Building the GNU toolchain for ARM targets

[This document is also available as PostScript.]

The ARM support in the various GNU compilers and libraries is fairly good, and very workable. Unfortunately, binary distribution of recent versions seems to be a little thin on the ground, and hence this document has been written to help you compile up from source.

If you're after binary versions of native compilers, there are many places you could try; for RedHat/RPM-based systems:

- Carleton's Netwinder RPM site (armv4l and above)
- My own RPM collection (armv3l and above)
- netwinder.org's experimental RPM area (armv4l and above)

For Debian/.deb-based systems:

- Jim Pick's personal ftp site (armv3l and above)
- Debian's master FTP site (armv3l and above)

I've yet to find anyone supplying binaries for cross compilers.

Note that throughout this document you will see example shell input, like this:

```
% hello --help
```

Please note that these are meant to be a *guide* only -- they are meant to help clarify the text and indicate the correct path to follow; they are by no means meant to be literally and blindly entered at a shell prompt.

Table of contents

Picking a target name

The first thing to decide is precisely which flavour of the tools you wish to build. Here are the basic types:

arm-linux
: This is the most likely target you'll want. This compiles ELF support up for 'Linux/ARM' (i.e. standard ARM Linux). 'ELF' is the best and most recent form for binaries to be compiled in, although most Acorn Linux/ARM users will still be using the old 'a.out' format.

arm-linuxaout
: This produces Linux/ARM flavour, again, but using the 'a.out' binary format, instead of ELF. This is older, but produces binaries which will run on very old ARM Linux installations. *This is now strongly deprecated*; there should be no reason to use this target.

arm-aout, *arm-coff*, *arm-elf*
: These all produce 'flat', or standalone binaries, not tied to any operating system. *arm-elf* is in fact an indepedent ELF port by Cygnus, and is currently less stable than others.

Phil Blundell has this to say on the ELF vs. a.out debate:

> *For a developer, ELF is better because it's a more flexible binary format. It is easy to do things that were difficult with a.out, like shared libraries and binaries with many sections. It's also a more popular format and this means the tools are more widely used and less buggy.*
>
> *For a user, ELF is better purely because it's what all the developers use and so what the software is written for.*
>
> *In the specific case of ARM GNU/Linux, ELF is better for users because it no longer suffers from a 12MB limit on the amount of memory that dynamic-linked programs can use.*

You can fiddle with the *arm* bit of the target name in order to tweak the toolchain you build by replacing it with any of these:

armv2
: This makes support for the ARM v2 architecture, as seen in the older ARM2 and ARM3 processors. Specifically, this forces the use of 26-bit mode code, as this is the only type supported in the v2 architecture.

armv3l, *armv3b*
: This makes support for the ARM v3 architecture, as seen in the ARM610 for the RiscPC. The *l* or *b* suffix indicates whether little-endian or big-endian support is desired (this will almost always be

little-endian).

armv4l, *armv4b*
: This makes support for the ARM v4 architecure, as used in the StrongARM.

Currently, specifying *arm* is equivalent to specifying *armv4l*; i.e. it will cater for the majority of cases (configuring for *arm*/*armv4l* will support *armv3l* as well, it should be noted).

(The target name makes very little difference to binutils; the main importance is in deciding what format *libgcc.a* is (see below for more details). It should also be re-noted that configuring for *arm*/*armv4l* actually builds libgcc to support *armv3l*; therefore not excluding ARM610 or StrongARM RiscPC users.)

Unless you have some specific need (e.g. *armv2*), I strongly suggest you simply use *arm*.

Directory structure

In many of the shell commands listed in this document you'll see italicised and emboldened bits of text. These are, on the whole, directory paths which will change depending on exactly how you've configured your toolchain, meaning I can't put an actual directory path in an example, as it could be different for your setup. You need to substitute in the correct value for your setup yourself in any commands I've listed in the document. Here is a list of these items:

PREFIX
: This is the base directory containing all the other subdirectories and bits of your toolchain; the default for any system is almost always `/usr`, so unless you have a particular desire to put your toolchain in `/usr/local`, or `/usr/arm/tools` or somewhere else (to keep it separate, perhaps), I recommend you use my default, `/usr`.

TARGET-PREFIX
: If you're building a cross-toolchain, this is equal to ***PREFIX***/*target-name* (e.g. `/usr/arm-linux`). If you're building a native compiler, this is simply equal to ***PREFIX***.

binutils

Downloading, unpacking and patching

The first thing you need to build is GNU binutils. Currently the latest recommended version is 2.9.5.0.14. (Please note that there are some significant differences between the 2.9.1 and the 2.9.5 versions on binutils; these are addressed later.)

Download the latest version you can find (or 2.9.5.0.14 if you want to be safe) from any of these sites:

- H. J. Lu's own site -- try here first (US)
- src.doc.ic.ac.uk (UK)
- tsx-11.mit.edu (US)
- sunsite.unc.edu (US)

Unpack the archive somewhere handy, like `/usr/src`:

```
% cd /usr/src
% tar xzf /.../binutils-2.9.5.0.14.tar.gz
```

There may be ARM-specific patches available for binutils which resolve various bugs, or perhaps improve performance; it's usually a good idea to apply these (to the source), if they exist.

Useful places to search for binutils patches are:

- Phil Blundell's Netwinder FTP space

Search these sites (listed in order of preference) for suitable-sounding filenames. For example, currently the latest is `binutils-2.9.1.0.19a-arm-diff-981230.gz` -- you will note that this patch is for the 2.9.1 versions only, and therefore should not be applied to the 2.9.5 versions.

If you can't find a patch file for the version you downloaded, you could either just try building a 'clean' binutils (which way well work perfectly), or you could try applying a patch for an older version of binutils. If you decide not to bother patching binutils, just skip to Configuring and compiling.

You'll probably download a gzipped patch file (denoted by it ending in `.gz`). If so, unzip it:

```
% gunzip binutils-patch.gz
```

You can now patch binutils. Go into your unpacked binutils source directory, and run `patch` with your downloaded patch file, with argument `-p1`:

```
% cd /usr/src/binutils-version
% patch -p1 < binutils-patch
```

You should see a plethora of messages like

```
Hunk #1 succeeded at 112.
```

After patching has finished, you can check whether things have worked by searching for files ending in `.rej` (rejected patches):

```
% find . -name '*.rej'
```

In theory, there should be none of these files; all of the patches should have succeeded. However, if there are one or two failed files, it doesn't necessarily mean it won't build. (Especially if the failed files are documentation (have names like `README`, or end in `.info` or `.texi`), there's probably no reason to worry.)

If you have several of these files, however, it could indicate something seriously wrong, and that it won't build. If you downloaded a patch that wasn't actually for your version (you couldn't find one recent enough), you could either try downloading the version of binutils that corresponds to the latest patch, or just restart with the version of binutils you already have, but not bother with patching it.

Configuring and compiling

You want to essentially follow the instructions provided in the file called `INSTALL`. In practice you'll probably use one of the next two examples.

If you're building a native toolchain, i.e. you're builiding on an ARM machine *for* your ARM machine, try this from inside the *binutils* directory:

```
% ./configure target-name --prefix=PREFIX
```

If you're building on another machine (such as an x86 Linux box), and you want to build a cross-compiler for the ARM, try this:

```
% ./configure --target=target-name --prefix=PREFIX
```

This should succeed (i.e. proceed without stopping with anything that looks like a patent error message), and you can then actually start the build. Invoke `make` in the binutils directory:

```
% make
```

This should proceed without incident, also.

If it works, you can then install your new binutils tools, making sure you've read the overwriting warning below:

```
% make install
```

You'll notice your fab new set of tools in **`PREFIX`**`/target-name/` (or wherever you put them; mine are in `/usr/armv2-linux/`).

Right, we're done with binutils. Now we move on to the compiler.

gcc

Kernel headers

Note that the following headers section may well be redundant if you're building a native compiler, as it will almost certainly have already been done.

(If you *have* a working Linux/ARM kernel installed already, configured for the same sort of machine you're trying to build a toolchain for, simply:

```
% cd TARGET-PREFIX
% mkdir include
% cd include
% ln -s your-linux-kernel-directory/include/asm-arm asm
% ln -s your-linux-kernel-directory/include/linux linux
```

The overwhelming chances are that *your-linux-kernel-directory* will be `/usr/src/linux`. Now skip the rest of this section.)

First off we need to get hold of a current Linux/ARM kernel. Download the latest kernel archive you can

find (currently version 2.2.12):

- src.doc.ic.ac.uk (UK)
- ftp.kernel.org (US)

I recommend you use a version 2.2 kernel (i.e. one in the `v2.2` directory), as development on version 2.0 has ceased. If you're feeling adventurous, you could even try a version 2.3 (i.e. unstable) kernel.

Unpack this somewhere, although preferably *not* in `/usr/src`: if you're on a Linux system, the chances are you'll trash whatever Linux kernel source you already have installed on your system.

There are a wide variety of patches you can apply to Linux kernel source for ARM. Unlike the the binutils patches, applying the kernel patches tends to be mandatory. The two basic patches I recommend are:

- the latest patch you can find on ftp.arm.linux.org.uk for your version of the kernel (currently `patch-2.2.12-rmk1.gz` for version 2.2);
- the latest patch you can find in Phil Blundell's directory for your version of the kernel (currently `linux-2.2.12rmk1-cvs990915.diff.gz` for version 2.2).

You may possibly require patches for specific hardware (e.g. CATS), but this is unlikely for here: we are only trying to get the kernel headers into a state where they can be used to compile gcc; we don't have to worry about device driver support and so forth.

Apply these two patches as shown before, in sequence (assuming you want to use both of them). I believe using the latter patch is mandatory if you want to use a version 2.9.5 binutils.

You need to now tweak the make file, to ensure the configure scripts select the ARM portion of the kernel. Load up the file `Makefile` in the top-level kernel source directory into your favourite text editor, and find the line that looks like this:

```
ARCH := $(shell uname -m | sed -e s/i.86/i386/ -e s/sun4u/sparc64/ -e s/arm.*/
```

Delete it, or comment it out, and insert this:

```
ARCH = arm
```

Now you have to configure the kernel, even though you won't necessarily want to compile from it. Fire up `make menuconfig` in the top-level kernel source directory:

```
% make menuconfig
```

Go into the top option: 'System and processor type', and select a system consistent with the tools you're building. For example, if you're building a set for *armv2-linux*, select 'A5000'.

Exit the configuration program, tell it to save the changes, and then run:

```
% make dep
```

This now ensures your kernel headers are in tip-top condition for the toolchain.

Having patched up your kernel appropriately, you can now copy the headers across into your new toolchain's directory:

```
% mkdir TARGET-PREFIX/include
% cp -dR your-linux-kernel-directory/include/asm-arm TARGET-PREFIX/include/asm
% cp -dR your-linux-kernel-directory/include/linux TARGET-PREFIX/include/linux
```

Now gcc will have its headers, and compile happily.

Downloading, unpacking and patching gcc

Previously egcs was the recommended compiler for use on ARM targets, however as of July 1999, egcs has become the official gcc -- now gcc 2.95 is recommended.

Download the latest version (currently 2.95.1) from any of these sites:

- src.doc.ic.ac.uk (UK)
- egcs.cygnus.com (US)

I suggest you grab `gcc-core-2.95.1.tar.bz2` (in the `gcc-2.95.1` directory on the egcs site), or even `gcc-2.95.1.tar.bz2` if you're feeling masochistic (that is, want the whole thing).

Following through the same instructions as above, unpack your downloaded gcc. Then you may choose to apply patches if they exist; you'll find them in these sites:

- Phil Blundell's Netwinder FTP space (US)

The latest current patch is `gcc-2.95-diff-990730.gz` -- note that patches to gcc tend to be rather more mandatory than binutils patches; you probably should try quite hard to find some you can apply.

(Extra caveat: if you're using gcc 2.95 or 2.95.1, then you should download and apply this patch to your tree as well.)

`gunzip` your downloaded patches if necessary, and patch the gcc source tree directory in the same way you patched binutils above.

Configuring and compiling

You can now configure it in a similar way (reading `INSTALL` as you go). I will reiterate an important point: most people will have *arm-linux*, or *armv4l-linux* as the target name they're configuring for. The configure scripts detect this, and they actually build gcc as though for a target name of *armv3l-linux* -- i.e. for an ARM v3 processor that *doesn't* include half-word data-handling instructions. This means that the average person who compiles up an *arm-linux* toolchain will in fact get an *armv3l-linux* toolchain, and thus one that can safely generate code that will run on all Acorn RiscPCs.

However, you may actually wish to build a genuine ARM v4 toolchain. The easiest way to do this is to add an extra option on the end of your `configure` line: `--with-cpu=strongarm110`.

Configuring done, now we can build the C compiler portion.

This is probably the trickiest stage to get right; there are several caveats to the next step. There are several factors to consider:

- Do you have a fully-working and installed version of glibc *for the same ABI* (i.e. selection of binary format, processor type, etc.) as that for which you are building gcc? If this is your first time building a cross compiler, then the answer is almost certainly no. If this is not your first time building, *and* you built glibc previously, in the same format as you're using for gcc now, then the answer might be yes.

  If the answer is no, then you cannot build support for any language than C, because all the other front-ends depend on libc (i.e. the final gcc binary would expect to link with libc) -- if this is your category, then you must append `LANGUAGES="c"` to each `make` command line inside gcc.

- Do you even have the libc headers for your target? If this is the very first time you have built a cross-compiler on your host, then the chances are that the answer is no; however, if you have previously successfully completed a compilation of a cross-compiling gcc, and installed it in a location that can be found this time round, the answer is probably yes.

  If the answer is no, you will probably need to employ the `-Dinhibit_libc` hack; however, it's worth attempting a build first to see whether you're affected or not. See later for information that will allow you to recognise this problem from the output of `make`.

- Are you trying to build with a version 2.9.5 binutils? If so, load up the file `gcc/config/arm/linux-elf.h`, and change the line that reads `#if 0` (near the top) to `#if 1`.

- Are you building version 2.95 or 2.95.1 of gcc? If so, did you remember to apply the `fold-const.c` patch mentioned above?

Invoke `make` inside the top-level gcc directory, with your chosen parameters:

```
% make [LANGUAGES="c"]
```

The most common error looks like this:

```
./libgcc2.c:41: stdlib.h: No such file or directory
./libgcc2.c:42: unistd.h: No such file or directory
make[3]: *** [libgcc2.a] Error 1
```

and is common with first time building of cross compilers (see above). You can fix it, this time, using the `-Dinhibit_libc` hack -- follow through the instructions in the hack, and then restart at the configure stage.

(This is the most common place that errors seem to occur; if you can't fix them by reading this guide through, then see the bug reporting info.)

Assuming that worked, you can now install your spiffy new compiler:

```
% make [LANGUAGES="c"] install
```

If this worked, congratulations. You (probably) have a compilation environment capable of compiling kernel-mode software, such as the Linux kernel itself, and if you're only after cross-compiling kernels, feel free to stop here. If you want the ability to compile user-space binaries, press on.

glibc

Downloading and unpacking

glibc is split into bits; the *linuxthreads* code comes in a separate archive, and the crypto stuff is only available outside the US (see the USA's particularly enlightened cryptography export laws).

Fetch the main glibc archive (currently `glibc-2.1.2pre3.tar.gz`) and the corresponding *linuxthreads* archive from one of:

- sourceware.cygnus.com (US)
- ftp.funet.fi (Finland)

Now fetch a corresponding (if possible) `glibc-crypt` archive; try the Finland site above.

Unpack the main glibc archive somewhere handy like `/usr/src`. Then unpack the two add-on archives *inside* the directory created when you unpacked the main glibc archive. All set.

Configuring and compiling

This is slightly more complicated than previously. The most important point is that before doing *any* configuring or compiling, you must set the C compiler that you're using to your cross compiler, otherwise glibc will compile as a horrible mix of ARM code and native code. Do this in the same shell you're going to compile in:

```
% CC=target-name-gcc
```

Now we can configure glibc. Go into the top-level glibc directory, and you'll probably want to run configure more or less like this:

```
% ./configure arm-target-name --build=native-target --prefix=TARGET-PREFIX --e
```

An explanation of all the variables? *arm-target-name* is important: at present the glibc configuration scripts don't recognise the various mutations of the *arm-* bit of the target name, so here you have to specify your normal target name, but *changing the first arm- bit back* to simply *arm*, rather than, say, *armv3l*, or whatever.

native-target is the target name of the machine you're building on; for instance on an x86 Linux machine, `i586-linux` would probably do nicely.

You'll notice the prefix is different this time: not just ***PREFIX*** (or whatever you're using), but with the target name component on the end as well. *Don't*, whatever you do, forget to specify this last component, or you may hose your local libraries, and thus screw up your system.

Okay, go ahead and configure, reading `INSTALL` if you want to check out all the options. Assuming that worked, just run:

```
% make
% make install
```

And if *those* worked, you're sorted. You have a full ARM toolchain and library kit available for use on your system.

You can run the glibc or gcc test suites, and also go back to gcc, and compile up all the parts of the compiler you couldn't before (i.e. you can leave off `LANGUAGES="c"` now).

You can now try compiling things by using *`target-name`*`-gcc` as your compiler; just pull the same `CC=arm...` trick as shown above before compiling any given package, and it should work.

I'd much like to receive notification of any success/failure with toolchain-building, and in particular this guide, so please drop me a line!

Appendices

libgcc

A short lesson on gcc's hierarchy. Whatever target name you build gcc for, the *main* code engine still contains support for all the different ARM variations (i.e. it's the same whatever target name you build with). *However*, there is a library accompanying gcc, containing some fundamental support routines and suchlike, called `libgcc.a`. This is the thing that will differ between different target names, and this is what makes different toolchains binary incompatible. Note that exactly the same incompatibilities apply to glibc, as well.

Overwriting an existing toolchain

If you're building a native compiler, you must be aware that it is extremely easy to trash your toolchain half-way through building. The most common cause of this is trying to build a native set of ELF tools on a system where gcc was built to produce a.out code (e.g. most Linux/ARM systems running on Acorn hardware). The crucial breaking point is the `make install` command to install binutils. In the example scenario I've just described, this will leave you unable to link any programs, as gcc's libraries will be in a.out format, but all your binutils will be unable to understand anything but ELF.

So, how does one get round this? An example solution is to initially build everything with a ***PREFIX*** of something like `/usr/local/arm-tmp`, so as not to interfere with the existing toolchain. Then, go back and compile everything again, but using your proper prefix (e.g. `/usr`), but making sure `/usr/local/arm-tmp` (or whatever you used) is on your `$PATH` environment variable. Then, having built everything in the correct directory, swipe `/usr/local/arm-tmp`.

If you *do* manage to trash your toolchain, go and fetch the binutils and gcc RPMs (e.g. from ftp.arm.linux.org.uk) and reinstall them, using `rpm --replacepkgs`.

Issues with binutils 2.9.5

There are some significant differences between 2.9.1 and 2.9.5 versions of binutils; the following is an extract from a post from Phil Blundell describing a little about them, for the curious:

> *This is the one issue that makes me reluctant to tell everyone to throw away binutils 2.9.1 and use 2.9.5 instead. Basically, owing to a series of fairly uninteresting slip-ups, we have ended up in a situation where binutils 2.9.5 is not 100% compatible with the older versions. Normal programs shouldn't care but gcc and glibc need to know the difference.*
>
> *I hope it will be fixed in glibc before 2.1.2 is actually released, though I can't make any promise.*
>
> *As a side note, I am about to change the mainline egcs sources so that a target name of 'arm-*-linux-gnu' implies binutils 2.9.5, and a target name of 'arm-*-linux-gnuoldld' implies an older version. Config.guess can tell the difference so there should be no reason for concern unless you set target names by hand. This will probably end up being backported to my 2.9.5 patchset sooner or later.*

The `-Dinhibit_libc` hack

Upon installing a successful build of gcc, some headers will get put in the target's `include` directory. However, if you are building a (cross) compiler for the very first time, or with a different set of paths, it won't have these headers to hand. For the first time you build a gcc, then, you can follow through these steps to fix the problem:

- Edit `gcc/config/arm/t-linux`, and add `-Dinhibit_libc` and `-D__gthr_posix_h` to `TARGET_LIBGCC2_CFLAGS`. That is, change the line that looks like:

  ```
  TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC
  ```

  to:

  ```
  TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC -Dinhibit_libc -D__gthr_pos
  ```

- Re-run `configure`, but supplying the extra parameter `--disable-threads`.

Links

- Most of the information on this page was derived from the original guide by Philip Blundell. Phil is responsible for most of the ARM toolchain, and you can find his web site, with a bunch of other goodies on at <http://www.tazenda.demon.co.uk/phil/>. Phil is much more on the ball than me -- his web site is a good place to check for information that isn't chronically out-of-date, like mine.
- The crossgcc FAQ is an excellent source of information on building cross compilers, and should be read.
- Someone else has had a go at a similar guide: <http://ssdl-redstone.stanford.edu/~zskiraly/salinux/>. (Recent: this seems to be down?)
- Chris Sawer has a useful site with kernel-compiling hints: <http://members.xoom.com/chrissawer/armlinux.html>.
- The central ARM Linux pages often have useful information on them, and are a useful starting

point.

Bugs

If you find any bugs in this document, or have any questions or things to contribute, don't hesitate to contact me. If you find you try to follow through the document and something doesn't work (this won't be an uncommon state of affairs, until the ARM stuff settles down a bit), I suggest you try posting to the *linux-arm* mailing-list. You'll find subscription details for this on Russell's site. I tend to read the list regularly, so you'll get a response from me if it's appropriate.

(Last modified: $Date: 1999/10/03 10:58:53 $)
Chris Rutter <chris@fluff.org>