OpenEmbedded/Yocto Project for Kernel Developers

Daniel Thompson

Tuesday, September 6, 2022 | 9 mins read

Kernel Development Linux Kernel OpenEmbedded The Yocto Project Training Yocto

OpenEmbedded and the Yocto Project provide powerful tools that are capable of building fully fledged GNU/Linux distributions complete with OS images and package streams ready to push to your update servers. However OpenEmbedded can also be scaled back and used to build simple custom root file systems suitable for kernel development and testing. In this blog post we will attempt to map the shortest route a kernel programmer could take from the initial git fetch to having a working filesystem image ready to deploy to a device.

What's so special about kernel development?

In the context of embedded Linux distributions then the most obvious, and most unique, thing about kernel developers is that they normally bring their own kernel; they do not need or want any kernel to be bundled with the filesystem! For that reason this post will not mention anything about how to cross-compile the kernel. We will assume you already have that covered and that you have a kernel in `arch/${ARCH}/boot/Image` ready to go. You just need a userspace to partner with it!

The other major thing that affects kernel hackers more than other developers is that, at least some of the time, they will spend time working on kernels with missing or incomplete drivers. It is therefore useful to have a test image that can be deployed on minimised kernels with few features enabled.

In short, to meet a kernel developer's needs we want to build images such that:

- The filesystem is available in multiple formats: cpio (to allow run-from-RAM systems based on initramfs), ext4 filesystem image and tarball (for flexible and creative deployment)
- It boots on a wide variety of hardware with sane defaults (e.g. serves a getty on the serial ports, brings up eth0 automatically, etc) and with some simple tools for driver testing
- It includes an SSH server for remote shell access and file transfer
- It has the ability to enrich the image with additional custom tools and libraries

Quickstart guide

This step by step guide applies all of the ideas above to generate a root filesystem that meets all of the above for a 64-bit Armv8-A system:

1. Install all the prerequisite host packages described in the Development Tasks Manual.
2. Download an appropriate poky branch (at the time of writing, kirkstone is the most suitable branch):

```
git clone git://git.yoctoproject.org/poky -b kirkstone
```

3. Source the environment variables required to build OpenEmbedded systems and, optionally, change the prompt for this session so that you can more easily keep track of which sessions have the environment configured to build/rebuild images.

```
cd poky
. oe-init-build-env
PS1="[bitbake] $PS1"
```

Note: *oe-init-build-env* will change your current working directory and, in order to run bitbake commands, this step will need to be repeated every time you start a new terminal session.

4. Create a new machine configuration file inside the bitbake build directory:

```
mkdir -p conf/machine
cat > conf/machine/v8a-arm64.conf <<'EOF'
#@TYPE: Machine
#@NAME: v8a-arm64
#@DESCRIPTION: Generic Arm64 machine for generating a basic rootfs

require conf/machine/include/arm/arch-armv8a.inc

# Don't build an actual kernel
PREFERRED_PROVIDER_virtual/kernel ?= "linux-dummy"

# Generate filesystem as initramfs, ext4 and tarball
IMAGE_FSTYPES ?= "cpio.gz cpio.xz ext4.gz tar.xz"

# List the most common device names for serial ports on Arm systems
# (and use SERIAL_CONSOLES_CHECK to avoid errors for non-existent
# devices)
SERIAL_CONSOLES = "115200;ttyS0 115200;ttyS1"
SERIAL_CONSOLES += "115200;ttyAMA0"
SERIAL_CONSOLES += "115200;ttyMSM0"
SERIAL_CONSOLES += "115200;hvc0"
SERIAL_CONSOLES_CHECK = "${SERIAL_CONSOLES}"

# This is just a guess about what features the kernel has drivers
# for. It doesn’t matter it the kernel doesn’t actually implement
# everything here.
MACHINE_FEATURES:append = " alsa bluetooth rtc screen usbhost vfat wifi"
EOF
```

The role of the machine configuration file is to tell OpenEmbedded what target device you are building for. This includes things like compiler tuning (this is handled by `arch-armv8a.inc`), choice of kernel, what image formats are best suited for the machine and what features the machine has or could have. The machine configuration is mostly one-off, unless you need to add support for additional names for the serial port (or for a whole different architecture) then you won’t need to edit this much after you have created it.

5. Now we need to modify `local.conf` to adopt the above machine configuration. We will also use this to make personal (“local”) configuration tweaks to enrich the example images with additional features and packages:

```
cp conf/local.conf conf/local.conf.bak
cat - conf/local.conf.bak > conf/local.conf <<'EOF'
# Choose the simple 64-bit Armv8-A machine we just created
MACHINE ?= "v8a-arm64"

# Permit root login without a password
EXTRA_IMAGE_FEATURES += "debug-tweaks"

# Enable SSH access
EXTRA_IMAGE_FEATURES += "ssh-server-dropbear"

# Adding the init-ifupdown package ensures that eth0, if it exists,
# will be brought up at boot time using DHCP.
CORE_IMAGE_EXTRA_INSTALL += "init-ifupdown"

# vim rocks... so lets add that to the package list too
CORE_IMAGE_EXTRA_INSTALL += "vim-tiny"

# Conserve disk space during the build by remove working
# directories as we go
INHERIT += "rm_work"

#
# Original contents of local.conf files
#

EOF
```

6. Done! All that is left is to kick off the build:

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

This initial run will take a long time to complete as the build system downloads and builds the required components. Even on relatively powerful machines with fast network connections then initial build times of an hour or more would not be unexpected.

Once completed the resulting images can be found in `$BUILDDIR/tmp/deploy/images/v8a-arm64`. The root filesystem will have been prepared as an initramfs (use cpio.gz or cpio.xz files depending on which decompressors you have enabled in your kernel), a compressed ext4 filesystem and as a tarball.

Testing using qemu

Qemu is usually an attractive tool to run a quick sanity test on your new userspace. Qemu is usually well supported by defconfig kernels meaning, if you have already got a kernel compiled and ready for testing then we are one command away from booting the new userspace:

```
qemu-system-aarch64 -nographic \
    -cpu cortex-a72 -M virt -smp 4 -m 2048 \
    -nographic -nic user,model=virtio \
    -kernel arch/arm64/boot/Image \
    -initrd $BUILDDIR/tmp/deploy/images/v8a-arm64/core-image-base-v8a-arm64.cpio.xz
[    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd083]
[    0.000000] Linux version 6.0.0-rc1 (drt@maple) (gcc (Debian 12.1.0-8) 12.1.0, GNU ld (GNU
Binutils for Debian) 2.38.90.20220713) #3 SMP PREEMPT Thu Aug 18 13:58:25 BST 2022
…
[    8.955913] Freeing unused kernel memory: 7168K
[    8.976578] Run /init as init process
INIT: version 3.01 booting
Framebuffer /dev/fb0 not detected
Boot splashscreen disabled
Starting udev
[    9.109321] udevd[146]: starting version 3.2.10
[    9.114671] udevd[147]: starting eudev-3.2.10

Poky (Yocto Project Reference Distro) 4.0 v8a-arm64 ttyAMA0

v8a-arm64 login:
```

Aside: Injecting modules into initramfs images

Kernel developers who are working with modular kernels and real filesystems (like ext4) can easily loop mount the filesystem and install modules directly into the filesystem. However it can be more of a challenge to inject modules into an initramfs image. Nevertheless with a little care this can be automated by getting kbuild to install the modules into a temporary directory:

```
# Cleanup (if needed) and then install the modules into a directory
rm -rf mod-rootfs
make modules_install INSTALL_MOD_PATH=$PWD/mod-rootfs INSTALL_MOD_STRIP=1

# Compress the module overlay as a separate cpio archive and append
# it to the rootfs image. The kernel will automatically unpack the
# concatenated archives into the initial ramfs. Note that the
# -Ccrc32 flag is required on recent distros to ensure the xz
# checksums are compatible with the kernel decompressor.
(cd mod-rootfs; find . | cpio --create -H newc --quiet | xz -cvT0 -Ccrc32) \
    | cat $BUILDDIR/tmp/deploy/images/v8a-arm64/core-image-base-v8a-arm64.cpio.xz -
    > rootfs.cpio.xz
```

After using the above commands the combined archive can be found at `rootfs.cpio.xz` ready to be deployed to the target.

What about other architectures?

In the quickstart guide above we used arm64 as an example but the concepts and principles apply equally to other architectures. For example we can easily adapt the above instructions to generate a machine description suitable for 32-bit Arm platforms.

```
cat > conf/machine/v7a-arm32.conf <<'EOF'
#@TYPE: Machine
#@NAME: v7a-arm32
#@DESCRIPTION: Generic Armv7-A machine (with NEON) for generating a basic rootfs

# Tuning for all of Armv7-A is slightly tricky (because there are
# multiple FPU configurations). This tuning is right for most (but
# not all of) the Cortex-A family.
DEFAULTTUNE ?= "armv7athf-neon"
onf/machine/include/arm/arch-armv7a.inc

# Don't build an actual kernel
PREFERRED_PROVIDER_virtual/kernel ?= "linux-dummy"

e filesystem as initramfs, ext4 and tarball
YPES ?= "cpio.gz cpio.xz ext4.gz tar.xz"

# List the most common device names for serial ports on Arm systems
e SERIAL_CONSOLES_CHECK to avoid errors for non-existent
)
SERIAL_CONSOLES = "115200;ttyS0 115200;ttyS1"
SERIAL_CONSOLES += "115200;ttyAMA0"
SERIAL_CONSOLES += "115200;ttyMSM0"
NSOLES += "115200;hvc0"
NSOLES_CHECK = "${SERIAL_CONSOLES}"

# This is just a guess about what features the kernel has drivers
doesn't matter if the kernel doesn't actually implement
ing here
MACHINE_FEATURES:append = " alsa bluetooth rtc screen usbhost vfat wifi"
EOF
```

Adding machine configurations that support other architectures is left as an exercise for the reader (hint: take a look at `$BUILDDIR/../meta/conf/machine/include`). Note also that users of x86 systems do not need to add any machine configuration at all since the BSP provides a generic x86 (64) machine so simply setting `MACHINE = "genericx86-64"` in `local.conf` is enough to 64 images!

Coming soon

t one of a two part series.

So far you have learned how to build a simple, compact and useful userspace which weighs in at about 12MB (compressed). The comments we added to `local.conf` should give you some clues about how to extend the image with additional features. For example adding the following to `local.conf` does exactly what you (hopefully) think it does:

```
E_EXTRA_INSTALL += "alsa-tools socat stress-ng rt-tests"
```

However even with the extensive commenting there are plenty of customization tricks I haven't shared yet. Look out for part two where we'll cover other ways to tweak the local configuration to make things either more compact or more feature rich.

son is the Technical Lead for the Support and Solutions team at Linaro and, among other things, leads the [Linaro training](https://www.linaro.org) other members and our services customers. He is an experienced kernel developer with a long interest in cross-compiled GNU/Linux distributions. He frequently delivers both our OpenEmbedded/Yocto training and our intermediate and advanced courses on Linux kernel development and debugging.

Recent Posts

Multi-Circular Queue (MCQ) support gets added to the UFS subsystem

Tue Feb 14 2023

In this blog, we look at Multi-Circular Queue (MCQ) support and how it is being implemented in the Linux kernel. Click here to read more!

Top Byte Ignore For Fun and Memory Savings

Wed Feb 08 2023

In this article, David Spickett talks about how Top Byte Ignore works and how to use it. Read more here!

Linaro & Huawei Tech Day

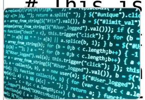
Fri Feb 03 2023

Come see the latest Open Source advancements and plans from Huawei and Linaro.

Linaro to Acquire Arm Forge Software Tools Business

Mon Jan 30 2023

In this press release Linaro announces the acquisition of the Arm Forge Software Tools Business. Read more here!

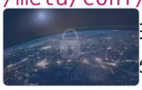


HTTP now supported in U-boot

Tue Jan 24 2023

In this blog, Paul Liu talks about how HTTP is now supported in U-boot and how you can enable this feature. Read more here!

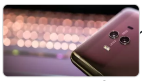
Other Posts



OP-TEE Test: Testing a Trusted Execution Environment

Wed Feb 10 2016

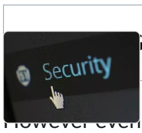
In this article, Joakim Bech provides a general background about OP-TEE as well as testing OP-TEE using a tool called xtest (optee_test). Read more here!



Android 13 now available on Qualcomm Robotics Reference RB3 and RB5 Platforms

Thu Aug 18 2022

This blog talks about the Android 13 release and how it boots straight out of the box on the Linaro supported Reference Boards RB5 and RB3.



New Trust Sources for Linux Kernel Keyrings

Tue Nov 02 2021

In this blog, our Engineer talks about how Linaro helped generalize the Trusted Keys sub-system in Linux to add support for new trust sources. Read more here.

Linaro releases LEDGE Reference Platform v0.2

Thu May 20 2021

This blog talks about the LEDGE Reference Platform v0.2 release and what new features users can expect to see.

Linaro contributions to the 5.17 Linux Kernel Release

Sat Mar 12 2022

In this blog we talk about Linaro's contributions to the 5.17 Linux Kernel Release.