Building Raspberry Pi Systems with Yocto

28 Jun 2019

This post is about building Linux systems for Raspberry Pi boards using software from the Yocto Project.

Yocto is a set of tools for building a custom embedded Linux distribution. The systems are usually targeted for a particular application like a commercial product.

If you are looking to build a general purpose development system with access to pre-built packages, I suggest you stick with a more user-friendly distribution like Raspbian.

And while the Yocto system is very powerful, it does have a substantial learning curve. You may want to look at another popular, but simpler tool for building embedded systems Buildroot.

Yocto uses what it calls **meta-layers** to define the configuration. Within each meta-layer are recipes, classes and configuration files that support the primary build tool, a python app called **bitbake**.

I have created a custom meta-layer for the RPi boards called meta-rpi.

The systems built from this layer use the same GPU firmware, linux kernel and include the same dtb overlays as the official Raspbian systems. This means that no hardware functionality is lost with these Yocto built systems as compared to the “official” Raspbian distro. It is only the userland software that differs and that is completely configurable by you.

There are a some example images in meta-rpi that support the programming languages and tools that I commonly use in my own projects.

When using this repository for customer projects, I first fork and move it to another repository, usually with a different name. I recommend you do the same if you require stability. I use the meta-rpi layer for my experiments.

My systems use **sysvinit**, but Yocto supports **systemd**.

If you are Qt5 developer then you will appreciate that the RPi comes with working OpenGL drivers for the RPi GPU. This means Qt OpenGL and QML applications will work when using the eglfs platform plugin.

I am using the official Yocto meta-raspberrypi layer, but have updated recipes for the Linux kernel and gpu firmware to keep them more current. I also have occasional ‘fixes’ to other components, sometimes for bugs, but often just because I don’t like the **meta-raspberrypi** defaults.

I have access to all of the RPi boards and have at one time or another tested these builds with all of them including the RPi CM and CM3 modules.

Most of the time I test only with RPi3 and RPi0-W boards.

Downloads

If you want a quick look at the resulting systems, you can download some pre-built images here.

Instructions for installing onto an SD card are in the README.

The login user is **root** with password **jumpnowtek**.

You will be prompted to change the password on first login.

All systems are setup to use a serial console. For the RPi’s that have it, a dhcp client will run on the ethernet interface and there is an ssh server running.

Note: There is a custom firewall rule that will lock your IP out for 1 minute if you fail to login with ssh after 3 attempts.

System Info

The Yocto version is **2.7.0**, the `[warrior]` branch.

The **4.19** Linux kernel comes from the github.com/raspberrypi/linux repository.

These are **sysvinit** systems using eudev.

The Qt version is **5.12.1** There is no **X11** and no desktop installed. Qt GUI applications can be run fullscreen using one of the Qt embedded linux plugins like **eglfs** or **linuxfb**, both are provided. The default is **eglfs**.

Python **3.7.2** with a number of modules is included.

gcc/g++ **8.3.0** and associated build tools are installed.

git **2.20.1** is installed.

wireguard **20190702** is installed.

omxplayer is installed for playing video and audio from the command line, hardware accelerated.

Raspicam the command line tool for using the Raspberry Pi camera module is installed.

There is an example image that I use for a couple of Raspberry Pi music systems. They use either an IQaudIO Pi-DigiAMP+ or HiFiBerry Amp+ add-on board and pianobar, a console-based client for Pandora internet radio.

The Adafruit PiTFT 3.5" and PiTFT 2.8" resistive touchscreens work. Support for some other TFT displays is included, but I haven't tested them.

Raspi2fb is included for mirroring the GPU framebuffer to the small TFT displays. This allows for running Qt GUI applications on the TFTs.

Ubuntu Setup

I primarily use **16.04** 64-bit servers for builds. Other versions should work.

You will need at least the following packages installed

```
build-essential
chrpath
diffstat
gawk
libncurses5-dev
texinfo
```

For **16.04** you also need to install the **python 2.7** package

```
python2.7
```

And then create some links for it in `/usr/bin`

```
sudo ln -sf /usr/bin/python2.7 /usr/bin/python
sudo ln -sf /usr/bin/python2.7 /usr/bin/python2
```

For all versions of Ubuntu, you should change the default Ubuntu shell from **dash** to **bash** by running this command from a shell

```
sudo dpkg-reconfigure dash
```

Choose **No** to dash when prompted.

Fedora Setup

I have used a **Fedora 27** 64-bit workstation.

The extra packages I needed to install for Yocto were

```
chrpath
perl-bignum
perl-Thread-Queue
texinfo
```

and the package group

```
Development Tools
```

Fedora already uses **bash** as the shell.

Clone the dependency repositories

For all upstream repositories, use the `[warrior]` branch.

The directory layout I am describing here is my preference. All of the paths to the meta-layers are configurable. If you choose something different, adjust the following instructions accordingly.

First the main Yocto project **poky** layer

```
~# git clone -b warrior git://git.yoctoproject.org/poky.git poky-warrior
```

Then the dependency layers under that

```
~$ cd poky-warrior
~/poky-warrior$ git clone -b warrior git://git.openembedded.org/meta-openembedded
~/poky-warrior$ git clone -b warrior https://github.com/meta-qt5/meta-qt5
~/poky-warrior$ git clone -b warrior git://git.yoctoproject.org/meta-raspberrypi
```

These repositories shouldn't need modifications other then periodic updates and can be reused for different projects or different boards.

Clone the meta-rpi repository

Create a separate sub-directory for the **meta-rpi** repository before cloning. This is where you will be doing your customization.

```
~$ mkdir ~/rpi
~$ cd ~/rpi
~/rpi$ git clone -b warrior git://github.com/jumpnow/meta-rpi
```

The `meta-rpi/README.md` file has the last commits from the dependency repositories that I tested. You can always checkout those commits explicitly if you run into problems.

Initialize the build directory

Again much of the following are only my conventions.

Choose a build directory. I tend to do this on a per board and/or per project basis so I can quickly switch between projects. For this example I'll put the build directory under `~/rpi/` with the `meta-rpi` layer.

You could manually create the directory structure like this

```
$ mkdir -p ~/rpi/build/conf
```

Or you could use the Yocto environment script **oe-init-build-env** like this passing in the path to the build directory

```
~$ source poky-warrior/oe-init-build-env ~/rpi/build
```

The Yocto environment script will create the build directory if it does not already exist.

Customize the configuration files

There are some sample configuration files in the **meta-rpi/conf** directory.

Copy them to the **build/conf** directory (removing the '-sample')

```
~/rpi$ cp meta-rpi/conf/local.conf.sample build/conf/local.conf
~/rpi$ cp meta-rpi/conf/bblayers.conf.sample build/conf/bblayers.conf
```

If you used the **oe-init-build-env** script to create the build directory, it generated some generic configuration files in the **build/conf** directory. If you want to look at them, save them with a different name before overwriting.

It is not necessary, but you may want to customize the configuration files before your first build.

Warning: Do not use the '**~**' character when defining directory paths in the Yocto configuration files.

Edit bblayers.conf

In **bblayers.conf** file replace **${HOME}** with the appropriate path to the meta-layer repositories on your system if you modified any of the paths in the previous instructions.

WARNING: Do not include **meta-yocto-bsp** in your **bblayers.conf**. The Yocto BSP requirements for the Raspberry Pi are in **meta-raspberrypi**.

For example, if your directory structure does not look exactly like this, you will need to modify `bblayers.conf`

```
~/poky-warrior/
     meta-openembedded/
     meta-qt5/
     meta-raspberrypi
     ...

~/rpi/
    meta-rpi/
    build/
        conf/
```

Edit local.conf

The variables you may want to customize are the following:

- MACHINE
- TMPDIR
- DL_DIR
- SSTATE_DIR

The defaults for all of these work fine with the exception of **MACHINE**.

MACHINE

The **MACHINE** variable is used to determine the target architecture and various compiler tuning flags.

See the conf files under `meta-raspberrypi/conf/machine` for details.

The choices for **MACHINE** are

- raspberrypi (BCM2835)
- raspberrypi0 (BCM2835)
- raspberrypi0-wifi (BCM2835)
- raspberrypi2 (BCM2836 or BCM2837 v1.2+)
- raspberrypi3 (BCM2837)

- raspberrypi-cm (BCM2835)
- raspberrypi-cm3 (BCM2837)

You can only build for one type of **MACHINE** at a time.

There are really just two *tuning* families using the default Yocto configuration files

- arm1176jzfshf - for the the BCM2835 boards
- cortexa7thf-neon-vfpv4 - for the BCM2836 and BCM2837 boards

Boards in the same family can generally run the same software.

One exception is **u-boot**, which is NOT the default for the systems being built here.

One of the reasons you would want to use **u-boot** with the RPis is to work with the Mender upgrade system.

TMPDIR

This is where temporary build files and the final build binaries will end up. Expect to use at least **50GB**.

The default location is under the **build** directory, in this example **~/rpi/build/tmp**.

If you specify an alternate location as I do in the example conf file make sure the directory is writable by the user running the build.

DL_DIR

This is where the downloaded source files will be stored. You can share this among configurations and builds so I always create a general location for this outside the project directory. Make sure the build user has write permission to the directory you decide on.

The default location is in the **build** directory, **~/rpi/build/sources**.

SSTATE_DIR

This is another Yocto build directory that can get pretty big, greater then **8GB**. I often put this somewhere else other then my home directory as well.

The default location is in the **build** directory, **~/rpi/build/sstate-cache**.

ROOT PASSWORD

There is only one login user by default, **root**.

The default password is set to **jumpnowtek** by these two lines in the **local.conf** file

```
INHERIT += "extrausers"
EXTRA_USERS_PARAMS = "usermod -P jumpnowtek root; "
```

These two lines force a password change on first login

```
INHERIT += "chageusers"
CHAGE_USERS_PARAMS = "chage -d0 root; "
```

You can comment them out if you do not want that behavior.

If you want no password at all (development only hopefully), comment those four lines and uncomment this line

```
EXTRA_IMAGE_FEATURES = "debug-tweaks"

#INHERIT += "extrausers"
#EXTRA_USERS_PARAMS = "usermod -P jumpnowtek root; "
```

```
#INHERIT += "chageusers"
#CHAGE_USERS_PARAMS = "chage -d0 root; "
```

You can always add or change the password once logged in.

Run the build

You need to source the Yocto environment into your shell before you can use bitbake. The **oe-init-build-env** will not overwrite your customized conf files.

```
~$ source poky-warrior/oe-init-build-env ~/rpi/build

### Shell environment set up for builds. ###

You can now run 'bitbake '

Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-toolchain-sdk
    adt-installer
    meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'
~/rpi/build$
```

I don't use any of those *Common targets*, but instead always write my own custom image recipes.

The **meta-rpi** layer has some examples under **meta-rpi/images/**.

Build

To build the **console-image** run the following command

```
~/rpi/build$ bitbake console-image
```

You may occasionally run into build errors related to packages that either failed to download or sometimes out of order builds. The easy solution is to clean the failed package and rerun the build again.

For instance if the build for **zip** failed for some reason, I would run this

```
~/rpi/build$ bitbake -c cleansstate zip
~/rpi/build$ bitbake zip
```

And then continue with the full build.

```
~/rpi/build$ bitbake console-image
```

To build the `qt5-image` it would be

```
~/rpi/build$ bitbake qt5-image
```

The **cleansstate** command (with two s's) works for image recipes as well.

The image files won't get deleted from the **TMPDIR** until the next time you build.

Copying the binaries to an SD card (or eMMC)

After the build completes, the bootloader, kernel and rootfs image files can be found in **/deploy/images/$MACHINE** with **MACHINE** coming from your **local.conf**.

The **meta-rpi/scripts** directory has some helper scripts to format and copy the files to a microSD card.

See this post for an additional first step required for the RPi Compute eMMC.

mk2parts.sh

This script will partition an SD card with the minimal 2 partitions required for the RPI.

Insert the microSD into your workstation and note where it shows up.

lsblk is convenient for finding the microSD card.

For example

```
~/rpi/meta-rpi$ lsblk
NAME    MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda       8:0    0 931.5G  0 disk
|-sda1    8:1    0  93.1G  0 part /
|-sda2    8:2    0  93.1G  0 part /home
|-sda3    8:3    0  29.8G  0 part [SWAP]
|-sda4    8:4    0     1K  0 part
|-sda5    8:5    0   100G  0 part /oe5
|-sda6    8:6    0   100G  0 part /oe6
|-sda7    8:7    0   100G  0 part /oe7
|-sda8    8:8    0   100G  0 part /oe8
|-sda9    8:9    0   100G  0 part /oe9
`-sda10   8:10   0 215.5G  0 part /oe10
sdb       8:16   1   7.4G  0 disk
|-sdb1    8:17   1    64M  0 part
`-sdb2    8:18   1   7.3G  0 part
```

So I will use **sdb** for the card on this machine.

It doesn't matter if some partitions from the SD card are mounted. The **mk2parts.sh** script will unmount them.

WARNING: This script will format any disk on your workstation so make sure you choose the SD card.

```
~$ cd ~/rpi/meta-rpi/scripts
~/rpi/meta-rpi/scripts$ sudo ./mk2parts.sh sdb
```

You only have to format the SD card once.

Temporary mount point

You will need to create a mount point on your workstation for the copy scripts to use.

This is the default

```
~$ sudo mkdir /media/card
```

You only have to create this directory once.

If you don't want that location, you will have to edit the following scripts to use the mount point you choose.

copy_boot.sh

This script copies the GPU firmware, the Linux kernel, dtbs and overlays, config.txt and cmdline.txt to the boot partition of the SD card.

This **copy_boot.sh** script needs to know the **TMPDIR** to find the binaries. It looks for an environment variable called **OETMP**.

For instance, if I had this in `build/conf/local.conf`

```
TMPDIR = "/oe4/rpi/tmp-warrior"
```

Then I would export this environment variable before running `copy_boot.sh`

```
~/rpi/meta-rpi/scripts$ export OETMP=/oe4/rpi/tmp-warrior
```

If you didn't override the default **TMPDIR** in `local.conf`, then set it to the default **TMPDIR**

```
~/rpi/meta-rpi/scripts$ export OETMP=~/rpi/build/tmp
```

The `copy_boot.sh` script also needs a **MACHINE** environment variable specifying the type of RPi board.

```
~/rpi/meta-rpi/scripts$ export MACHINE=raspberrypi3
```

or

```
~/rpi/meta-rpi/scripts$ export MACHINE=raspberrypi0-wifi
```

Then run the **copy_boot.sh** script passing the location of SD card

```
~/rpi/meta-rpi/scripts$ ./copy_boot.sh sdb
```

This script should run very fast.

If you want to customize the **config.txt** or **cmdline.txt** files for the system, you can place either of those files in the **meta-rpi/scripts** directory and the **copy_boot.sh** script will copy them as well.

Take a look at the script if this is unclear.

copy_rootfs.sh

This script copies the root file system to the second partition of the SD card.

The **copy_rootfs.sh** script needs the same **OETMP** and **MACHINE** environment variables.

The script accepts an optional command line argument for the image type, for example **console** or **qt5**. The default is **console** if no argument is provided.

The script also accepts a **hostname** argument if you want the host name to be something other then the default **MACHINE**.

Here's an example of how you would run **copy_rootfs.sh**

```
~/rpi/meta-rpi/scripts$ ./copy_rootfs.sh sdb console
```

or

```
~/rpi/meta-rpi/scripts$ ./copy_rootfs.sh sdb qt5 rpi3
```

The **copy_rootfs.sh** script will take longer to run and depends a lot on the quality of your SD card. With a good **Class 10** card it should take less then 30 seconds.

The copy scripts will **NOT** unmount partitions automatically. If an SD card partition is already mounted, the script will complain and abort. This is for safety, mine mostly, since I run these scripts many times a day on different machines and the SD cards show up in different places.

Here's a realistic example session where I want to copy already built images to a second SD card that I just inserted.

```
~$ sudo umount /dev/sdb1
~$ sudo umount /dev/sdb2
~$ export OETMP=/oe4/rpi/tmp-warrior
~$ export MACHINE=raspberrypi2
~$ cd rpi/meta-rpi/scripts
```

```
~/rpi/meta-rpi/scripts$ ./copy_boot.sh sdb
~/rpi/meta-rpi/scripts$ ./copy_rootfs.sh sdb console rpi3
```

Once past the development stage I usually wrap all of the above in another script for convenience.

Both **copy_boot.sh** and **copy_rootfs.sh** are simple scripts, easily customized.

Some custom package examples

spiloop is a **spidev** test application.

The **bitbake recipe** that builds and packages **spiloop** is here

```
meta-rpi/recipes-misc/spiloop/spiloop_git.bb
```

Use it to test the **spidev** driver before and after placing a jumper between pins **19** and **21**.

tspress is a Qt5 GUI application installed with the `qt5-image`. I use it for testing touchscreens.

The recipe is here and can be used a guide for your own applications.

```
meta-rpi/recipes-qt/tspress/tspress_git.bb
```

Check the **README** in the tspress repository for usage.

Adding additional packages

To display the list of available recipes from the **meta-layers** included in **bblayers.conf**

```
~$ source poky-warrior/oe-init-build-env ~/rpi/build

~/rpi/build$ bitbake -s
```

Once you have the recipe name, you need to find what packages the recipe produces. Use the **oe-pkgdata-util** utility for this.

For instance, to see the packages produced by the **openssh** recipe

```
~/rpi/build$ oe-pkgdata-util list-pkgs -p openssh
openssh-keygen
openssh-scp
openssh-ssh
openssh-sshd
openssh-sftp
openssh-misc
openssh-sftp-server
openssh-dbg
openssh-dev
openssh-doc
openssh
```

These are the individual packages you could add to your image recipe.

You can also use **oe-pkgdata-util** to check the individual files a package will install.

For instance, to see the files for the **openssh-sshd** package

```
~/rpi/build$ oe-pkgdata-util list-pkg-files openssh-sshd
openssh-sshd:
        /etc/default/volatiles/99_sshd
        /etc/init.d/sshd
        /etc/ssh/moduli
        /etc/ssh/sshd_config
        /etc/ssh/sshd_config_readonly
```

```
        /usr/libexec/openssh/sshd_check_keys
        /usr/sbin/sshd
```

For a package to be installed in your image it has to get into the **IMAGE_INSTALL** variable some way or another. See the example image recipes for some common conventions.

Playing videos

The RPi project has a hardware-accelerated, command-line video player called omxplayer.

Here's a reasonably sized example from the Blender project to test

```
root@rpi3:~# wget https://download.blender.org/demo/movies/Cycles_Demoreel_2015.mov
```

You can play it like this (**-o hdmi** for hdmi audio)

```
root@rpi3:~# omxplayer -o hdmi Cycles_Demoreel_2015.mov
Video codec omx-h264 width 1920 height 1080 profile 77 fps 25.000000
Audio codec aac channels 2 samplerate 48000 bitspersample 16
Subtitle count: 0, state: off, index: 1, delay: 0
V:PortSettingsChanged: 1920x1080@25.00 interlace:0 deinterlace:0 anaglyph:0 par:1.25 display:0 layer:0 alpha:25
```

If you get errors like this

```
COMXAudio::Decode timeout
```

Increase memory allocated to the GPU in `config.txt`

```
gpu_mem=128
```

The RPi GPU can support more then one display, (the DSI display is the default), though apps have to be built specifically to support the second display. Omxplayer is an app with this ability.

So for example, with the RPi DSI touchscreen and an HDMI display attached at the same time, you could run a video on the HDMI display from the touchscreen this way

```
root@rpi3:~# omxplayer --display=5 -o hdmi Cycles_Demoreel_2015.mov
Video codec omx-h264 width 1920 height 1080 profile 77 fps 25.000000
Audio codec aac channels 2 samplerate 48000 bitspersample 16
Subtitle count: 0, state: off, index: 1, delay: 0
V:PortSettingsChanged: 1920x1080@25.00 interlace:0 deinterlace:0 anaglyph:0 par:1.25 display:5 layer:0 alpha:25
```

I was not able to run a **eglfs** Qt app on the RPi DSI display while simultaneously playing a movie with omxplayer on the HDMI display. Perhaps a **linuxfb** Qt app that doesn't use the GPU could run simultaneously. Some more testing is needed.

Using the Raspberry Pi Camera

The raspicam command line tools are installed with the **console-image** or any image that includes the **console-image**

- raspistill
- raspivid
- raspiyuv

To enable the RPi camera, add or edit the following in the RPi configuration file **config.txt**

```
start_x=1
gpu_mem=128
disable_camera_led=1   # optional for disabling the red LED on the camera
```

To get access to **config.txt**, mount the boot partition first

```
root@rpi# mkdir /mnt/fat
root@rpi# mount /dev/mmcblk0p1 /mnt/fat
```

Then edit, save and reboot.

```
root@rpi# vi /mnt/fat/config.txt
```

or

```
root@rpi# nano /mnt/fat/config.txt
```

A quick test of the camera, flipping the image because of the way I have my camera mounted and a timeout of zero so it runs until stopped.

```
root@rpi2# raspistill -t 0 -hf -vf
```