



Professional eBook

HOW TO BUILD YOUR OWN TAILOR-MADE IoT LINUX OS



Creating & Compiling Your Own Linux
on a Raspberry Pi 3 Using Buildroot
& U-Boot Bootloader

BROUGHT TO YOU BY FIREDOME

What follows is a comprehensive updated guide to building your own custom, embedded Linux for your next IoT device using the latest versions of Buildroot and U-Boot bootloader.

The learning process will go from installing Buildroot for customizing the most used aspects of any Linux system to how to create the different types of images and configure the networking. The settings shown are to run on a real target, specifically, the Raspberry Pi 3 model B board, due to its price and availability.

WHAT YOU ARE GOING TO LEARN

1. Serial connection using UART
2. Linux build systems
3. Toolchains
4. Compiling and configuring Linux kernels & modules
5. Pre-installing packages
6. Configuring system parameters
7. Flashing firmware image to an SD card
8. Bootloaders (changing default RPI bootloader to U-Boot)
9. Static compilation of packages

TABLE OF CONTENTS

PREREQUISITES..... p3

SETTING UP THE INFRASTRUCTURE..... p4

Serial Connection	p4
Connecting UART to RPI 3	p4
Configuring Raspberry Pi 3	p4
Raspberry Pi 3 UART pins	p5
Plug in the adapter/cable	p5
What's a Linux Build System?	p10
Buildroot	p11
Installing Mandatory Packages	p12
Installing Buildroot	p12
Buildroot Directory Structure	p12

CONFIGURING AND BUILDING YOUR LINUX SYSTEM..... p13

Configuring	p13
Target Options	p14
Build Options	p15
Toolchain	p15
What's a toolchain?	p15
Selecting toolchain on Buildroot	p16
System Configuration	p18
Custom scripts	p18
Root Filesystem Overlay	p18
Kernel	p19

Target packages	p20
Filesystem Images	p23
Bootloaders	p24
Building	p24
Compilation flow (source)	p25
Flashing The New Firmware	p28
Enabling UART output on the Raspberry Pi	p28
Mazel Tov! You've Built Your First Custom-made IoT Linux System	p29
Further reading	p31

REPLACING THE BOOTLOADER..... p31

RPI Bootloading Process	p31
Stage 1	p31
Stage 2	p32
Stage 3	p32
RPI Bootloading Process with U-Boot	p32
Compile U-Boot Using Buildroot	p33
Preparing Files	p34
Mazel Tov! You're Now Using U-Boot Bootloader	p35
Bonus: Static Compilation of Packages	p38
Compiling Netcat Statically	p38

SUMMARY..... p39

PREREQUISITES

1. Target device: Raspberry Pi 3 Model B
 - a. SD card + reader for your PC
2. [USB to TTL Adapter cable](#) ([FTDI adapter](#) could also work)
3. Build device: Ubuntu (18.04 LTS recommended, for Windows users, [WSL](#) works as well)

SETTING UP THE INFRASTRUCTURE

Serial Connection

To connect to control (read and write) the target device, you need some kind of a console. SSH will be the go-to solution when you have a fully running system with TCP/IP stack, network interfaces, IP configured and the right SSH server running. Unfortunately, this is not the case when you develop a system from scratch, you need a direct console connection to all stdin/out/err of the board, way before you even have an OS installed or an HDMI display driver loaded (dealing with bootloaders, for example), that's where UART comes into the picture:

UART (Universal Asynchronous Receiver/Transmitter) is a serial communication protocol in which data transfers serially, i.e., bit by bit. Asynchronous serial communication is common for byte-oriented transmission. In Asynchronous serial communication, a byte of data transfers at a time.

Connecting UART To RPI 3

Configuring Raspberry Pi 3

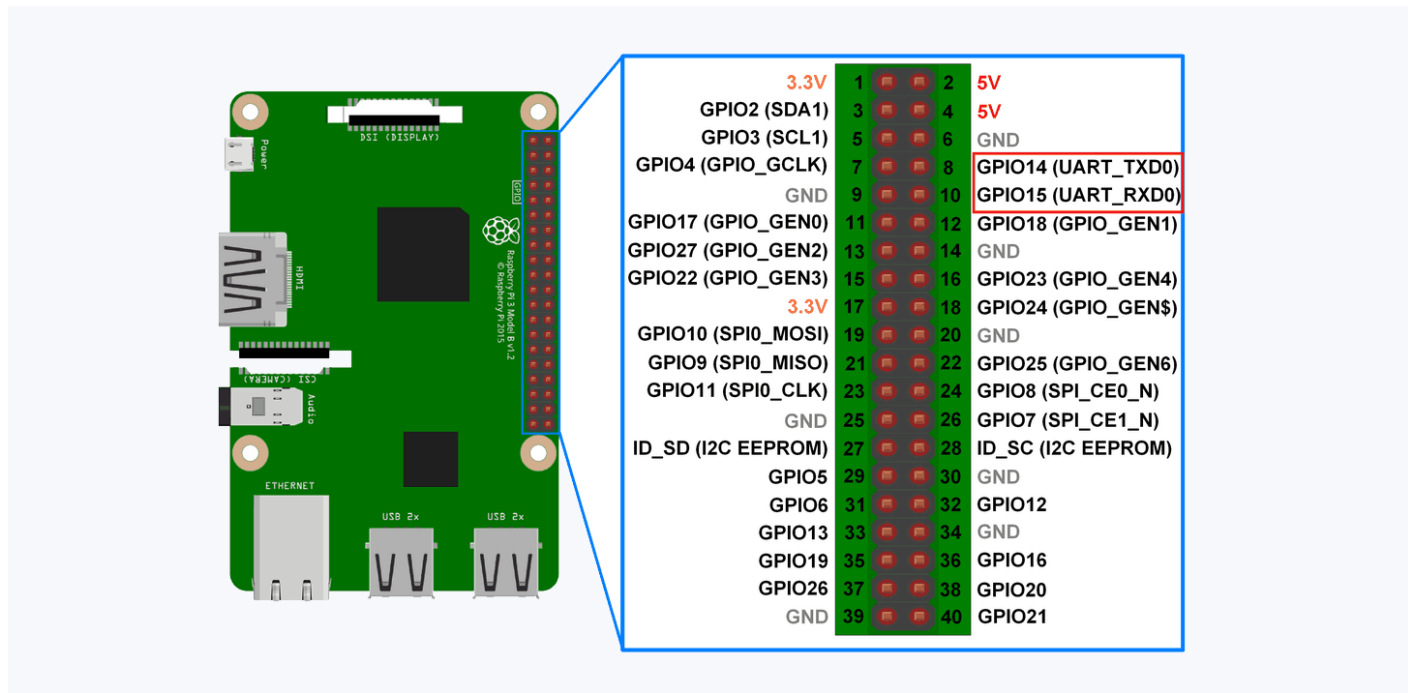
All you need to do is add the following line to the ***config.txt*** file on the /boot partition:

```
enable_uart=1
```

There's more on this after you create the Linux image.

Once you have set up everything, go to the next step. (Connecting will just give you a blank console at this point.)

Raspberry Pi 3 UART Pins



[Source](#)

Plug in the adapter/cable

If you're using USB to TTL, then simply connect the pins in the following order:



1. Connect the **black** wire to pin **#6** (ground).
2. Connect the **green** wire to pin **#8** (RXD to TXD)
3. Connect the **white** wire to pin **#10** (TXD to RXD)

If for some reason that doesn't work and you get a blank console, then switch the white and the green.

SETTING UP THE INFRASTRUCTURE

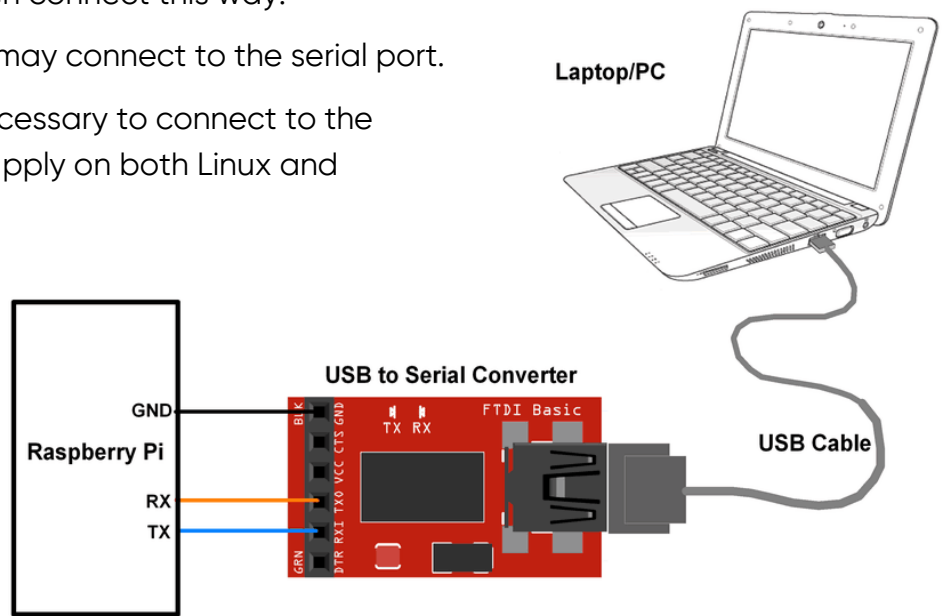
If you're using an FTDI board, then connect this way:

Once you plug in the cable, you may connect to the serial port.

The following parameters are necessary to connect to the Raspberry Pi console, and they apply on both Linux and Windows

[\(complete how-to connect to serial port on windows/linux\)](#):

- Speed (baud rate): 115200
- Bits: 8
- Parity: None
- Stop Bits: 1
- Flow Control: None



[Source](#)

Windows setup:

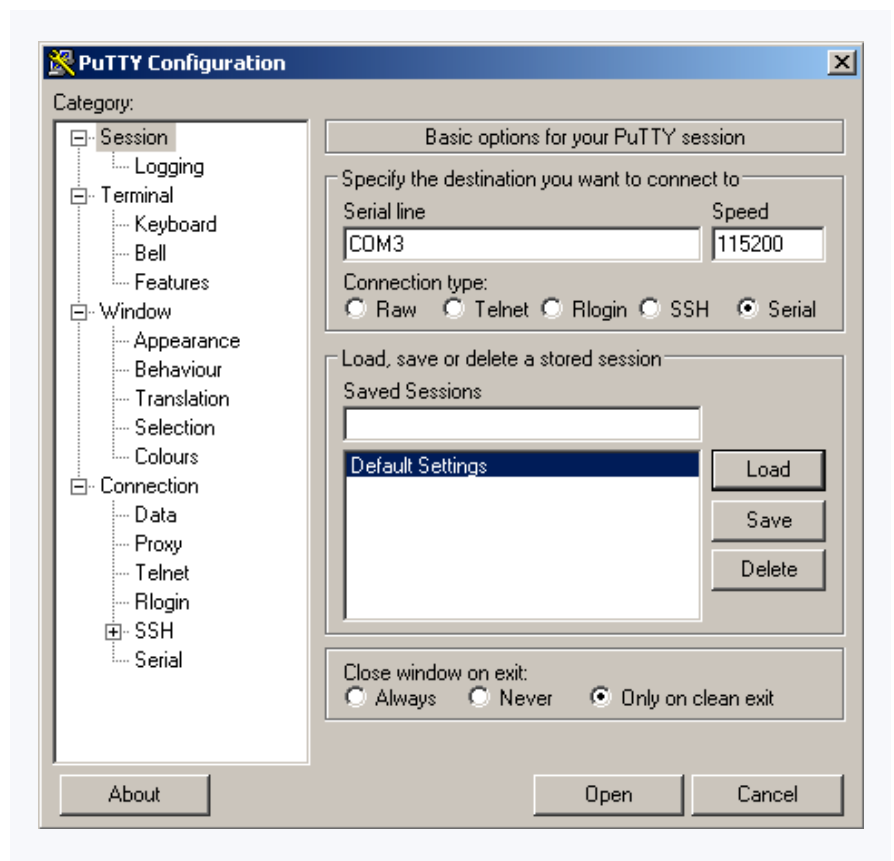
To know the port name of its serial port, simply run **mode** on a cmd:

```
C:\WINDOWS\system32\cmd.exe
C:\Users\shkedi>mode

Status for device COM3:
-----
Baud:          115200
Parity:        None
Data Bits:     8
Stop Bits:     1
Timeout:       ON
XON/XOFF:      OFF
CTS handshaking: OFF
DSR sensitivity: OFF
DTR circuit:   OFF
RTS circuit:   OFF

Status for device CON:
-----
Lines:         9001
Columns:       115
Keyboard rate: 31
Keyboard delay: 1
Code page:     437
```

Using PuTTY:



[Source](#)

Linux terminal setup:

To know the port name of its serial port:

- Built-in (standard) Serial Port: the Linux standard is `/dev/ttyS0`, `/dev/ttyS1`, and so on
- USB Serial Port Adapter: `/dev/ttyUSB0`, `/dev/ttyUSB1`, and so on.
 - Some types of USB serial adapter may appear as `/dev/ttyACM0` ...

You will need to be a member of the **dialout** group to access this port (for later releases, the required group is `tty`). You can check which is necessary with:

```
ls -l /dev/ttyUSB0
```

and you will see something like `"crw-rw----T 1 root dialout ..."`, `c` means character device, and `root` can "read,write" and the group **dialout** can "read,write" to the port and everyone else cannot access it.

To find out if you, the current user, is in the group **dialout**, use the command:

```
id
```

If you do not see **dialout** listed, add yourself with the command

```
sudo usermod -a -G dialout username
```

Connect to the terminal:

```
screen port_name 115200 # e.g. port_name == /dev/ttyUSB0
```

What's a Linux Build System?

First, you should read [this great intro by Free Electrons](#). (Slide 6 is enough to read for now.)

The development environment in embedded systems programming is usually quite different from the testing and production environments. They may use different chip architectures, software stacks, and even operating systems. Development workflows are different for embedded developers vs. desktop and web developers. Typically, the build output will consist of an entire software image for the target device, including the kernel, device drivers, libraries, and application software (and sometimes the bootloader).

1. Build systems allow an embedded Linux developer to generate a working embedded Linux system from scratch.

2. They automate the process of downloading, configuring, patching, compiling, and installing all the free software packages.
 - a. You have a well-known procedure for rebuilding your system from scratch.
 - b. You can easily integrate patches, bug fixes, or new upstream versions.
 - c. Your colleagues can easily take over your work since there's a documented procedure for system generation.
3. The build system already knows about most free software packages.
 - a. These packages manage dependencies and already solve cross-compiling issues.

Common build systems for IoT devices are Buildroot, Yocto, and OpenWRT (used mostly for routers and networking devices).

To run a Linux system in any target device, basically, three things are needed ([source](#)):

1. **Bootloader**, to load the kernel/different OS (discussion later)
2. **The Linux kernel**, which is the software the bootloader will copy to RAM memory and interact with the hardware, which gives the ability to execute programs and manage device drivers
3. **The root filesystem (RFS)**, which is the place where applications and system configuration are and where user files are stored

So, it's clear you need to generate both the kernel and the RFS; this is not an easy task in an embedded system due to the very specific nature of the involved hardware and the peripheral combinations. Some tools exist to make this task easier for the engineers; they're usually a collection of scripts that download, prepare, compile, and deploy the necessary tools and packages to finally get a binary image the target board will use.

Buildroot, under the GNU software license, is in this category. Created in 2001 by uClibc developers, it's for testing small embedded systems built with uClibc. Today, Peter Korsgaard is the chief maintainer with a very active community of users and contributors, and he is up-to-date with many new hardware drivers and software packages.

Buildroot

The [Buildroot project](#) is defined as "a simple, efficient, and easy-to-use tool to generate embedded Linux systems through cross-compilation." It shares many of the same objectives as the Yocto project, however, it focuses on simplicity and minimalism. In general, Buildroot will disable all optional compile-time settings for all packages (with a few notable exceptions), resulting in the smallest possible system. It's up to the system designer to enable the settings that are appropriate for a given device.

Buildroot is useful mainly for people working with embedded systems. Embedded systems often use processors that are not the regular x86 processors everyone is used to having in their PCs. They can be PowerPC processors, MIPS processors, ARM processors, etc.

Buildroot builds all components from the source but does not support on-target package management. As such, users sometimes call it a firmware generator since the images are largely fixed at build time. Applications can update the target filesystem, but there's no mechanism to install new packages into a running system.

The Buildroot output consists broadly of three components:

1. The root filesystem image and any other auxiliary files needed to deploy Linux to the target platform
2. The kernel, bootloader, and kernel modules appropriate for the target hardware
3. The toolchain used to build all the target binaries

Installing Mandatory Packages

You need to install the necessary packages that Buildroot needs to work in your Linux system (you can get the exhaustive list of software and libraries that Buildroot needs to work in the ***docs/manual/prerequisite.txt*** file of Buildroot). you can run the following command:

```
sudo apt-get install bison g++ flex gettext texinfo patch git-core libtool autoconf  
build-essential libncurses5-dev ssh minicom telnet ncurses-base ncurses-bin dialog  
screen git wget cpio unzip rsync bc
```

Installing Buildroot

```
wget https://buildroot.org/downloads/buildroot-2020.02.6.tar.gz  
tar xzvf buildroot-2020.02.6.tar.gz  
cd buildroot-2020.02.6
```

Buildroot Directory Structure

The Buildroot directory structure and their description:

- The **arch** directory: storage for configurations for all the supported architectures
- The **board** directory: storage for default configurations for different hardware platforms
- The **configs** directory: where to allocate generic configurations for kernel and packages
- The **dl** directory: where to copy sources and repositories, as the previous step before compiling
- The **docs** directory: contains documentation about Buildroot
- The **Linux** directory: contains sources and configurations to generate the Linux kernel
- The **toolchain** directory: contains the recipes to generate the toolchain that works with our platform
- The **output/images** directory: where to generate binary images; both RFS and Linux kernel will copy here

CONFIGURING AND BUILDING YOUR LINUX SYSTEM

Now, it's time to configure the Buildroot script for a Raspberry Pi 3 model B configuration. This will set up all the basic configuration options to work properly with our hardware. Buildroot already has some preconfigured settings for some of the most popular boards, including RPI, that you can use, but in this case to learn what to do, create your own Buildroot configuration.

The next steps will briefly and individually cover everything involved in the creation of system images. Buildroot performs the following steps when you execute it: first, it downloads all the related source files, compiles the toolchain that will be used to cross-compile the kernel and the rest of the applications, compiles the Linux kernel, and generates a basic RFS using the BusyBox tool. Finally, it compiles the configuration files and the third-party applications, and the system deploys according to the user configuration.

Configuring

Inside the Buildroot directory, **as non-root user** run:

```
make menuconfig
```

Please make sure to edit your configuration to match the one shown below.

```
home/sh/buildroot-2020.02.6/.config - Buildroot 2020.02.6 Configuration
Buildroot 2020.02.6 Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
submenus ----). Highlighted letters are hotkeys. Pressing <Y> selects a
feature, while <N> excludes a feature. Press <Esc><Esc> to exit, <?> for
Help, </> for Search. Legend: [*] feature is selected [ ] feature is
not selected

Target options --->
Build options --->
Toolchain --->
System configuration --->
Kernel --->
Target packages --->
Filesystem images --->
Bootloaders --->
Host utilities --->
Legacy config options --->

<Select> <Exit> <Help> <Save> <Load>
```

Target Options

For a Raspberry Pi 3 board, set the following target options parameters:

```
Target options
Arrow keys navigate the menu. <Enter> selects submenus
submenus ----). Highlighted letters are hotkeys. Press
feature, while <N> excludes a feature. Press <Esc><Esc>
Help, </> for Search. Legend: [*] feature is selected

Target Architecture (ARM (little endian)) --->
Target Binary Format (ELF) --->
Target Architecture Variant (cortex-A53) --->
Target ABI (EABIhf) --->
Floating point strategy (NEON/VFPv4) --->
ARM instruction set (ARM) --->
```

Build Options

```

Build options
s submenus ---> (or empty submenus --->). Highlighted letters are hotke
, <?> for Help, </> for Search. Legend: [*] feature is selected [ ] fe

Commands --->
(/home/sh/my_raspberrypi3_defconfig) Location to save buildroot config
$(TOPDIR)/dl) Download dir
$(BASE_DIR)/host) Host dir
Mirrors and Download locations --->
(0) Number of jobs to run simultaneously (0 for auto)
[*] Enable compiler cache
$(HOME)/.buildroot-ccache) Compiler cache location
() Compiler cache initial setup
[*] Use relative paths
[*] build packages with debugging symbols
[*] strip target binaries
() executables that should not be stripped
() directories that should be skipped when stripping
gcc optimization level (optimize for size) --->
libraries (shared only) --->
$(CONFIG_DIR)/local.mk) location of a package override file
() global patch directories
Advanced --->
*** Security Hardening Options ***
[ ] Build code with PIC/PIE
*** Stack Smashing Protection needs a toolchain w/ SSP ***
RELRO Protection (None) --->
*** Fortify Source needs a glibc toolchain and optimization ***

```

Toolchain

What's a toolchain?

In software, a toolchain is a set of programming tools used to perform a complex software development task or to create a software product, which is typically another computer program or a set of related programs.

A toolchain is a set of distinct software development tools that link (or chain) together at specific stages such as GCC, binutils and glibc (a portion of the GNU Toolchain). Optionally, a toolchain may contain other tools such as a debugger or a compiler for a specific programming language, such as C++. Quite often, the toolchain used for embedded development is a cross-toolchain, or more commonly known as a cross compiler. All the programs (like GCC) run on a host system of a specific architecture (such as x86), but they produce binary code (executables) to run on a different architecture (for example, ARM). This is cross-compilation and is the typical way of building embedded software. It's possible to compile natively, running GCC on your target ([source](#)).

More simply, the toolchain is the building block (set of libraries and software) to compile the Linux system for a CPU architecture different from yours.

Selecting toolchain on Buildroot

In this case, you'll use the built-in toolchain that comes with Buildroot, with the right kernel headers for the image.

Kernel Headers - What Are They?

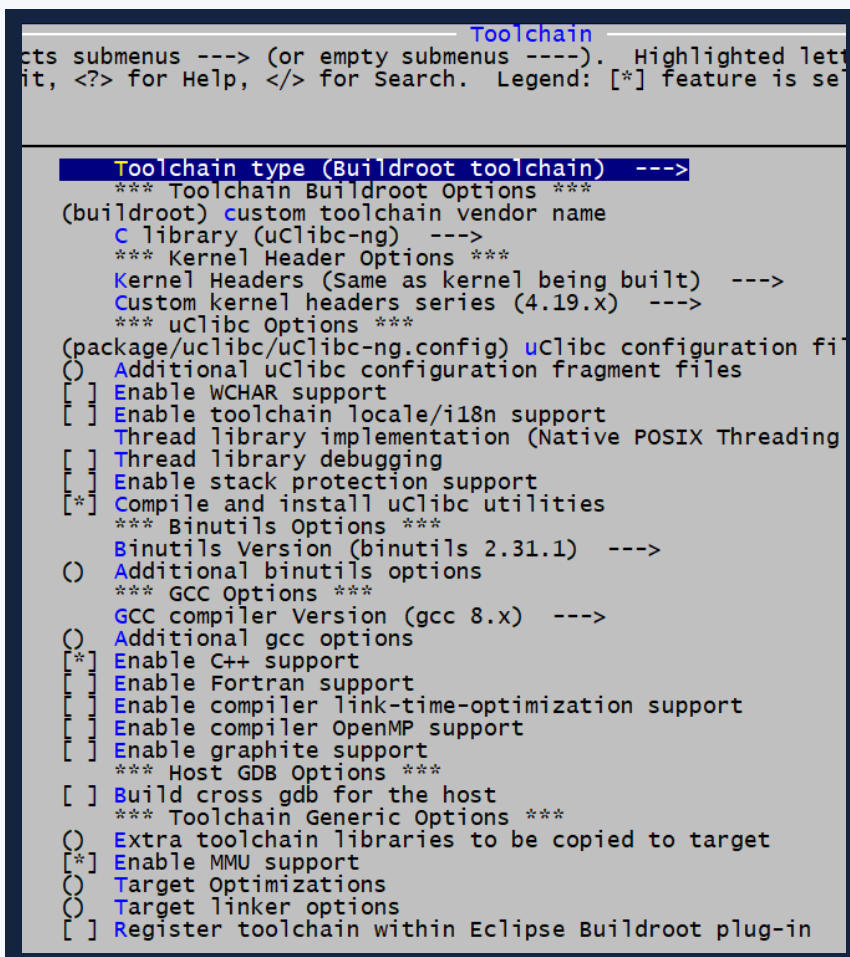
The header files define an interface: they specify how the functions in the source file are defined.

They are used so that a compiler can check if the usage of a function is correct as the function signature (return value and parameters) is present in the header file. For this task the actual implementation of the function is not necessary.

You could do the same with the complete kernel sources but you will install a lot of unnecessary files. Example: if I want to use the function

```
int foo(double param);
```

In a program, I do not need to know how the implementation of foo is, I just need to know that it accepts a single param (double) and returns an integer.



```
Toolchain
cts submenus ---> (or empty submenus ----). Highlighted letters
it, <?> for Help, </> for Search. Legend: [*] feature is selected

Toolchain type (Buildroot toolchain) --->
*** Toolchain Buildroot Options ***
(builtroot) custom toolchain vendor name
C library (uClibc-ng) --->
*** Kernel Header Options ***
Kernel Headers (Same as kernel being built) --->
Custom kernel headers series (4.19.x) --->
*** uClibc Options ***
(package/uClibc/uClibc-ng.config) uClibc configuration files
() Additional uClibc configuration fragment files
[*] Enable WCHAR support
[*] Enable toolchain locale/i18n support
[*] Thread library implementation (Native POSIX Threading)
[*] Thread library debugging
[*] Enable stack protection support
[*] Compile and install uClibc utilities
*** Binutils Options ***
Binutils Version (binutils 2.31.1) --->
() Additional binutils options
*** GCC Options ***
GCC compiler Version (gcc 8.x) --->
() Additional gcc options
[*] Enable C++ support
[*] Enable Fortran support
[*] Enable compiler link-time-optimization support
[*] Enable compiler OpenMP support
[*] Enable graphite support
*** Host GDB Options ***
[*] Build cross gdb for the host
*** Toolchain Generic Options ***
() Extra toolchain libraries to be copied to target
[*] Enable MMU support
() Target Optimizations
() Target linker options
[*] Register toolchain within Eclipse Buildroot plug-in
```

System Configuration

Entering fun stuff, configure a name, password, and other parameters for your OS:

```

System Configuration
s submenus ---> (or empty submenus --->). Highlighted letters are hotkeys. Pressing <Y> se
, <?> for Help, </> for Search. Legend: [*] feature is selected [ ] feature is excluded

Root FS skeleton (default target skeleton) --->
(shkediPi) System hostname
(Welcome to ShkediOS) System banner
  Passwords encoding (sha-256) --->
  Init system (BusyBox) --->
    /dev management (Dynamic using devtmpfs + mdev) --->
  (system/device_table.txt) Path to the permission tables
  [ ] support extended attributes in device tables
  [ ] Use symlinks to /usr for /bin, /sbin and /lib
  [*] Enable root login with password
  (1) Root password
    /bin/sh (busybox' default shell) --->
  [*] Run a getty (login prompt) after boot --->
  [*] remount root filesystem read-write during boot
  (eth0) Network interface to configure through DHCP
  (/bin:/sbin:/usr/bin:/usr/sbin) Set the system's default PATH
  [*] Purge unwanted locales
  (C en_US) Locales to keep
    *** NLS support needs a toolchain w/ wchar, dynamic library ***
  [ ] Install timezone info
  ( ) Path to the users tables
  ( ) Root filesystem overlay directories
  (board/raspberrypi3/post-build.sh) Custom scripts to run before creating filesystem images
  ( ) Custom scripts to run inside the fakeroot environment
  (board/raspberrypi3/post-image.sh) Custom scripts to run after creating filesystem images
  (--add-pi3-miniuart-bt-overlay) Extra arguments passed to custom scripts

```

Custom scripts

Custom scripts will configure (before and after building the image/kernel) certain parameters and change files, for example, if you use a specific board, you can configure its kernel boot parameters to enable/disable certain features that are specific to that chip.

Root filesystem overlay

A filesystem overlay is a tree of files that copies directly over the target filesystem after it's built. Filesystem overlays are simply directories that copy over the root filesystem at the end of the build.

You may use this feature to prepare files and directories that will be part of the final root filesystem (for example a `/etc/wpa_supplicant.conf` file containing the default Wi-Fi and password to connect to).

Kernel

```

Kernel
submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> sel
<?> for Help, </> for Search. Legend: [*] feature is selected [ ] feature is excluded

[*] Linux Kernel
Kernel version (Custom tarball) --->
($call github,raspberrypi,linux,5eeff139ea9135db6e01a58ef613338f1d0899d8)/linux-5eeff139ea9
() Custom kernel patches
Kernel configuration (Using an in-tree defconfig file) --->
(bcm2709) Defconfig name
() Additional configuration fragment files
() Custom boot logo file path
Kernel binary format (zImage) --->
Kernel compression format (gzip compression) --->
[*] Build a Device Tree Blob (DTB)
  DTB is built by kernel itself
(bcm2710-rpi-3-b bcm2710-rpi-3-b-plus bcm2710-rpi-cm3) In-tree Device Tree Source file names
() Out-of-tree Device Tree Source file paths
--*-- Build Device Tree with overlay support
[*] Install kernel image to /boot in target
[*] Needs host OpenSSL
[*] Needs host libelf
Linux Kernel Extensions --->
Linux Kernel Tools --->

```

Kernel version

Use the following kernel version (Choose "Custom tarball" and enter the following:

```
$(call github,raspberrypi,linux,5eeff139ea9135db6e01a58ef613338f1d0899d8)/ linux-5eeff139ea9135db6e01a58ef613338f1d0899d8.tar.gz
```

In-tree device tree source file names

Enter the following:

```
bcm2710-rpi-3-b bcm2710-rpi-3-b-plus bcm2710-rpi-cm3
```

Target packages

Here you'll choose which packages and binaries to install by default on the system.

```

Target packages
s submenus ---> (or empty submenus ----). Highlighted letters are hot
, <?> for Help, </> for Search. Legend: [*] feature is selected [ ]

-- BusyBox
(package/busybox/busybox.config) BusyBox configuration file to use?
() Additional BusyBox configuration fragment files
[ ] Show packages that are also provided by busybox
[ ] Individual binaries
[ ] Install the watchdog daemon startup script
Audio and video applications --->
Compressors and decompressors --->
Debugging, profiling and benchmark --->
Development tools --->
Filesystem and flash utilities --->
Fonts, cursors, icons, sounds and themes --->
Games --->
Graphic libraries and applications (graphic/text) --->
Hardware handling --->
Interpreter languages and scripting --->
Libraries --->
Mail --->
Miscellaneous --->
Networking applications --->
Package managers --->
Real-Time --->
Security --->
Shell and utilities --->
System tools --->
Text editors and viewers --->

```

Under **Hardware Handling > Firmware**, select:

```

→ Target packages → Hardware handling → Firmware
Firmware
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted le
excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature is s

[*] am33x-cm3
[ ] armbian-firmware
[ ] b43-firmware
[ ] linux-firmware
[ ] murata-cyw-fw
[*] rpi-bt-firmware
[*] rpi-firmware
    rpi variant (rpi 0/1/2/3) --->
    Firmware to boot (default) --->
[*] Install DTB overlays
    *** vcdbg needs a glibc toolchain w/ C++ ***
[*] rpi-wifi-firmware
[ ] sunxi script.bin board file
[ ] ts4900-fpga
[ ] ux500-firmware
[ ] wilc1000-firmware
[ ] wilink-bt-firmware
[ ] zd1211-firmware

```


Under **Networking Applications**, select:

```
→ Target packages → Networking applications Networking applications
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenu ----). Highlighted
excludes a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature is

[ ] (-)
[*] dnsmasq
[*] drbd-utils
[*] dropbear
  [*] client programs
  [*] disable reverse DNS lookups
  [*] optimize for size
  [*] log dropbear access to wtmp
  [*] log dropbear access to lastlog
  [*] enable legacy crypto
  [*] path to custom localoptions.h definitions file
[ ] ebttables
  *** ejabberd needs erlang, toolchain w/ C++ ***
[ ] ethtool
[ ] faifa
[ ] fastd
[ ] fcgiwrap
[ ] flannel
[ ] fping
  *** freeswitch needs a toolchain w/ C++, dynamic l
[ ] gerbera
  *** gesftpsrv needs a toolchain w/ wchar, thread
[ ] glorytun
  *** gupnp-tools needs libgtk3 ***
[ ] hans
[ ] haproxy
[ ] hiawatha
[ ] hostapd
  *** httping needs a toolchain w/ wchar ***
  *** i2pd needs a toolchain w/ C++, NPPL, wchar ***
[ ] ibrdtn-tools
  [*] ibrdtn
  [*] ifmetric
  [*] iftop
  [*] ifupdown scripts
```

```
[*] wpa_supplicant
  [*] Enable nl80211 support
  [*] Enable AP mode
    [*] Enable Wi-Fi Display
    [*] Enable mesh networking
  [*] Enable autoscan
  [*] Enable EAP
  [*] Enable HS20
  [*] Enable syslog support
  [*] Enable WPS
  [*] Enable WPA3 support
  [*] Install wpa_cli binary
  [*] Install wpa_client shared library
  [*] Install wpa_passphrase binary
  [*] Enable support for the DBus control interface
[*] wpa-tools
```

And select *iw* and *iputils* packages, as well.

Other than that, feel free to add any package you'd like :)

Filesystem Images

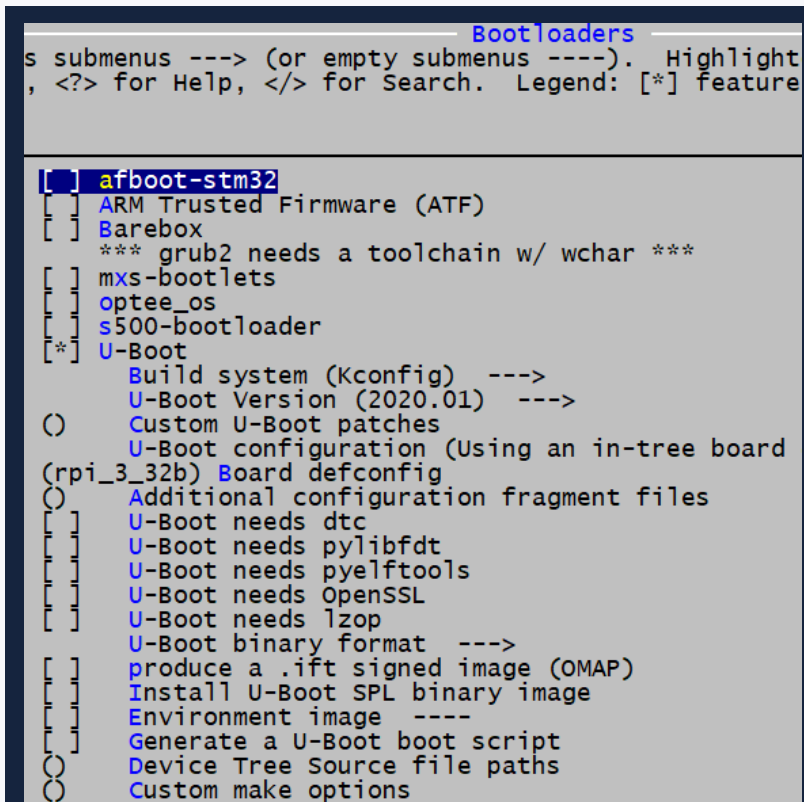
```
Filesystem images
ts submenus ---> (or empty submenus ----). Highlighted letters are ho
t, <?> for Help, </> for Search. Legend: [*] feature is selected [ ]

[*] axfs root filesystem
[ ] btrfs root filesystem
[ ] cloop root filesystem for the target device
[ ] cpio the root filesystem (for use as an initial RAM filesystem)
[ ] cramfs root filesystem
[*] ext2/3/4 root filesystem
    ext2/3/4 variant (ext4) --->
    ( ) filesystem label
    (120M) exact size
    (0) exact number of inodes (leave at 0 for auto calculation)
    (5) reserved blocks percentage
    (-O ^64bit) additional mke2fs options
    compression method (no compression) --->
[ ] f2fs root filesystem
[ ] initial RAM filesystem linked into linux kernel
[ ] jffs2 root filesystem
[ ] romfs root filesystem
[ ] squashfs root filesystem
[ ] tar the root filesystem
[ ] ubi image containing an ubifs root filesystem
[ ] ubifs root filesystem
[ ] yaffs2 root filesystem
```

Go with a hardcoded 120MB of / root partition space, using *ext4* format.

Bootloaders

Skip for now, we'll come back to this later.



```

Bootloaders
s submenus ---> (or empty submenus ----). Highlight
, <?> for Help, </> for Search. Legend: [*] feature

[*] afboot-stm32
  [ ] ARM Trusted Firmware (ATF)
  [ ] Barebox
  [ ] *** grub2 needs a toolchain w/ wchar ***
  [ ] mxs-bootlets
  [ ] optee_os
  [ ] s500-bootloader
  [*] U-Boot
    Build system (Kconfig) --->
    U-Boot Version (2020.01) --->
    ( ) Custom U-Boot patches
    U-Boot configuration (Using an in-tree board
    (rpi_3_32b) Board defconfig
    ( ) Additional configuration fragment files
    [ ] U-Boot needs dtc
    [ ] U-Boot needs pylibfdt
    [ ] U-Boot needs pyelftools
    [ ] U-Boot needs OpenSSL
    [ ] U-Boot needs lzop
    U-Boot binary format --->
    [ ] produce a .ift signed image (OMAP)
    [ ] Install U-Boot SPL binary image
    [ ] Environment image ----
    [ ] Generate a U-Boot boot script
    ( ) Device Tree Source file paths
    ( ) Custom make options
  
```

Building

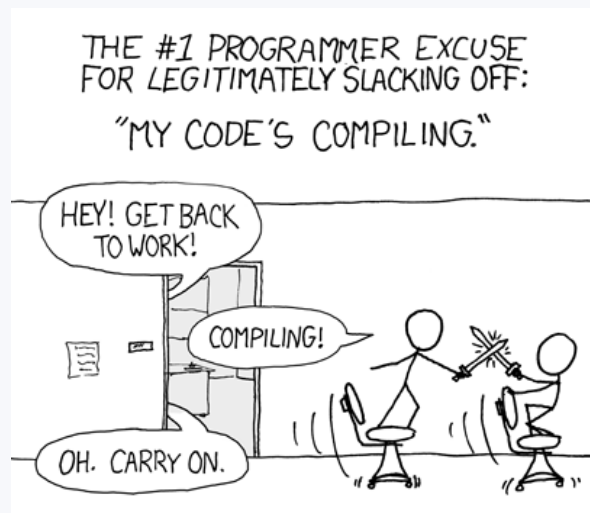
Save and exit, it's a good idea to copy your **.config** file to a separated space (git is best).

Now it's time for compilation, under the **buildroot-2020.02.6** directory; run:

```
make
```

Note: There is another way to better keep all your board-specific configs, packets and patches in a separate directory outside and setup BR2_EXTERNAL during the build. That allows updating buildroot easily and also keep modifications organized in one place.

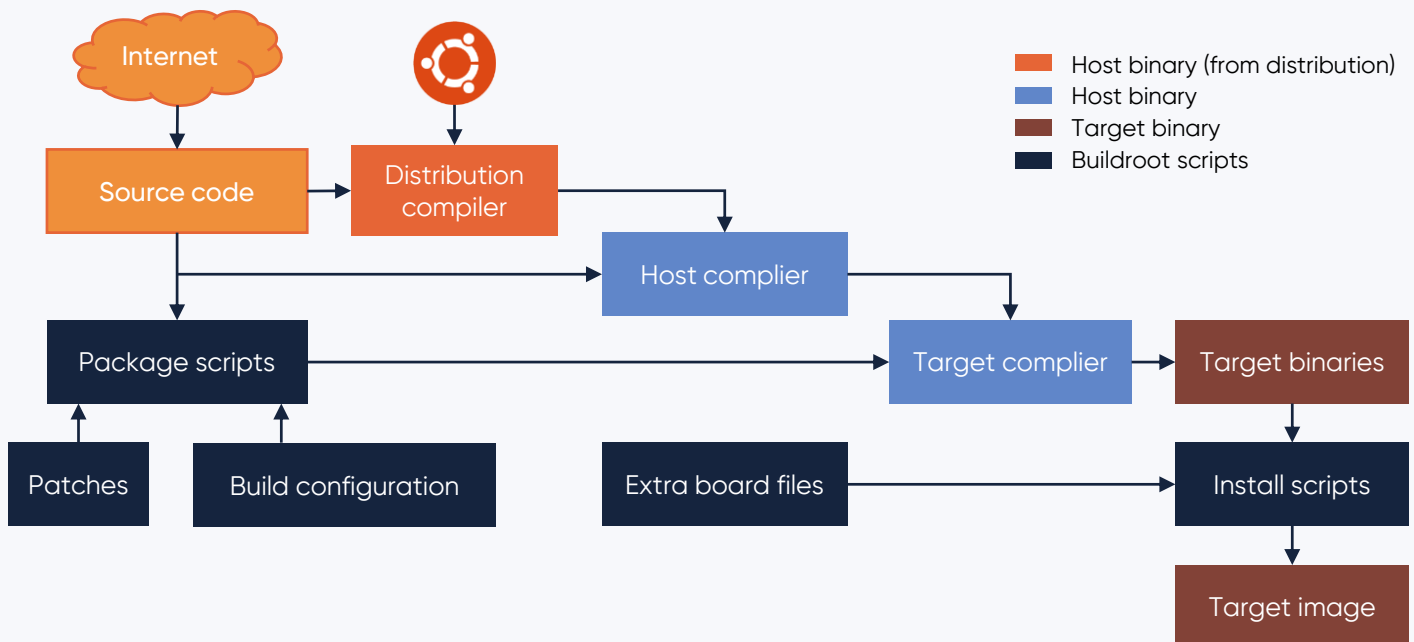
This step takes a lot of time to complete. Keep in mind that Buildroot scripts need to download all of the related resources, including the Linux kernel, and then compile everything. The first time you run this command, it will take even longer than usual just because the cross-compiling toolchain needs to be built, so be prepared to wait to have your system image.



If something fails for any reason and you can't understand why it's a good practice to run **make clean** to remove any files from the previous compilation that might interfere with your new settings (Buildroot can be problematic when recompiling over and over).

Compilation flow ([source](#))

From a high-level point of view, here is the workflow that Buildroot automates:



[Source](#)

1. Buildroot builds the toolchain, which consists of the cross-compilers and other tools it needs to compile the target system (green boxes).
2. The source code (blue boxes) for each piece of software downloads from the internet.
3. Using Buildroot scripts (gray boxes), the source is unpacked, patched, configured, compiled, and installed into the target output directory that forms the root filesystem ("rootfs") for the target (purple boxes).
4. Extra files, such as on-device configuration files, also copy into the target output directory.
5. Finally, scripts assemble the final firmware image from the generated rootfs.

After a while, you should see something like this:

```
INFO: vfat(boot.vfat): adding file 'rpi-firmware/overlays' as 'rpi-firmware/overlays'
INFO: vfat(boot.vfat): cmd: "MTTOOLS_SKIP_CHECK=1 mcopy -bsp -i '/home/sh/buildroot-2020
e/overlays' '::'" (stderr):
INFO: vfat(boot.vfat): adding file 'zImage' as 'zImage'
INFO: vfat(boot.vfat): cmd: "MTTOOLS_SKIP_CHECK=1 mcopy -bsp -i '/home/sh/buildroot-2020 '"
(stderr):
INFO: hdimage(sdcard.img): adding partition 'boot' (in MBR) from 'boot.vfat' ...
INFO: hdimage(sdcard.img): adding partition 'rootfs' (in MBR) from 'rootfs.ext4' ...
INFO: hdimage(sdcard.img): writing MBR
sh@LAPTOP-: ~/buildroot-2020.02.6$
```

These lines indicate the SD card image (*sdcard.img*) for the Pi has generated. It built the image from the root filesystem in *output/target/*, which you can inspect:

```
sh@LAPTOP-: ~/buildroot-2020.02.6$ ls -alh output/target/
total 72K
drwxr-xr-x 17 sh sh 4.0K Sep 23 23:09 .
drwxrwxr-x 6 sh sh 4.0K Sep 23 23:24 ..
-rw-r--r-- 1 sh sh 1.4K Sep 23 22:51 THIS_IS_NOT_YOUR_ROOT_FILESYSTEM
drwxr-xr-x 2 sh sh 4.0K Sep 23 23:23 bin
drwxr-xr-x 4 sh sh 4.0K Sep 5 22:10 dev
drwxr-xr-x 11 sh sh 4.0K Sep 23 23:24 etc
drwxr-xr-x 5 sh sh 4.0K Sep 23 23:23 lib
lrwxrwxrwx 1 sh sh 3 Sep 23 22:51 lib32 -> lib
lrwxrwxrwx 1 sh sh 11 Sep 23 23:09 linuxrc -> bin/busybox
drwxr-xr-x 2 sh sh 4.0K Sep 5 22:10 media
drwxr-xr-x 2 sh sh 4.0K Sep 5 22:10 mnt
drwxr-xr-x 2 sh sh 4.0K Sep 5 22:10 opt
drwxr-xr-x 2 sh sh 4.0K Sep 5 22:10 proc
drwxr-xr-x 2 sh sh 4.0K Sep 5 22:10 root
drwxr-xr-x 3 sh sh 4.0K Sep 23 23:09 run
drwxr-xr-x 2 sh sh 4.0K Sep 23 23:23 sbin
drwxr-xr-x 2 sh sh 4.0K Sep 5 22:10 sys
drwxr-xr-x 2 sh sh 4.0K Sep 5 22:10 tmp
drwxr-xr-x 7 sh sh 4.0K Sep 23 23:24 usr
drwxr-xr-x 4 sh sh 4.0K Sep 23 23:11 var
```

On the **output/image** directory you may see the firmware files:

```
sh@LAPTOP- [REDACTED]:~/buildroot-2020.02.6$ ls -alh output/images/
Total 175M
drwxr-xr-x 3 sh sh 4.0K Sep 23 23:24 .
drwxrwxr-x 6 sh sh 4.0K Sep 23 23:24 ..
-rw-r--r-- 1 sh sh 26K Sep 23 23:23 bcm2710-rpi-3-b-plus.dtb
-rw-r--r-- 1 sh sh 25K Sep 23 23:23 bcm2710-rpi-3-b.dtb
-rw-r--r-- 1 sh sh 24K Sep 23 23:23 bcm2710-rpi-cm3.dtb
-rw-r--r-- 1 sh sh 32K Sep 23 23:24 boot.vfat
-rw-r--r-- 1 sh sh 120K Sep 23 23:24 rootfs.ext2
lrwxrwxrwx 1 sh sh 11 Sep 23 23:24 rootfs.ext4 -> rootfs.ext2
drwxr-xr-x 3 sh sh 4.0K Sep 23 23:11 rpi-firmware
-rw-r--r-- 1 sh sh 153M Sep 23 23:22 sdcard.img
-rw-r--r-- 1 sh sh 467K Sep 23 23:12 u-boot.bin
-rw-r--r-- 1 sh sh 5.1M Sep 23 23:23 zImage
```

Looks good. Now, burn this image to the Pi's SD card and start connecting the Pi to the computer.

Flashing the New Firmware

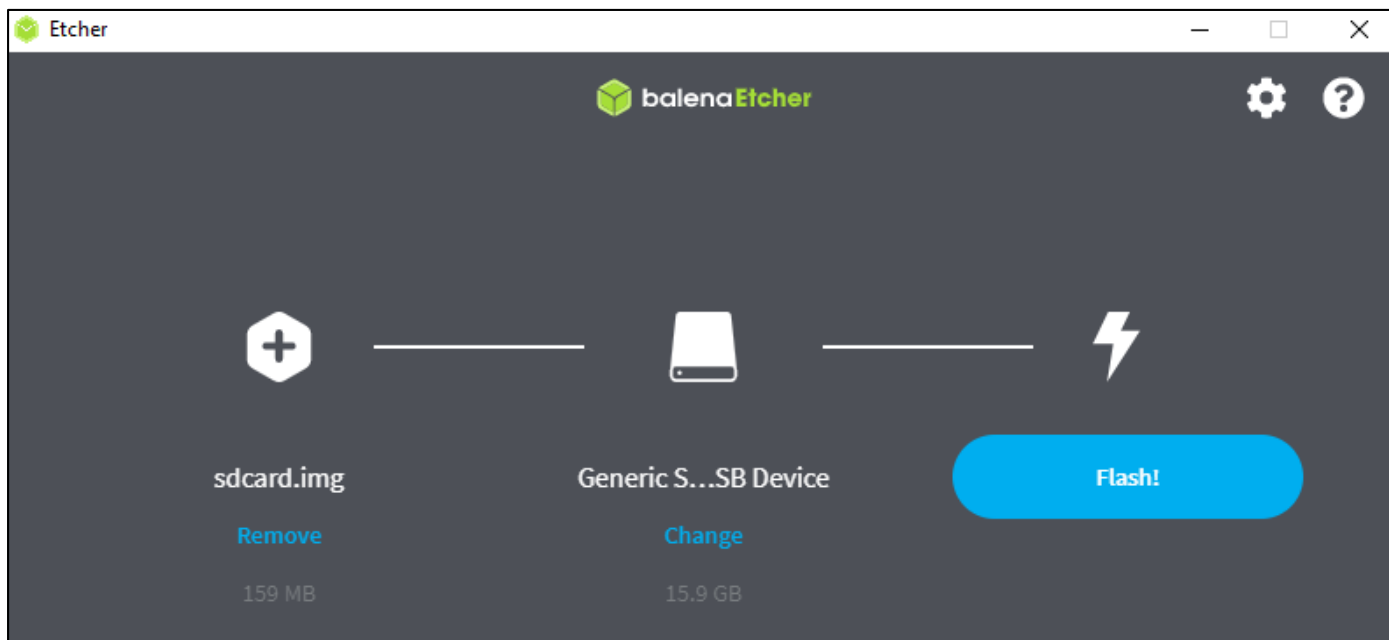
When Buildroot has completed the build process, the results are stored in the **output/images** directory. You will find the generated files in the **output/images** directory. The **zImage** file is the Linux kernel image that will load in the memory.

The **sdcard.img** file is the RFS (root file system) itself; you need to deflate this file in the SD card that will allocate your system. Also, the bootloader files generate inside the **output/images/rpi-firmware directory**; these files do not generate with the make command but download just as a binary format because of the closed nature of the GPU of Raspberry Pi.

The **bootcode.bin** file is the original bootloader that comes with Raspberry Pi.

The **sdcard.img** is the binary image you will flash to an SD memory card.

Use <https://www.balena.io/etcher/> for the flashing process (supports Mac, Linux, and Windows), which is much easier than using the **dd** command.



Enabling UART output on the Raspberry Pi:

To enable the serial interface on the RPI board, add the following line to the *config.txt* file on */boot* partition, so the file looks like this:

```
start_file=start.elf
fixup_file=fixup.dat
kernel=zImage
disable_overscan=1
gpu_mem_256=100
gpu_mem_512=100
gpu_mem_1024=100
enable_uart=1
```

Mazel Tov! You've Built Your First Custom-made IoT Linux System

Save all changes, if you're running on Linux, then type in the *sync* command to make sure all changes are flushed.

Eject the SD card safely, put it into the Raspberry Pi, make sure the UART cable is plugged in, and that using *screen/putty* you've connected to your *COM* or */dev/ttyS** port.

Turn on the board, and you should receive an output similar to the following:

```
COM4 - PuTTY
[ 3.538876] [vc_sm_connected_init]: installed successfully
[ 3.554277] bcm2835_mmal_vchiq: module is from the staging directory, the quality is unknown, you have been warned.
[ 3.580282] bcm2835_v4l2: module is from the staging directory, the quality is unknown, you have been warned.
[ 3.621471] bcm2835_codec: module is from the staging directory, the quality is unknown, you have been warned.
[ 3.645117] bcm2835-codec bcm2835-codec: Device registered as /dev/video10
[ 3.656053] bcm2835-codec bcm2835-codec: Loaded V4L2 decode
[ 3.667956] bcm2835-codec bcm2835-codec: Device registered as /dev/video11
[ 3.678744] bcm2835-codec bcm2835-codec: Loaded V4L2 encode
[ 3.690900] bcm2835-codec bcm2835-codec: Device registered as /dev/video12
[ 3.701431] bcm2835-codec bcm2835-codec: Loaded V4L2 isp
[ 3.813446] cfg80211: Loading compiled-in X.509 certificates for regulatory database
[ 4.218891] cfg80211: Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[ 4.232899] platform regulatory.0: Direct firmware load for regulatory.db failed with error -2
[ 4.248706] cfg80211: failed to load regulatory.db
[ 4.312259] brcmfmac: brcmf_fw_alloc_request: using brcm/brcmfmac43430-sdio for chip BCM43430/1
Saving random seed: [ 4.329083] usbcore: registered new interface driver brcmfmac
[ 4.334495] random: dd: uninitialized urandom read (512 bytes read)
OK
[ 4.369837] brcmfmac mmcl:0001:1: Direct firmware load for brcm/brcmfmac43430-sdio.txt failed with error -2
Starting network: [ 4.583101] NET: Registered protocol family 10
[ 4.592792] Segment Routing with IPv6
[ 4.713822] smsc95xx 1-1.1:1.0 eth0: hardware isn't capable of remote wakeup
[ 4.724885] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
udhcpd: started, v1.31.1
[ 4.763866] random: mktemp: uninitialized urandom read (6 bytes read)
udhcpd: sending discover
[ 5.394627] brcmfmac: brcmf_sdio htclk: HT Avail timeout (1000000): clkctl 0x50
[ 6.333118] smsc95xx 1-1.1:1.0 eth0: link up, 100Mbps, full-duplex, lpa 0x41E1
[ 6.346089] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
udhcpd: sending select for 192.168.0.106
udhcpd: lease of 192.168.0.106 obtained, lease time 86400
deleting routers
[ 8.248389] random: mktemp: uninitialized urandom read (6 bytes read)
adding dns 1.1.1.1
adding dns 8.8.8.8
Successfully initialized wpa_supplicant
rfkill: Cannot get wiphy information
Could not read interface wlan0 flags: No such device
WEXT: Could not set interface 'wlan0' UP
wlan0: Failed to initialize driver interface
FAIL
Starting dropbear sshd: [ 8.803654] random: dropbear: uninitialized urandom read (32 bytes read)
OK

ifconfig wlan0 192.168.0.106 up

[ 28.572048] random: crng init done
ssh-keygen: generating new host keys: RSA DSA ECDSA ED25519
Starting sshd: OK

Welcome to ShkediOS
shkediPi login: root
Password:
# uname -a
Linux shkediPi 4.19.23-v7 #1 SMP Thu Sep 17 18:53:56 EDT 2020 armv7l GNU/Linux
#
```


Further reading

1. [How Linux Works](#) by Brian Ward is a great book about many topics in Linux. It's relevant for desktop and server Linux, as well as embedded. It covers basics such as shell commands yet still goes into important, complex topics like the X11 window system and the DBus messaging bus. It's the right amount of detail to give you a good mental picture of how everything works, while still being approachable.
2. The very prolific team at Bootlin, a French company that does embedded Linux development maintains [Bootlin's Buildroot](#) training. You can pay the company to give your entire team training using this material. If you're willing to read through its slide decks, it's very thorough, although there's no lecture accompanying it.
3. [Packpub book](#) is useful for hands-on training.

The Buildroot user manual is the place to learn about hacking on Buildroot. The various areas of the build system are well-described and the reference manual for writing new packages is superb. The downside is that because it is a user manual, not a tutorial, it's quite dense. You'll definitely get familiar with it as you use Buildroot going forward.

REPLACING THE BOOTLOADER

Raspberry Pi comes with its own bootloader built-in, but on a real IoT device, you need to select and configure a bootloader that will load the kernel, so let's get things interesting.

RPI Bootloading Process

First, let's understand how the current RPI bootloader process works.

Raspberry Pi has a fairly complicated boot process with two bootloaders. The first one resides in built-in ROM and is responsible for starting the GPU. The GPU executes `bootcode.bin`, the second bootloader, which, in the end, runs the kernel.

Stage 1

To reduce cost, the Raspberry Pi (models A & B) omits any onboard, non-volatile memory used to store the bootloaders, Linux kernels, and file systems as seen in more traditional embedded systems. Rather, an SD/MMC card slot is for this purpose. The Raspberry Pi compute module has 4GB eMMC flash onboard.

Stage 1 boot is in the on-chip ROM. Loads Stage 2 in the L2 cache. The Raspberry Pi's Broadcom BCM2835 system on a chip (SoC) powers up with its ARM1176JZF-S 700 MHz processor held in reset. The VideoCore IV GPU core is responsible for booting the system. It loads the first stage bootloader from a ROM embedded within the SoC. The first stage bootloader is to load the second stage bootloader (bootcode.bin) from a FAT32 or FAT16 file system on the SD card.

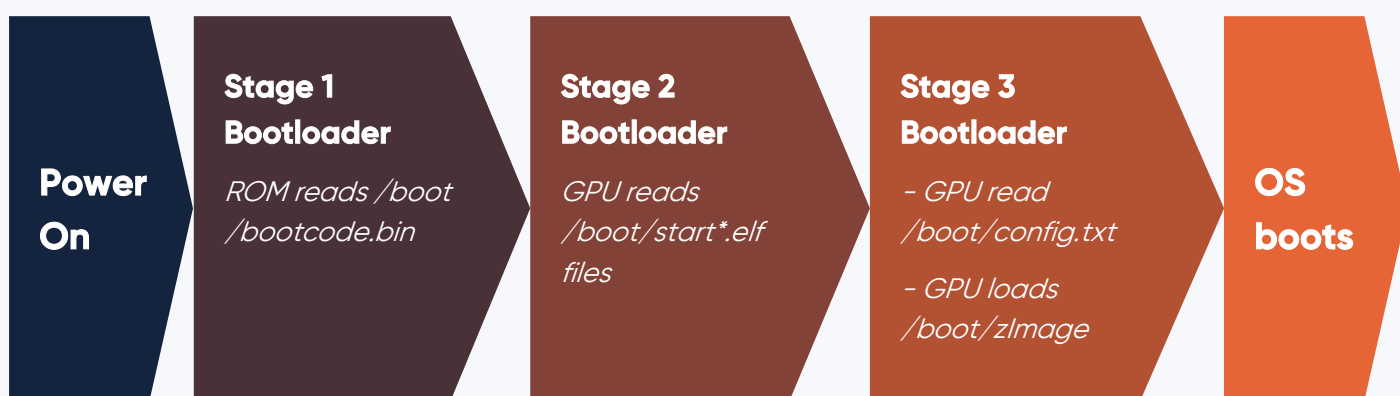
Stage 2

The second stage bootloader, bootcode.bin, executes on the VideoCore GPU and loads the third stage bootloader, **start.elf**.

Stage 3

The third stage bootloader, **start.elf**, is where all the action happens. **It starts by reading config.txt**, a text file containing configuration parameters for both the VideoCore (Video/HDMI modes, memory, console frame buffers, etc.) and **loading of the Linux kernel** (load addresses, device tree, uart/console baud rates, etc.). **start.elf** can load any file named **zImage** (the compiled kernel image).

In short, the standard (non-U-Boot) boot process is as follows:

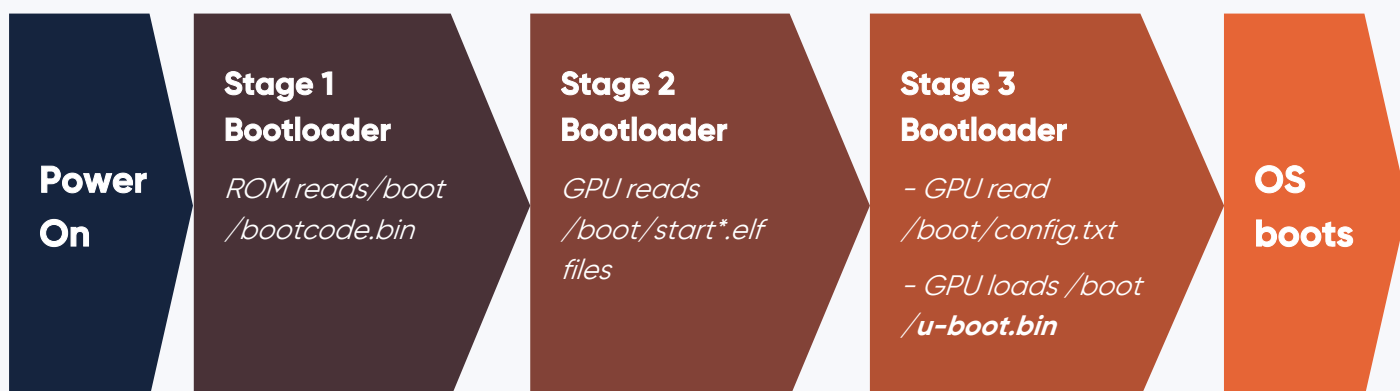


Note that on the RPI, the GPU initializes the system and performs the boot's initial stages.

RPI Bootloading Process with U-Boot

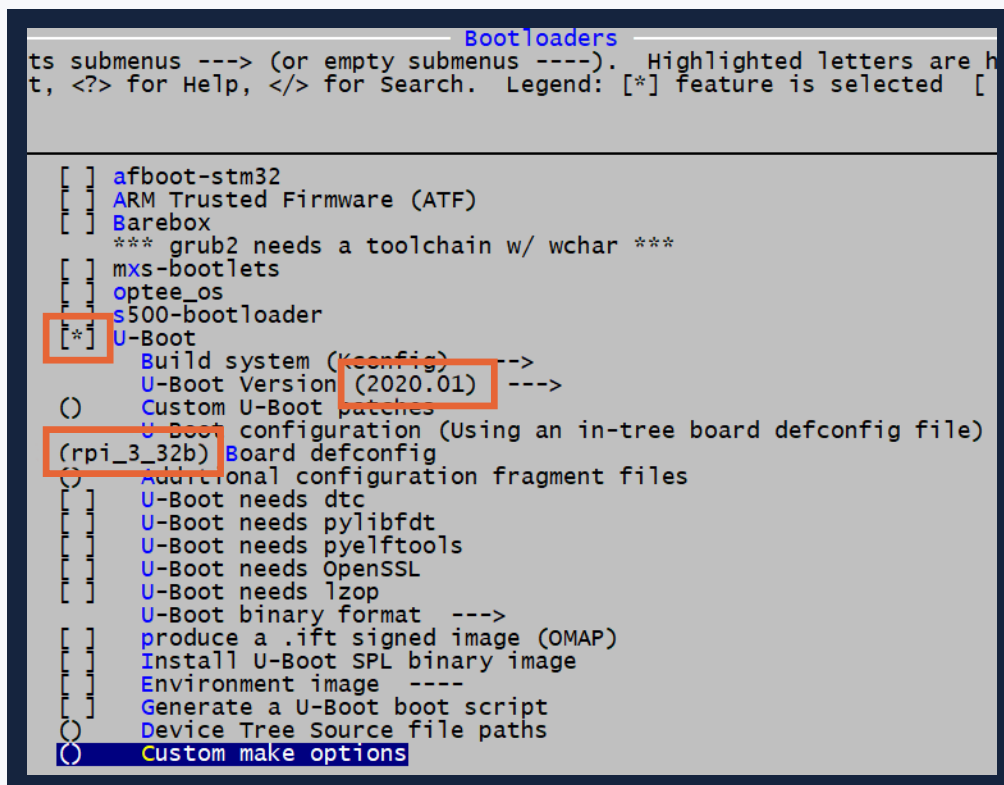
To trick the RPI bootloader to load U-Boot, set ***start.elf*** into loading the **U-Boot image** instead; use the compiled ***u-boot.bin*** to pass as a kernel image. This is the first thing that runs on the ARM processor.

The boot process with U-Boot enabled is as follows:



Compile U-Boot Using Buildroot

Yes, you can download the U-Boot repository and compile it yourself manually, but for that, you have Buildroot to automate it and make it part of your build process. So, configure Buildroot and enter the ***Bootloaders*** menu:



```
ts submenus ---> (or empty submenus ----). Highlighted letters are h
t, <?> for Help, </> for Search. Legend: [*] feature is selected [

[ ] afboot-stm32
[ ] ARM Trusted Firmware (ATF)
[ ] Barebox
    *** grub2 needs a toolchain w/ wchar ***
[ ] mxs-bootlets
[ ] optee_os
[ ] S500-bootloader
[*] U-Boot
    Build system (Kconfig) --->
    U-Boot Version (2020.01) --->
    Custom U-Boot patches
    U-Boot configuration (Using an in-tree board defconfig file)
    (rpi_3_32b) Board defconfig
    Additional configuration fragment files
    U-Boot needs dtc
    U-Boot needs pylibfdt
    U-Boot needs pyelftools
    U-Boot needs OpenSSL
    U-Boot needs lzop
    U-Boot binary format --->
    produce a .ift signed image (OMAP)
    Install U-Boot SPL binary image
    Environment image ----
    Generate a U-Boot boot script
    Device Tree Source file paths
    Custom make options
```

And run **make**.

At the end of the compilation you should see the U-Boot binary under the **output/images** directory:

```
-rw-r--r-- 1 sh sh 467K Sep 23 23:12 u-boot.bin
```

Preparing Files

1. Prepare config.txt:

Change it from:

```
kernel=zImage
```

to:

```
kernel=u-boot.bin
```

- Don't forget to add the **enable_uart=1** line to the **config.txt** file, as well.

2. Prepare the boot commands for U-Boot:

When U-Boot starts, it needs certain configuration to know what to boot next and how, instead of typing it manually each time, prepare it so U-Boot will read this configuration when it starts:

- a. Create a new ***boot_commands.txt*** file and enter:

```
mmc dev 0
fatload mmc 0:1 ${kernel_addr_r} zImage
fatload mmc 0:1 ${fdt_addr_r} bcm2710-rpi-3-b.dtb
setenv bootargs root=/dev/mmcblk0p2 rootfstype=ext4 console=tty1
console=ttyAMA0,115200 earlyprintk rootwait noinitrd
fdt addr ${fdt_addr} && fdt get value bootargs /chosen bootargs
bootz ${kernel_addr_r} - ${fdt_addr}
```

- b. Compile the commands to the U-Boot format:

```
mkimage -A arm -T script -C none -n "Boot script" -d boot_commands.txt boot.scr.uimg
```

- c. Copy the newly created ***boot.scr.uimg*** file into the ***/boot*** partition on your SD card.

3. Copy u-boot.bin to the ***/boot*** partition

Mazel Tov! You're Now Using U-Boot Bootloader

Save all changes. if you're running on Linux, then type in the **sync** command to make sure all changes are flushed.

Eject the SD card safely, put it into the Raspberry Pi, make sure that UART cable plugs in, and that using **screen/putty** you've connected to your **COM** or **/dev/ttyS*** port.

Turn on the board, and you should receive an output similar to the following:


```

COM4 - PuTTY

U-Boot 2020.01 (Sep 21 2020 - 19:18:44 +0300)

DRAM:  924 MiB
RPI 3 Model B (0xa02082)
MMC:   mmc@7e202000: 0, sdhci@7e300000: 1
Loading Environment from FAT... *** Warning - bad CRC, using default environment

In:     serial
Out:    vidconsole
Err:    vidconsole
Net:    No ethernet found.
starting USB...
Bus usb@7e980000: scanning bus usb@7e980000 for devices... 3 USB Device(s) found
           scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot:  0
switch to partitions #0, OK
mmc0 is current device
Scanning mmc 0:1...
Found U-Boot script /boot.scr.uimg
392 bytes read in 2 ms (191.4 KiB/s)
## Executing script at 02400000
switch to partitions #0, OK
mmc0 is current device
5248536 bytes read in 219 ms (22.9 MiB/s)
25469 bytes read in 3 ms (8.1 MiB/s)
Kernel image @ 0x080000 [ 0x000000 - 0x501618 ]
## Flattened Device Tree blob at 2eff9800
   Booting using the fdt blob at 0x2eff9800
   Using Device Tree in place at 2eff9800, end 2f002f6b

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.19.23-v7 (sh@LAPTOP-G8FAAP6L) (gcc version 8.4.0 (B
[    0.000000] CPU: ARMv7 Processor [410fd034] revision 4 (ARMv7), cr=10c5383d
[    0.000000] CPU: div instructions available: patching division code
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction
[    0.000000] OF: fdt: Machine model: Raspberry Pi 3 Model B Rev 1.2
[    0.000000] Memory policy: Data cache writealloc
[    0.000000] cma: Reserved 8 MiB at 0x39400000
[    0.000000] random: get_random_bytes called from start_kernel+0xb0/0x4b4 with c
[    0.000000] percpu: Embedded 17 pages/cpu @(ptrval) s39436 r8192 d22004 u69632
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 234465
[    0.000000] Kernel command line: 8250.nr_uarts=1 bcm2708_fb.fbwidth=720 bcm2708
console=ttyL console=ttyS0,115200

```

Bonus: Static Compilation of Packages

Forgot to add a specific package and don't want the hassle of recompiling the entire firmware image just for that? You may use static compilation to get your software running.

Compiling Netcat Statically

Take *netcat* as an example, first [download the source code](https://netix.dl.sourceforge.net/project/netcat/netcat/0.7.1/netcat-0.7.1.tar.gz):

```
wget https://netix.dl.sourceforge.net/project/netcat/netcat/0.7.1/netcat-0.7.1.tar.gz

tar -xzvf netcat-0.7.1.tar.gz
cd netcat-0.7.1
```

Configure for static compilation:

```
./configure --build x86_64-pc-linux-gnu --host arm-linux-gnueabi LDFLAGS="-static -fPIC"
```

Compile:

```
make
```

Congrats! now you may copy your compiled file to the IoT device and run it.

```
sh@LAPTOP-: ~/netcat-0.7.1/src$ file netcat
netcat: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked,
BuildID[sha1]=e58628ec60b804d32b0daaaefa6f7ce596b587dd, for GNU/Linux 3.2.0, with
debug_info, not stripped
```

If you would like to reduce its size, you may strip it from its symbols using *arm-linux-gnueabi-strip*:

```
sh@LAPTOP-: ~/netcat-0.7.1/src$ ls -al netcat
-rwxrwxr-x 1 sh sh 740452 Sep 25 13:14 netcat
sh@LAPTOP-: ~/netcat-0.7.1/src$ arm-linux-gnueabi-strip -s netcat
sh@LAPTOP-: ~/netcat-0.7.1/src$ ls -al netcat
-rwxrwxr-x 1 sh sh 529580 Sep 25 13:16 netcat
```



SUMMARY

This post has walked you through the steps of using a serial connection, creating your own Linux IoT embedded image, from the bootloader and all the way to the kernel and installed packages and drivers using an automated build system.

There's a lot more to learn about creating your own firmware image and using Buildroot and its neat features. I hope I've provided you with the basics you need to start.

Good luck!



Securing the Connected Future

For on going support and question about Firedome contact:

www.firedome.io | support@firedome.io | +1 (374) 826-6713 | Copyright © 2021 FIREDOME