LM32-toolchain

Rebuilding a compiler in 2020
March 2020 (5c009f3)

Alessandro Rubini

Table of Contents

Overview

This describes the simplest way (in my opinion) to build a cross-compiler for the ARM micro-controllers (cortex-m0 and cortex-m3 are my target). This is based on previous work of mine, and a collection of ports for the LM32 soft-core I did for GSI (`gsi.de`) related to the White Rabbit project. That project is public at `https://github.com/GSI-CS-CO/lm32-toolchain`.

1 Abstract (TL;DR)

This describes how to build a range of versions of *gcc* for ARM processors. There is specific selection of optimization types or CPU generations

It's just a generic thing to show how it works, but I also use the resulting compilers for my build-tests, and know what strange new warnings are present in recent versions of *gcc*.

2 Initial Remarks

2.1 Other Works

I'm well aware that there are lots of cross-compilers for ARM already available. I won't list them all here, not least because such a list is very volatile in nature.

The point here is getting the feeling of how you build a cross-compiler within the *gcc* suite.

2.2 gcc Version Numbering

Up to gcc-4.x, the second number in the version number was like a major release, so gcc-4.6 was a different thing than gcc-4.5. The third number used to be just a maintenance/bugfix release.

Starting with version 5, the first number is the major release. So gcc-5.3 is a bugfix release of the gcc-5 cycle.

What I'm building/running here is is everything from version 4.8 onwards. If some is missing it's because this is either work in progress or abandoned.

2.3 This Work

Note: this section is meta-information about this package; please goto Chapter 3 [The Build Script], page 2, if you want to start compiling.

This document is quite verbose, because I hope the information can be useful to my students or more. As a side effect, I use it as my own reference while working on this.

The package includes a shell script, configuration files, patches and a document. Binaries are not included.

To build the output formats of the document, just *make* in the `doc/` subdirectory of this repository. You may need to install the *texinfo* package.

The document is written in TeXinfo, the GNU project documentation standard. It may be old-fashioned, but it revealed a future-proof choice, when I made it. However, I love being able to place white space at the beginning of the lines, to make sense of the file in my editor, and avoid markup in @example section so I can copy them from the shell terminal into the editor (or from it to the shell). For this reason, I preprocess the real input file. The source file is *arm-toolchain.in* and not the `.texinfo` one; if you edit the latter, it will be overwritten at the next built (that's why I make it read-only, in Makefile).

If you pick stuff from here and you want to move to a different source format, please consider using the source file *arm-toolchain.in* as your starting point, as an alternative the text output file might be better, or the *html* output if you can import the italics and tty-face markup.

3 The Build Script

Obviously, everybody and their friend wrote a build script. Some are simple sequences of commands, some are very complex tools, like *buildroot* or (got forbid!) *yocto*. To show the concept while providing an almost-useful too, I prefer something in the middle: there is some factorization but not much, to keep things simple. If I need a real ARM compiler I use *buildroot*

Fact is, my script of 2010 still works fine, so I recycle it here. It is a shell script (`tools/build-generic`) that relies on a configuration file, which is a dozen lines long. Mainly, you state which package versions to download and what build options to apply. For example:

```
PREFIX="$(/bin/pwd)/install/arm-gcc-$(date +%y%m%d-%H%M)"
TARGET="arm-none-eabi"
GCC_CONFIG="--disable-libssp"

# table of programs,versions and so on
prog gcc      8.2.0    xz   https://ftp.gnu.org/gnu/gcc/gcc-8.2.0
prog binutils 2.31.1   xz   http://ftp.gnu.org/gnu/binutils
prog newlib   3.0.0    gz    ftp://sources.redhat.com/pub/newlib
[...]
# package list: what to get
plist="gcc binutils gdb newlib mpc mpfr gmp"
```

The script is then called with the configuration file as its only argument. To prevent any error and to be able to recover what your did, the script saves itself and the configuration file to the log file, before the build starts.

The build directory (and log file) currently use a timestamp-based name, because I prefer to keep all build logs, with errors; I don't want any to be overwritten while I work on several builds at the same time.

The installation directory includes the day of the build and the git commit of this repository.

Please note that the configurations in this package define `PREFIX` by themselves. The build script offers a default (within `/opt` but configurations override it.

For example, this is what I'm getting right now:

```
hsw2020% ./tools/build-generic ./configs/gcc-8.2.0
Using ./configs/gcc-8.2.0 as config file
Using PREFIX=//home/rubini/arm-toolchain/install/arm-gcc-200314-c7fe
Building in "/home/rubini/arm-toolchain/build-200314-18-06"
Log file is "/home/rubini/arm-toolchain/build-200314-18-06.log"
```

3.1 Downloads

As a first step, the script downloads source files. The download directory it `./downloads` where you invoke the script; such a directory can be a symbolic link. The script uses the version number, suffix and base URL as in the example above (for *gcc* there is a different directory for each version, thus the duplicate number); it downloads one *tar* file for each *package* listed in the `plist` variable (and for which is uses the corresponding `prog` line above it. If the file is already there, it is not downloaded, without a check of integrity. Files that are already in place can be symbolic links (for example, I already had most of them due to *buildroot* runs over the last years, so I symlinked them all and avoided downloading).

Then, all source *tar* files of interest are uncompressed into `./src/` where you invoked the script. All relevant packages create a directory with the same base name, and this name is preserved. For example:

```
laptopo% tar tf downloads/mpc-1.0.3.tar.gz | head -1
mpc-1.0.3/
```

```
laptopo% ls -d src/mpc*
src/mpc-0.9/  src/mpc-1.0.3/
```

If the target directory within `./src` exists, it is preserved. This allowed me to test my patches easily. You can remove them all to restart a clean build, if you want.

3.2 Applying Patches

The current version of the script applies patches, if any exist in the `patches` subdirectory of this package. It does so by creating a local *git* repository, committing the untarred files and then using `git-am` to patch. This initial commit of the whole source tree can be a very long operation, but fortunately it only applies once for each package, and not many packages are patched.

Currently, I only have one patch for the old binutils version that is used to build gcc-4.9.2. You can use that as an example if you need to add patches.

This is an example from a previous run of this script, building a 4.5.3 version:

```
Uncompressing ../downloads/gcc-4.5.3.tar.bz2...
Patching gcc-4.5.3
Initialized empty Git repository in ..../lm32-toolchain/src/gcc-4.5.3/.git/
Applying: gcc/doc: fix use of @itemx
[...]
```

please note that from "Initialized" to "Applying" above, you may wait more than for the uncompression step (which of the two is longer depends on your disk speed, RAM size and current load).

When patches exist, the script creates a marker file to note that they are already applied. You can remove the marker file (e.g. `gcc-4.5.3-patched`) and the `.git` directory within the package to start clean again.

For each package that I had to patch, I provide the git-generated patch-set in the subdirectory `patches/`*`pkg-x.y`*. This set is what is automatically used by the script.

3.3 The Build Directory

Each run of the script creates a new build directory, called `build-$(date +%y%m%d-%H-%M)` (for example, `200314-12-00` if I built at noon today). The log file has the same name, with a trailing `.log`.

If an error happens and you re-run the build, everything will start again in a different directory. This allows me to ensure I didn't forget something and what works for me will for you as well. If you need to debug a failed build, you can `cd` to the build directory and reproduce the error in there, to try your fixes.

The size of each build directory is from 1.5GB up to 2.3GB, in the range of versions I document here as working (the numbers above may not be up to date as I add new verions, I'm sorry). Don't be shy about removing those when you are done with each of them.

3.4 Using the Log File

The log file includes the complete compilation log, so your can look for errors, but also the script and the configuration file. I did this because I tend to forget saving all information about a build, for example because I edit the configuration file for the next version I try, but save with the same config so to recycle the command line.

To recover the configuration that was used in a build, you can use `tools/recover-config`:

```
laptopo% ./tools/recover-config build-190113-10-00.log > prev-config
```

Similarly, `tools/recover-script` is there, but I never had to change the script for all builds I describe in this document.

To look for errors, please grep for `Error` in the log file (note the upper-case 'E'). If none is there, the build was successful. I think I've been a little lazy with error checking in the script itself.

To find what your build time was (so to plan your coffee break when you run it again), check for lines starting with '`### `':

```
hsw2020% grep '^### ' build-200314-17-22.log
### Sat Mar 14 17:22:24 UTC 2020: config binutils: "../../src/binu[...]"
### Sat Mar 14 17:25:14 UTC 2020: make it all: "make -j 2"
### Sat Mar 14 18:06:19 UTC 2020: install it all: "make install"
### Sat Mar 14 18:06:53 UTC 2020: done: "true"
```

In each line above, the final string is the command being executed, that's why I added to finale `true`: to get a timestamp.

Finally, if you don't remember what build a specific log refers to, `tools/describe-log` extracts a subset of the configuration file, so you know the gcc, binutils and newlib versions, which is the most important information item:

```
hsw2020% ./tools/describe-log build-200314-17-22.log
build-200314-17-22.log: gcc 8.2.0
build-200314-17-22.log: binutils 2.31.1
build-200314-17-22.log: newlib 3.0.0
```

The size of the log file in the range 10-30MB.

3.5 Example Run

This is an example run of the script, as executed on `lx-pool.gsi.de`. Pleae note that I suggest to run in the root directory of this package (possibly after filling `./downloads` with the files you already have), because the `.gitignore` file already supports it.

```
hsw2020% ./tools/build-generic configs/gcc-5.4.0
Using configs/gcc-5.4.0 as config file
Using PREFIX=//home/rubini/arm-toolchain/install/arm-gcc-200314-fd129d-c7fe
Building in "/home/rubini/arm-toolchain/build-200314-15-40"
Log file is "/home/rubini/arm-toolchain/build-200314-15-40.log"
Dowloading https://ftp.gnu.org/gnu/gcc/gcc-5.4.0//gcc-5.4.0.tar.bz2
[...]
Uncompressing ../downloads/gcc-5.4.0.tar.bz2...
Uncompressing ../downloads/binutils-2.28.1.tar.bz2...
[...]
```

Then, the script applies the patches, if any, and builds, finally installing the compiler binary and support files.

4 Host System

The set of builds was tested on Ubuntu-18.04, on x86-64. installation. The host compiler is *gcc-7.5.0*.

I may check on different systems, or forget about it.

5 Installing the Compiler

After building, possibly using my convoluted installation names it is possible to make a *tar* file of the directory and uncompress it in a different pathname. When `arm-none-eabi-gcc` is called, it will find all its support files using relative pathnames.

For example:

```
cd install
D=arm-gcc-200314-c7fe
mv $D arm-gcc-8.2.0
tar cJf arm-gcc-8.2.0.tar.xz arm-gcc-8.2.0
mv arm-gcc-8.2.0 $D
```

Or just copy it in the final place

```
cp -a install/arm-gcc-200314-c7fe  /opt/arm-gcc-8.2.0
```

This works for all *gcc* builds described here.

6 The Various gcc Versions

This chapter documents what I achieved the various gcc versions.

As explained in Section 2.2 [gcc Version Numbering], page 1, the major number is either *4.y* for version numbers *4.y.z* or *x* for versions *x.y.0* with x >= 5.

6.1 Version 4.8

Yet untested, but errors expected, because our current host compiler (*gcc-7*) spits many warnings, and sometimes `-Werror` is used in compiler sources.

Earlier versions are not tested, either, as the older we go the more difficult the build is.

6.2 Version 4.9

This requires a patch to remove `-Werror` from binutils, as our current host compiler complains about fall-through `case:` statements. With the changes I provide in `patches/binutils-2.21.1` everything works.

6.3 Version 5

Version 5.4.0 is known to work, with `configs/gcc-5.4.0`.

6.4 Version 6

Version 6.5.0 is known to work, with `configs/gcc-6.5.0`.

6.5 Version 7

Version 7.4.0 is known to work, with `configs/gcc-7.4.0`.

6.6 Version 8

Version 8.2.0 is known to work, with `configs/gcc-8.2.0`.

6.7 Version 9

Still to be tested

7 Storage Requirements

This is a summary of the storage required for the builds described here, very rough (as I add versions over time)

- 1.0 GB for downloads
- 8-10 GB for uncompressed sources.
- 4-6 GB for installed trees.
- 0.2-0.5 GB in log files.
- 15-20GB for build directories

The exact size will vary according to you filesystem layout and the length of your pathnames (which end up many times as strings in the binaries), and how may versions you build.