull Request feedback and implement changes

   Get suggestions for improvements from reviewers, make changes, and push them to the branch. Once the reviewers are happy, they may suggest squashing and merging again to make a single commit. In this case you would repeat steps 1 through 6.

1.10.3 Git Resources

- Git Cheat Sheet (by GitHub)
- Git Book (online)

1.11 Resources

Here's a list of Apache NuttX resources that you might find helpful:

- Apache NuttX
  - NuttX website
  - Apache NuttX website
  - Apache NuttX online documentation
  - Apache NuttX mailing list – a very active mailing list, the place to get help with your application or any questions you have about NuttX.
  - Apache NuttX YouTube channel – Alan Carvalho de Assis's YouTube channel on NuttX. It's a source of a lot of great practical information.
  - Apache NuttX Coding Standard — How code should look when you submit new files or modify existing ones.
  - Apache NuttX Code Contribution Guidlines — The full workflow to follow for submitting code with all the details.
- Git
  - Git Cheat Sheet (by GitHub)
  - Git Book (online)

1.12 Contributing

The Apache NuttX Companion is made using the Sphinx documentation system. Sphinx documentation is written in ReStructured Text (RST). RST is the format used for Python documentation and is also used in many other projects.

Contributions and fixes to the Apache NuttX Companion are encouraged and welcome. Here's how to do it.

1. Fork the Apache NuttX Companion Repository

   Visit this link and hit the Fork button in the upper right of the page:

   - NuttX Companion

2. Clone the Forked Repository

   Click the "Clone or Download" button and copy the clone URL. Then do this:

   ```
   $ git clone <CLONE-URL-THAT-YOU-COPIED>
   ```

3. Install Sphinx

   On the command line, change directories to the newly-downloaded repository directory:

```
$ cd nuttx-companion
$ # install pyenv
$ curl -L https://github.com/pyenv/pyenv-installer/raw/master/bin/pyenv-
↪installer | bash
$ # install python
$ pyenv install 3.6.9
$ # install pipenv
$ pip install pipenv
$ # install sphinx
$ pipenv install
$ # activate the virtual environent
$ pipenv shell
```

4. Build the HTML Documentation Locally

```
$ cd docs
$ make html
Running Sphinx v2.4.1
making output directory... done
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 12 source files that are out of date
updating environment: [new config] 12 added, 0 changed, 0 removed
reading sources... [100%] user/simulator
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] user/simulator
generating indices...  genindexdone
writing additional pages...  searchdone
copying static files... ... done
copying extra files... done
dumping search index in English (code: en)... done
dumping object inventory... done
build succeeded.

The HTML pages are in ``_build/html``.
```

5. Check Out a New Branch

```
$ git checkout -b feature/my-branch
```

6. Make Your Changes

```
$ vim user/quickstart.rst  # or use the editor of your choice
                           # on whatever file you want to change
```

7. Rebuild the HTML Documentation

```
$ make html
```

8. View the Documentation in a Web Browser

   You can open the file `docs/_build/html/index.html`.

9. Iterate

   Repeat Steps 6, 7, and 8 until you're happy with the result.

10. Commit the Changes

```
$ git add user/quickstart.rst  # or whatever files you changed
$ git commit
```

11. Push to Your Branch

```
$ git push
```

12. Submit Your Pull Request

    Go to the Apache NuttX Companion page on Github and click the "Pull Request" button. See Github's Creating a Pull Request page for more info.

    Use this template for the Pull Request description text:

```
### Summary
### Impact
### Limitations / TODO
### Detail
### Testing
### How To Verify
```

    Fill out the sections describing your changes. The summary should be a concise bulleted list.

    The `How To Verify` section is only needed if you change how the project is built or add some other programs or scripts.

13. Make Changes If Requested

    When you submit your Pull Request, the Apache NuttX Companion team will review the changes, and may request that you modify your submission. Please work with them to get your changes accepted.

14. You're Done!

    Feel good that you've made Apache NuttX documentation better for yourself and others. You've just made the world a better place!

1.12.1 Sphinx Resources

- Sphinx documentation system
- ReStructured Text documentation
- Sphinx Guide to ReStructured Text
- Restructured Text cheat sheet

CHAPTER 2

Indices and tables

- genindex
- modindex
- search