Building the Cross-Compilation Toolchain

Pieter P

To compile software for the Raspberry Pi, you need a cross-compilation toolchain. It's a collection of files and programs that you can run on your computer, and that produce a binary that can be executed on the Raspberry Pi.

If you want to do development on the Pi itself, you'll need a native toolchain as well. This is a toolchain that runs on the Pi, and produces binaries for the Pi.

Docker

As explained on the previous page, building the toolchain happens inside of a Docker container. This allows you to experiment in a sandbox-like environment. Starting from scratch is really easy, and you don't have to worry about messing up your main Linux installation.

You can compare a Docker container with a virtual machine, but without the virtualization. Programs that run inside of the container use the same kernel as your main Linux OS, and there's almost no performance overhead.

It's very easy to share Docker images, so you don't have to build everything from scratch yourself, you can just pull it from Docker Hub.

Dockerfiles

A Dockerfile describes how the Docker image is built. In this project, we'll start from a standard Ubuntu image, install some build tools, and then compile the toolchain and the dependencies. Each step of the build process creates a new layer in the image. This is handy, because it means that if a build fails in one of the last steps, you can just fix it in your Dockerfile, and build it again. It'll then start from the last layer that was successfully built before, you don't have to start from the beginning (which would take a while, since we'll be building many large projects.)

The actual Dockerfiles used for the build can be found on my GitHub, I'll briefly go over them on this page.

Pulling the toolchain from Docker Hub

If you just want to use the toolchain, without understanding how it works, and without changing anything to the configuration, you can just pull the toolchains I built from Docker Hub.

To pull and export the toolchain, you can use:

```
$ ./toolchain/toolchain.sh <board> --export-toolchain
```

Where `<board>` is one of the following:

- `rpi`: Raspberry Pi 1 or Zero, cross-compilation toolchain
- `rpi-dev`: Raspberry Pi 1 or Zero, cross-compilation toolchain and native toolchain
- `rpi3-armv8`: Raspberry Pi 3, 32-bit, cross-compilation toolchain
- `rpi3-armv8-dev`: Raspberry Pi 3, 32-bit, cross-compilation toolchain and native toolchain
- `rpi3-aarch64`: Raspberry Pi 3, 64-bit, cross-compilation toolchain
- `rpi3-aarch64-dev`: Raspberry Pi 3, 64-bit, cross-compilation toolchain and native toolchain

Building the toolchain using the provided shell scripts

Instead of pulling from Docker Hub, you can build the toolchain locally instead. I would only recommend this if you made changes to the toolchain configuration files, or if you have a really fast computer with a slow internet connection.

Building one toolchain took around 25 minutes on a 2018 Dell XPS i7-8750H, and 40 minutes on a 2017 Dell XPS i7-7700HQ.

```
$ ./toolchain/toolchain.sh <board> --build-toolchain --export-toolchain
```

The cross-compilation toolchain will be located in `toolchain/x-tools/<arch>-rpi*-linux-<abi>`, and the native toolchain will be in `toolchain/x-tools/HOST-<arch>-linux-<abi>/<arch>-rpi*-linux-<abi>`.

Deleting the toolchains

The toolchains are read-only, to prevent you from accidentally breaking them during a build. If you try to delete them using the usual methods, you'll get `Permission denied`. The solution is to make them writable first, using `chmod`.

The `toolchain/clean.sh` script will do this for you, and deletes all toolchains in the `toolchain` folder:

```
$ ./toolchain/clean.sh
```

The toolchains will still be available in the Docker images used to build them. This means that next time you run the `build-and-export-toolchain.sh` script, it will finish in just a couple of seconds, as it just has to export the toolchain, it doesn't build it from scratch.

If you want to delete these Docker images as well (to save disk space, for instance), you can use `docker image ls` to inspect all images on your system, and then you can delete them using `docker image rm <image>`.

Detailed information about configuring and building toolchains

As mentioned before, you don't need to read or understand the sections below if you just want to use the toolchain.
If you're interested in how the toolchains are actually configured and built, or if you want to customize the configuration, you'll find the necessary information below.

Crosstool-NG

The toolchain is built using Crosstool-NG.

It is installed in a CentOS 7 Docker container, because CentOS 7 was the oldest OS that I had to run the toolchain on. In this context, oldest refers to the Linux kernel version and the glibc version. They are backwards compatible, so you can run software compiled for an old version on a computer with a newer version, but you can't run software compiled for a new version on a computer with an older version.

The following Dockerfile downloads, builds and installs Crosstool-NG to the `~/.local` directory of the container.

`Dockerfile`

```
FROM centos:7 as ct-ng

# Install dependencies to build toolchain
RUN yum -y update && \
    yum install -y epel-release && \
    yum install -y autoconf gperf bison file flex texinfo help2man gcc-c++ \
    libtool make patch ncurses-devel python36-devel perl-Thread-Queue bzip2 \
    git wget which xz unzip && \
    yum clean all

# Add a user called `develop` and add him to the sudo group
RUN useradd -m develop && echo "develop:develop" | chpasswd && \
    usermod -aG wheel develop

USER develop
WORKDIR /home/develop

# Download and install the latest version of crosstool-ng
RUN git clone -b master --single-branch --depth 1 \
        https://github.com/crosstool-ng/crosstool-ng.git
WORKDIR /home/develop/crosstool-ng
RUN git show --summary && \
    ./bootstrap && \
    ./configure --prefix=/home/develop/.local && \
    make -j$(($(nproc) * 2)) && \
    make install &&  \
    cd && rm -rf crosstool-ng
ENV PATH=/home/develop/.local/bin:$PATH
```

The list of dependencies can be found on Crosstool-NG's GitHub: https://github.com/crosstool-ng/crosstool-ng/blob/master/testing/docker/centos7/Dockerfile

Cross-Compilation Toolchain

The following Dockerfile builds the toolchain:

`Dockerfile`

```
FROM tttapa/crosstool-ng-master as cross-toolchain

ARG HOST_TRIPLE

WORKDIR /home/develop
RUN mkdir /home/develop/RPi && mkdir /home/develop/src
WORKDIR /home/develop/RPi
COPY ${HOST_TRIPLE}.config .config

RUN ct-ng build || { cat build.log && false; } && rm -rf .build

ENV TOOLCHAIN_PATH=/home/develop/x-tools/${HOST_TRIPLE}
ENV PATH=${TOOLCHAIN_PATH}/bin:$PATH
WORKDIR /home/develop
```

The different configuration files have names that contain the full target triplet. They can be found in same folder as the Dockerfile.

Raspberry Pi 3, 64-bit (AArch64)

This configuration is based on the `aarch64-rpi3-linux-gnu` sample that comes with Crosstool-NG: https://github.com/crosstool-ng/crosstool-ng/tree/master/samples/aarch64-rpi3-linux-gnu

I changed the GCC version to the latest stable one, and the Linux kernel and glibc versions to match the versions that the Ubuntu Server 18.04 image ships with.

Raspberry Pi 3, 32-bit (ARMv8)

This configuration is based on the `armv8-rpi3-linux-gnueabihf` sample that comes with Crosstool-NG: https://github.com/crosstool-ng/crosstool-ng/tree/master/samples/armv8-rpi3-linux-gnueabihf

Even though the CPU will be running in 32-bit mode, you can still use the ARMv8 NEON instructions, so I changed the compiler's default FPU flag to `neon-fp-armv8`. I haven't tested if it actually makes any difference.

I changed the GCC version to the latest stable one, and the Linux kernel and glibc versions to match the versions that the Ubuntu Server 18.04 image ships with. These are also supported on Raspbian Buster.

Raspberry Pi 1 & Zero, 32-bit (ARMv6)

This configuration is based on the `armv6-rpi-linux-gnueabihf` sample that comes with Crosstool-NG: https://github.com/crosstool-ng/crosstool-ng/tree/master/samples/armv6-rpi-linux-gnueabi

Native toolchain

Building the native toolchain for the Raspberry Pi is very similar.

`Dockerfile`

```
ARG HOST_TRIPLE

FROM tttapa/rpi-cross-toolchain:${HOST_TRIPLE} as cross-native-toolchain

ARG HOST_TRIPLE

RUN rm -rf /home/develop/RPi && mkdir /home/develop/RPi
WORKDIR /home/develop/RPi
COPY ${HOST_TRIPLE}.config .config
RUN ct-ng build || { cat build.log && false; } && rm -rf .build
WORKDIR /home/develop
```

The toolchain type is "Cross-Native", and is implemented as a special case of the "Canadian Cross" toolchain. It uses the cross-compilation toolchain that was built earlier. More information can be found in the Crosstool-NG documentation.

Customizing the toolchain

You can customize all previously mentioned toolchains by running `ct-ng menuconfig` on the corresponding configuration file. The easiest way to do this is by running it inside of the `crosstool-ng-master` Docker container.

```
$ cd RPi-Cpp-Toolchain/
$ ./toolchain/docker/crosstool-ng-master/build.sh
$ docker run -it --rm --volume "$PWD/toolchain/docker:/mnt" tttapa/crosstool-ng-master
```

```
[docker] $ cd /mnt/merged/cross-toolchain/
[docker] $ cp aarch64-rpi3-linux-gnu.config .config  # Choose RPi model and architecture
[docker] $ ct-ng menuconfig
```

Now make your changes and save the result to `.config`.

```
[docker] $ # Rename the old configuration
[docker] $ mv aarch64-rpi3-linux-gnu.config aarch64-rpi3-linux-gnu.config.old
[docker] $ # Replace with new configuration
[docker] $ mv .config aarch64-rpi3-linux-gnu.config
[docker] $ exit
```

Now you can build the toolchain again using the `./toolchain.sh` script, as explained earlier:

```
$ ./toolchain/toolchain.sh <board> --build-toolchain --export-toolchain
```