Cross-Compiling the Dependencies

Pieter P

Sysroot and Staging Area

Because we don't have access to the actual root directory of the Raspberry Pi, with all of its system files and libraries, the toolchain uses a so-called **sysroot** folder. It contains the necessary system libraries, such as glibc and the C++ standard library. It's also the folder where the configure scripts of other libraries will look for the necessary libraries and headers. The sysroot is generated by Crosstool-NG, and it is part of the cross-compilation toolchain. It is not used for deploying your software to the Pi.

The sysroot of the toolchain is read-only, to keep it clean for future projects, so we'll make a copy of the sysroot for this build, and make it writable.
This copy will be used as the sysroot for all compilations, and all cross-compiled libraries will be installed to this folder. We have to do this, because the configure scripts (Autoconf's configure script, CMake, etc.) of other libraries or programs will search for their dependencies in the sysroot. If these dependencies are not found, configuration or compilation will fail, or parts of the program/library may be implicitly disabled.

Apart from the sysroot, we also need a folder containing the files we want to install to the Pi. It should contain the binaries we cross-compiled (such as `/usr/local/bin/python3.8`) and the necessary libraries (such as `/usr/local/lib/libpython3.8.so`). It doesn't contain the system libraries, because the Pi already has these installed. This folder is called the **staging area**.

Having both a sysroot and a staging area means we have to install every library twice, once in each of the two folders.
The sysroot is later used when compiling our program, the staging area will be copied to the Pi for running our program.

Cross-Compiling the dependencies using the provided shell scripts

To build all dependencies, you can use the shell script provided in the `toolchain` folder:

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

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

- `rpi`: Raspberry Pi 1 or Zero, dependencies only
- `rpi-dev`: Raspberry Pi 1 or Zero, dependencies and development tools
- `rpi3-armv8`: Raspberry Pi 3, 32-bit, dependencies only
- `rpi3-armv8-dev`: Raspberry Pi 3, 32-bit, dependencies and development tools
- `rpi3-aarch64`: Raspberry Pi 3, 64-bit, dependencies only
- `rpi3-aarch64-dev`: Raspberry Pi 3, 64-bit, dependencies and development tools

The dependencies that are cross-compiled are:

- **Zlib**: compression library (OpenSSL and Python dependency)
- **OpenSSL**: cryptography library (Python dependency)
- **FFI**: foreign function interface (Python dependency, used to call C functions using ctypes)
- **Bzip2**: compression library (Python dependency)
- **GNU ncurses**: library for text-based user interfaces (Python dependency, used for the console)
- **GNU readline**: library for line-editing and history (Python dependency, used for the console)
- **GNU dbm**: library for key-value data (Python dependency)
- **SQLite**: library for embedded databases (Python dependency)
- **UUID**: library for unique identifiers (Python dependency)
- **libX11**: X11 protocol client library (Tk dependency)
- **Tcl/Tk**: graphical user interface toolkit (Python/Tkinter dependency)
- **Python 3.8.2**: Python interpreter and libraries
- **ZBar**: Bar and QR code decoding library
- **Raspberry Pi Userland**: VideoCore GPU drivers
- **VPX**: VP8/VP9 codec SDK
- **x264**: H.264/MPEG-4 AVC encoder
- **Xvid**: MPEG-4 video codec
- **FFmpeg**: library to record, convert and stream audio and video
- **OpenBLAS**: linear algebra library (NumPy dependency)
- **NumPy**: multi-dimensional array container for Python (OpenCV dependency)
- **SciPy**: Python module for mathematics, science, and engineering
- **OpenCV 4.2.0**: computer vision library and Python module

The development tools that are cross-compiled are:

- **GCC 9.2.0**: C, C++ and Fortran compilers (see native toolchain on the previous page)
- **GNU Make**: build automation tool
- **Ninja**: faster, more light-weight build tool
- **CMake**: build system

- Distcc: distributed compiler wrapper (uses your computer to speed up compilation on the RPi)
- CCache: compiler cache
- cURL: tool and library for transferring data over the network (Git dependency)
- Git: version control system

Pulling the cross-compiled dependencies from Docker Hub

If you don't want to change anything to the build process, or if you have a slow computer, you can just pull the Docker images that I compiled from Docker Hub:

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

Detailed information about cross-compilation of libraries

Base Image

Before building the dependencies, I created a base image with Ubuntu and the necessary tools installed. I also created a non-root user.

`Dockerfile`

```
FROM ubuntu:latest as rpi-cpp-toolchain-base-ubuntu

# Install some tools and compilers + clean up
RUN apt-get update && \
    apt-get install -y sudo git wget \
                       gcc g++ cmake make autoconf automake \
                       gperf diffutils bzip2 libbz2-dev xz-utils \
                       flex gawk help2man libncurses-dev patch bison \
                       python-dev gnupg2 texinfo unzip libtool-bin \
                       autogen libtool m4 gettext pkg-config && \
    apt-get clean autoclean && \
    apt-get autoremove -y && \
    rm -rf /var/lib/apt/lists/*

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

USER develop
WORKDIR /home/develop
```

Cross-Compiling the dependencies

If you want to write a program that uses the OpenCV library, you have to cross-compile OpenCV and install it to the Raspberry Pi. But in order to cross-compile OpenCV itself, you need to cross-compile all of its dependencies as well. This can keep going for a while, and you may end up with a pretty large hierarchy of dependencies.

The main Dockerfile discussed below sets up the build environment, and then runs the installation scripts in the `toolchain/docker/merged/cross-build/install-scripts` folder. If you want to omit some of the libraries, you can comment them out in the Dockerfile, but keep in mind that it might break other libraries that depend on it.
Some libraries such as SciPy need to be patched to get them to cross-compile correctly. These patches can be found in the `patches` folder.

For most packages, the build procedure is very simple:

1. Download
2. Extract
3. Run the `configure` script with the right options
4. `make`
5. `make install`

An example is given below.

Board-specific configuration

The configuration for the different Raspberry Pi models is passed to the build scripts using environment variables. For example:

`aarch64-rpi3-linux-gnu.env`

```
export CROSS_TOOLCHAIN_IMAGE="aarch64-rpi3-linux-gnu"
export HOST_ARCH="aarch64"
export HOST_TRIPLE="aarch64-rpi3-linux-gnu"
export HOST_TRIPLE_NO_VENDOR="aarch64-linux-gnu"
export HOST_TRIPLE_LIB_DIR="aarch64-linux-gnu"
export HOST_BITNESS=64
```

Compiling a library for the Docker container

In the first section of the main Dockerfile, some packages are built for both the build machine (the Docker container) and for the host machine (the Raspberry Pi), because we need to build Python for both machines in order to cross-compile the OpenCV and NumPy modules later on.

Since these are just basic native installations, you can simply follow the documentation for the library in question. All of the necessary tools should be installed in the base image.

Cross-Compiling a library for the Raspberry Pi

Cross-compiling is a bit harder, because most libraries don't provide good documentation about the cross-compilation process. However, once you understand the concepts, you'll be able to apply them to almost any library, and with the necessary tweaks, you should be able to get it working.

We'll have a look at a typical example, compiling `libffi`, a foreign function interface library that is used by Python's `ctypes` module.

`libffi.sh`

```
1  #!/usr/bin/env bash
2
3  set -ex
4
5  # Download
6  version=3.3
7  URL="https://codeload.github.com/libffi/libffi/tar.gz/v$version"
8  pushd "${DOWNLOADS}"
9  wget -N "$URL" -O libffi-$version.tar.gz
10 popd
11
12 # Extract
13 tar xzf "${DOWNLOADS}/libffi-$version.tar.gz"
14 pushd libffi-$version
15
16 # Configure
17 . cross-pkg-config
18 ./autogen.sh
19 ./configure \
20     --host="${HOST_TRIPLE}" \
21     --prefix="/usr/local" \
22     CFLAGS="-O3" \
23     CXXFLAGS="-O3" \
24     --with-sysroot="${RPI_SYSROOT}"
25
26 # Build
27 make -j$(($(nproc) * 2))
28
29 # Install
30 make install DESTDIR="${RPI_SYSROOT}"
31 make install DESTDIR="${RPI_STAGING}"
32
33 # Cleanup
34 popd
35 rm -rf libffi-$version
```

Downloading and extracting

The first couple of lines are really straightforward, they simply download the library from GitHub, extract it, and enter the library's directory.

pkg-config

Most configure scripts use `pkg-config`, a simple tool to find libraries that are installed on the system, and to determine what flags are necessary to use them. We want `pkg-config` to find the libraries in the Raspberry Pi sysroot, not in the root folder of the Docker container, because these libraries are for the wrong architecture.
The `cross-pkg-config` script that is sourced on line 16 sets some environment variables in order to tell `pkg-config` to only search for libraries in the Raspberry Pi sysroot.

Autoconf

Many older libraries use GNU Autoconf to configure the project before building. Autoconf generates the `configure` script, and then the `configure` script generates the makefiles that are used to actually compile and install the software. When downloading a library directly from GitHub (or some other source control host), it usually doesn't include the `configure`. In that case, you have to run Autoconf before configuring. If you download a released tarball from the project's website, this isn't usually required.

Configure

The `configure` script checks the system configuration, looks for libraries, and generates the makefiles.
We have to specify that we are cross-compiling by providing the `--host` flag. As mentioned before, the `--prefix` option specifies the directory where the library will be installed on the Raspberry Pi. Software that isn't managed by the system's package manager should be installed to `/usr/local`.
You can add your own compiler flags if you want, using the `CFLAGS`, `CXXFLAGS` and `LDFLAGS` variables. Finally, we tell the configure script to use our custom sysroot, instead of the toolchain's default. Some configure scripts don't support the `--with-sysroot` flag. In that case, you can add the `--sysroot="..."` option to the `CFLAGS`, `CXXFLAGS` and `LDFLAGS` environment variables.

To find out what options you can pass to the `configure` script, you can run `./configure --help`. Usually, a library also comes with some documentation about these options, for example in the `README` or `INSTALL` documents.

Compilation

This is probably the easiest step, you can simply type `make` to compile everyting. To speed up the build, `make`'s `-j` (`--jobs`) option is used to compile multiple source files in parallel. `nproc` returns the number of available processing units (cores/threads) in your computer. It's multiplied by two, and then passed to `make`.

Installation

Once everything has been compiled, we'll install everything in the sysroot and in the staging area. When running `make install DESTDIR="..."`, the files will be installed `DESTDIR/PREFIX`, that is, the `DESTDIR` variable passed to `make` and the `--prefix` specified during configuration will be concatenated. The prefix matters both during compilation/installation and at runtime, the `DESTDIR` just determines where they are installed, and is not used at runtime.
The `DESTDIR` variable also makes it very easy to install the package at two locations, the sysroot and the staging area without building the package twice.

Cleanup

Finally, we just delete the entire build directory to keep the size of the Docker image low. When debugging, you might want to keep the build directory, so you can start a docker container and try out some things in an interactive shell.