Embedded Linux System using Yocto Project and Docker

Devendra Tewari

August 9, 2021

Embedded Linux System with Yocto Project and Docker

Embedded Linux System with Yocto Project and Docker

This project provides a template to build an embedded Linux system for the Raspberry Pi using Yocto Project and Docker.

Hardware and Software Requirements

- A host machine running Linux, macOS, or Windows, with Git and Docker
  - 16 GB RAM - 8 GB assigned to Docker, on macOS or Windows
- A target machine
  - Raspberry Pi Zero W (default target)
  - 8 GB Class 10 Micro SD Card
  - Power adapter
  - Headless (optional)
    - USB to TTL Serial 3.3V Adapter Cable
  - HDMI Display (optional)
    - HDMI cable and a mini or micro HDMI adapter
    - Micro USB male to USB female adapter
    - USB mouse and keyboard (wireless works)

Background

This topic gives you some background knowledge required to effectively use this project.

Docker

Docker enables you to build this project exactly as intended. You can download and install Docker Desktop on Linux, macOS, and Windows. You'll need to have some familiarity with `docker` commands such as cp, build, buildx, exec, image, ps, rm, run, and start.

You'll also need to understand Dockerfile syntax to tweak the build's Dockerfile.

Yocto Project

- Is a Linux Foundation Collaborative Project
- Uses a declarative, layered, build configuration that leverages BitBake
- Downloads software from Git and other sources
- Builds cross-compile toolchain, board support package (BSP), and Linux kernel image
- Builds software by automatically invoking make, autotools, or cmake
- Builds software as packages and installs to generate file system image
- Maintains system state (sstate) cache to speed up incremental builds
- Builds SDK or eSDK for application development

Layers and Recipes

The embedded Linux system is built from recipes available in the following layers. A recipe typically builds one software package for the target machine, and its native, native SDK, debug, development, source, and documentation packages.

- poky - core Yocto Project container layer that provides
  - meta - openembedded-core distro-less layer
  - meta-poky - contains recipes for the `poky` distro
  - meta-yocto-bsp - core BSP and Linux kernel recipes
- meta-raspberrypi - BSP layer that extends poky to build the Raspberry Pi Linux kernel
- meta-openembedded - container layer that provides
  - meta-oe - provides hostapd
  - meta-networking - provides dnsmasq
  - meta-python
- meta-clang - builds the system with Clang - a LLVM based front-end and back-end for compiling C/C++ that is unencumbered by GPL.

Configuration files and kas

Yocto Project provides no means to download layers, and setup configuration files, for different builds.

kas is a build tool for Yocto Project that

- Is configured through YAML or JSON e.g. kas.yml
- Downloads layers - checks out to a specified version, and applies patches
- Generates build directory with
  - `conf/bblayers.conf` - list of layers to build
  - `conf/local.conf` - MACHINE and DISTRO configuration
- Performs multiconfig builds specified via `KAS_TARGET` environment variable

Learning Resources

- Yocto Project Presentation Videos
- Alessandro Flaminio's Master Thesis
- Yocto Project Mega Manual
- BitBake User Manual

Build instructions

This section discusses how you can perform a build and save its history in a Docker image.

Prepare a Docker container for build

Create a Docker container

```
docker run -it --name berrydev crops/yocto:ubuntu-20.04-base
```

Open a privileged shell to install kas

```
docker exec -it -u 0 berrydev /bin/bash
```

Install kas

```
apt install python3-pip
pip3 install kas
```

Build on a Linux host or Docker container

```
git clone https://github.com/tewarid/berry
cd berry
kas build kas.yml
```

Build using Docker Build

Clone the project repo on host and run

```
git clone https://github.com/tewarid/berry
cd berry
docker build -t berry:latest .
```

Pick a different Raspberry Pi

By default, the image is built for Raspberry Pi Zero Wi-Fi. Edit `machine` in `kas.yml` to build for a different model

MACHINE	Model
raspberrypi-cm	Raspberry Pi Compute Module
raspberrypi-cm3	Raspberry Pi 3 Compute Module
raspberrypi	Raspberry Pi Model B+
raspberrypi0-wifi	Raspberry Pi Zero with Wi-Fi
raspberrypi0	Raspberry Pi Zero
raspberrypi2	Raspberry Pi 2
raspberrypi3-64	Raspberry Pi 3 64-bit build
raspberrypi3	Raspberry Pi 3 32-bit build
raspberrypi4-64	Raspberry Pi 4 64-bit build
raspberrypi4	Raspberry Pi 4 32-bit build

Multiconfig build

A multiconfig build can generate images for multiple targets in one build

```
export KAS_TARGET="mc:berry-raspberrypi0-wifi:core-image-base mc:berry-ras
./scripts/build.sh
```

Note that multiconfig builds may fail due to memory exhaustion when multiple memory intensive tasks such as LLVM build are run at the same time. The builds also require more storage due to usage of different `TMPDIR` for each target. Specify fewer targets in `KAS_TARGET` if your build environment is low on resources.

Additional targets can be created under `layers/meta-berry/conf/multiconfig`.

Access private Git repos in build

If you have encrypted your home folder using ecryptfs, you cannot use the procedure below.

Run ssh-agent on host and add default `~/.ssh/id_rsa` key

```
export SSH_AUTH_SOCK=~/.ssh/ssh-auth.sock
ssh-agent -a $SSH_AUTH_SOCK
ssh-add ~/.ssh/id_rsa
ssh-add -l
```

Edit Dockerfile and add `--mount=type=ssh,mode=777` to build

```
RUN --mount=type=ssh,mode=777 kas build kas.yml
```

Build with BuildKit or `docker buildx`

```
export DOCKER_BUILDKIT=1
docker build \
  --ssh default=$SSH_AUTH_SOCK \
  -t berry:latest .
```

Incremental development

This section shows how you can create a container from a Docker image, to do additional development, and perform incremental builds.

Create a container

Create a container called `berrydev` for incremental development

```
docker run --name berrydev -it berry:latest
```

Start a stopped container

```
docker start -ai berrydev
```

See whether the container is running or stopped

```
docker ps -a
```

Run incremental build

Make the necessary changes to source code and rebuild

`kas build kas.yml`

Note that BitBake may fail with Invalid cross-device link error. Follow the link for additional information and a patch.

Access private Git repos in container

If you have encrypted your home folder using ecryptfs, you cannot use the procedure below.

Create Docker container with access to ssh-agent on host

```
export SSH_AUTH_SOCK=~/.ssh/ssh-auth.sock
docker run --name berrydev -it \
  -v $SSH_AUTH_SOCK:/run/host-services/ssh-auth.sock \
  -e SSH_AUTH_SOCK="/run/host-services/ssh-auth.sock" \
  berry:latest
```

Enable non-root access to ssh-agent

Since we're using a non-root user, you may get an access denied message when you run

```
# Access your Git server instead of example.com
ssh git@example.com
```

If so, you will need to fix access to ssh-agent socket at least once

```
docker exec -u 0 -it berrydev /bin/bash
chmod 777 /run/host-services/ssh-auth.sock
```

Download cache

A download cache can be setup under `build/downloads`. It will be copied into the image along with the source code. This can reduce build times significantly.

To copy download folder from a container to the host

```
docker cp \
  berrydev:/home/yoctouser/berry/build/downloads \
  build/
```

Working with BitBake

Setup build environment in a container to gain access to BitBake

```
source layers/poky/oe-init-build-env
# this will leave you in the build directory
```

Run incremental build using BitBake in the build directory

```
bitbake core-image-base
```

Clean a recipe

You can clean state of any recipe to build it from scratch

`bitbake recipe_name -c cleansstate`

`cleanall` also cleans the download cache for the recipe.

Run devshell

devshell enables you to work in a recipe's build environment

```
bitbake recipe_name -c devshell
```

Use `exit` to close devshell.

Generate SDK or eSDK

```
bitbake core-image-base -c populate_sdk
# OR
bitbake core-image-base -c populate_sdk_ext
```

SDK should be located at `tmp/deploy/sdk` under build directory.

Working with the kernel

To clean kernel build

```
bitbake virtual/kernel -c clean
```

To change kernel config and produce a diff

```
bitbake virtual/kernel -c menuconfig
# using menuconfig to change kernel config here
bitbake virtual/kernel -c diffconfig
```

To build kernel

```
bitbake virtual/kernel
```

Write image to SD card

This section discusses how to use bmaptool to copy image file to SD card.

Copy image file to host

To copy image files from the Docker container to host, use docker cp

```
docker cp \
  berrydev:/home/yoctouser/berry/build/tmp/deploy/images/raspberrypi/core-i
  build/tmp/deploy/images/raspberrypi/
docker cp \
  berrydev:/home/yoctouser/berry/build/tmp/deploy/images/raspberrypi/core-i
  build/tmp/deploy/images/raspberrypi/
```

To write image to a SD card, use bmaptool.

Install and use bmaptool on Ubuntu

```
sudo apt install bmap-tools
```

Use `lsblk` to find SD card device, unmount boot and root partitions, if mounted, and write image to SD card

```
lsblk
sudo umount /dev/mmcblk0p1
sudo umount /dev/mmcblk0p2
sudo bmaptool copy \
  --bmap build/tmp/deploy/images/raspberrypi0-wifi/core-image-base-raspberr
  build/tmp/deploy/images/raspberrypi0-wifi/core-image-base-raspberrypi-202
  /dev/mmcblk0
```

Install and use bmaptool on macOS

```
git clone https://github.com/intel/bmap-tools.git
cd bmap-tools
python3 setup.py install
pip3 install six
```

Find SD card device using `diskutil list`, then unmount disk, and write image to SD card

```
diskutil unmountDisk /dev/disk2
sudo bmaptool copy \
  --bmap build/tmp/deploy/images/raspberrypi0-wifi/core-image-base-raspber
  build/tmp/deploy/images/raspberrypi0-wifi/core-image-base-raspberrypi-20
  /dev/rdisk2
```

Note the `r` in device path i.e. `/dev/rdisk2` in bmaptool command.

Login shell

A login shell is available through HDMI display. Log in as root with a blank password. You can set a password using `passwd root`.

Serial console

To log in to device without an HDMI display, use a USB to TTL Serial 3.3V cable connected to the expansion header.

To disable serial console, comment out the following section in `kas.yml`

```
console: |
  ENABLE_UART = "1"
```

Enable ADB daemon

ADB daemon is enabled by default. To disable USB debugging at build time, remove `USB_DEBUGGING_ENABLED` or set to “0”.

Subsequently, to enable ADB daemon on the device

```
touch /var/usb-debugging-enabled
reboot
```

On a host machine connected to Raspberry Pi Zero W via USB OTG port, list the device and gain shell access

```
adb devices
adb shell
```

Configure audio

`alsa-utils` package is built into the image. You can disable it by removing the `audio` section in `kas.yml`.

To see a list of device names

```
aplay -L
```

To play test sound to HDMI display

```
speaker-test -c2 iec958
```

Use `alsamixer` and `amixer` to change audio settings.

WM8960 soundcard support

Support is enabled for ReSpeaker 2-Mic Pi Hat and similar hats.

You need to adjust volume and toggle some switches that are off by default

```
amixer -c1 sset 'Headphone',0 80%,80%
amixer -c1 sset 'Speaker',0 80%,80%
amixer -c1 sset 'Left Input Mixer Boost' toggle
amixer -c1 sset 'Left Output Mixer PCM' toggle
amixer -c1 sset 'Right Input Mixer Boost' toggle
amixer -c1 sset 'Right Output Mixer PCM' toggle
```

The following should output white noise

```
speaker-test -c2 -Dhw:1
```

The following should loop mic input to speaker

```
arecord -f cd -Dhw:1 | aplay -Dhw:1
```

Note that audio capture on a ReSpeaker 2-Mic Pi Hat is very noisy.

Configure Wi-Fi

Use `wpa_passphrase` utility to print out network configuration

`wpa_passphrase ssid password`

Copy the output and add, all but the commented out plain text password line, to end of `/etc/wpa_supplicant.conf`.

Restart wpa_supplicant service

`systemctl restart wpa_supplicant`

With the board on the network, you can access it using ssh from a host.

Configure Software Access Point

If you want to configure Raspberry Pi as a software access point (SoftAP/hotspot) and access it via ssh, follow the instructions at Setting up a Raspberry Pi as a routed wireless access point.

Configure Bluetooth

Bring up interface and make device discoverable

```
hciconfig hci0 up
hciconfig hci0 piscan
```

DBUS can also used to bring up interface programmatically

```
dbus-send --system --print-reply \
  --dest=org.bluez \
  /org/bluez/hci0 \
  org.freedesktop.DBus.Properties.Set \
  string:"org.bluez.Adapter1" \
  string:"Powered" \
  variant:boolean:true
```

Matter

Matter is a connectivity standard and implementation for home devices. The meta-matter layer builds and installs the `chip-all-clusters-app` in the device image.

Wi-Fi provisioning

Provision Wi-Fi network using `chip-tool` over Bluetooth LE

```
chip-tool pairing ble-wifi ssid "password" 0 20202021 3840
```

To toggle onoff switch at endpoint 1

```
chip-tool onoff toggle 1
```

Clear storage and start over

```
rm /tmp/chip_*
```