This chapter contains information on building a GNU toolchain for ARM targets.

The GNU Toolchain for ARM targets HOWTO

Wookey

Chris Rutter

Jeff Sutherland

Paul Webb

This document contains information on setting up a GNU toolchain for ARM targets. It details both some pre-compiled toolchains which you can install, and how to compile your own toolchain, as either a native or a cross-compiler.

1. Credits

This document is based on the work of Chris Rutter (now, sadly, deceased) who's 'Building the GNU toolchain for ARM targets' document was gospel for some time. It eventually became out of date so Wookey (<`wookey@armlinux.org`>) updated it and gave it a substantion rewrite, adding the pre-built toolchain info at the same time. Paul Webb (<`paul.webb@argonet.co.uk`>) did the initial conversion to DocBook, Phil

Blundell (<`philb@gnu.org`>) provided info on the current state of the art and comments on the draft. Jeff Sutherland (<`jeffs@accelent.com`>) then fixed the bits that were still wrong, and now maintains the doc, along with Wookey. Thanx to all.

As well as being on-line as a stand-alone HOWTO, this document is also available as a chapter of the book: A ’Guide to ARMLinux for Developers’ (http://www.aleph1.co.uk/armlinux/thebook.html)

This chapter contains information on building a GNU toolchain for ARM targets.

1. Toolchain overview

The toolchain actually consists of a number of components. The main one is the compiler itself gcc, which can be native to the host or a cross-compiler. This is supported by *binutils*, a set of tools for manipulating binaries. These components are all you need for compiling the kernel, but almost anything else you compile also needs the C-library glibc. As you will realise if you think about it for a moment, compiling the compiler poses a bootstrapping problem, which is the main reason why generating a toolset is not a simple exercise.

This chapter contains information on how the toolchain fits together and how to build it. However, for most developers it is not necessary to actually do this. Building the toolchain is not a trivial exercise and for most common situations pre-built toolchains already exist. Unless you need to build your own because you have an unusual situation not catered for by the pre-built ones, or you want to do it anyway to gain a deeper understanding, then we strongly recommend simply installing and using a suitable ready-made toolchain.

2. Pre-built Toolchains

This CD contains three pre-built toolchains, one from emdebian.org, one from the LART project and a third from compaq's handhelds.org team. The emdebian chain is newest, and we've had good sucess with it, but all are used by various people. They all have very similar functionality.

At the moment the most likely situation where a pre-built toolchain will not do the job is if you need Thumb support, in which case you need to use gcc v3 (not yet released at the time of writing, but available as snapshots).

2.1. Native Pre-built Compilers

For binary versions of native compilers (ie ones that run on ARM and compile for ARM), the current stable release is on the Aleph ARMLinux CD. You can also get them from:

2.1.1. Resources

- The current stable release on Debian's master FTP site (armv3l and above) (ftp://ftp.debian.org/debian/dists/stable/main/binary-arm/devel/).
- The latest release on Debian's master FTP site (armv3l and above) (ftp://ftp.debian.org/debian/dists/unstable/main/binary-arm/devel/).

Sometimes ARMLinux.org will have experimental versions available.

2.2. Emdebian

The emdebian version includes gcc version 2.95.2, binutils 2.9.5.0.37, and glibc 2.1.3. Installation is simple. If you have a Debian system then the emdebian cross-compiler is incredibly simple to install - just show apt the directory on the CD and do **apt-get install task-cross-arm**. What could be simpler?

Warning

The emdebian cross development environment will install files in `/usr/bin` so you will have to make sure that you do not overwrite any development tools which you may already have on your system.

2.2.1. Installing the Toolchain

task-cross-arm is a *virtual* package which includes:

1. gcc version 2.95.2;
2. binutils 2.9.5.0.37;
3. glibc 2.1.3.

This is made up of the following packages:

1. the C preprocessor: cpp-arm_2.95.2-12e4_i386.deb;
2. the C compiler: gcc-arm_2.95.2-12e4_i386.deb;
3. the C++ compiler: g++-arm_2.95.2-12e4_i386.deb;
4. gnu C library: libc6-dev-arm_2.1.3-8e4_i386.deb;
5. C++ library: libstdc++2.10-arm_2.95.2-12e4_i386.deb;
6. C++ library and headers: libstdc++2.10-dev-arm_2.95.2-12e4_i386.deb;
7. Binary utilities: binutils-arm_2.9.5.0.37-1e3_i386.deb.

They are available in both deb and RPM form.

In order to set up your cross development environment on a Debian system, proceed as follows:

- **su** to root by typing **su** at the prompt;
- add the line

```
deb http://www.emdebian.org/emdebian unstable main
```

to your `/etc/apt/sources.list` file;

- type **apt-get update** to tell apt/dselect to note the new packages available ;
- enter **apt-get install task-cross-arm** to install your new development environment;
- type **exit** to become a normal user and begin to cross-compile.

For the RPM form download it and use:

rpm -i g++-arm-1%3a2.95..2-12e4.i386.rpm

2.3. LART

The LART tarball contains:

1. gcc 2.95.2;
2. binutils 2.9.5.0.22;
3. glibc 2.1.2.

2.3.1. Installing the Toolchain

In order to install the LART tarball on your system, do the following:

```
mkdir /data
mkdir /data/lart
cd /data/lart
bzip2 -dc arm-linux-cross.tar.bz2 | tar xvf -
```

You can then add `/data/lart/cross/bin` to your path. The C and C++ compilers can then be invoked with **arm-linux-gcc** and **arm-linux-g++** respectively.

2.4. Compaq

The Compaq arm-linux cross toolchain includes:

1. gcc-2.95.2;
2. binutils-2.9.5.0.22;

3. glibc-2.1.2 with the international crypt library.

The toolchain is compiled with a i386 host with an armv41 target.

2.4.1. Installing the Toolchain

> **Note:** The toolchain must be installed in `/skiff/local` as it will not work from any other path.

The only other problem that you may have with the include files is that the tarball was set up for Linux 2.2.14. You may consequently need to set up symbolic links:

ln `-s` /usr/src/linux/include/asm
/skiff/local/arm-linux/include/asm
ln -s /usr/src/linux/include/linux
/skiff/local/arm-linux/include/linux

Alternatively, copy `/usr/src/linux/include/asm` and `/usr/src/linux/include/linux` to `/skiff/local/arm-linux/include` before running **make menuconfig** and **make dep**. This will verify that your kernel tree and correct symbolic links are up to date.

> **Note:** This toolchain has glibc symbol versioning. If you are using a NetWinder, you may have to compile your code with static libraries.

3. Building the Toolchain

In outline what you need to do is:

- Decide on the target name;
- Decide where to put the images;
- Work out headers policy;
- Compile binutils first;
- Then compile gcc;
- Produce gLibc last.

3.1. Picking a target name

A native compiler is one that compiles instructions for the same sort of processor as the one it is running on. A cross-compiler is one that runs on one type of processor, but compiles instructions for another. For ARM embedded development it is common to have a compiler that runs on an x86 PC but generates code for the target ARM device.

What type of compiler you build and the sort of output it produces is controlled by the ’target name’. This name is what you put in instead of ’TARGET-NAME’ in many of the examples in this chapter. Here are the basic types:

arm-linux

This is the most likely target you’ll want. This compiles ELF support for *Linux/ARM* (i.e. standard ARMLinux). ELF is the best and most recent form for binaries to be compiled in, although early Acorn Linux/ARM users may 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 ARMLinux installations. This is now strongly deprecated; there should be no reason to use this target; note that binutils 2.9.5 doesn't contain support for it (refer to Section 3.6.3).

arm-aout, arm-coff, arm-elf, arm-thumb

These all produce *flat*, or standalone binaries, not tied to any operating system. arm-elf selects Cygnus' ELF support, which shares much of its code with arm-linux.

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, and ARM710. 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 architecture, as used in the StrongARM, ARM7TDMI, ARM8, ARM9.

armv5l, armv5b

This makes support for the ARM v5 architecture, as used in the XScale and ARM10.

In practice the target name makes almost no practical difference to the toolchain you get anyway so you should always use plain 'arm'. This means that the toolchain itself is not unhelpfully tied to the type of processor it was built on. You get a toolchain that will run on all ARM processors and you can set the compiler output for the target processor when you build each part of the toolchain.

3.2. Choosing a 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. This means that we have not used an actual directory path in examples as it could be different from your setup. You need to substitute the correct value for your setup yourself for any commands that we have listed in this 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 the native toolchain on any system is almost always `/usr`. To keep from stepping on your system's native tools when you build a cross-compiler you should put your cross-development toolchain in `/usr/local`, or `/usr/arm/tools` or somewhere else that makes sense for you, in order to keep it separate and easy to maintain.

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.

KERNEL-SOURCE-LOCATION

> This is the place where your kernel source (or at least headers) are stored. Especially if you are cross compiling this may well be different to the native set of files. We recommend that you set this to TARGET-PREFIX/linux as a sensible default.

3.3. Binutils

Binutils is a collection of utilities, for doing things with binary files.

3.3.1. Binutils components

addr2line

Translates program addresses into file names and line numbers. Given an address and an executable, it uses the debugging information in the executable to figure out which file name and line number are associated with a given address.

ar

The GNU **ar** program creates, modifies, and extracts from archives. An archive is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called members of the archive).

as

GNU **as** is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called pseudo-ops) and assembler syntax.

as is primarily intended to assemble the output of the GNU C compiler gcc for use by the linker ld. Nevertheless, we've tried to make **as** assemble correctly everything that the native assembler would. This doesn't mean **as** always uses the same syntax as another assembler for the same architecture.

c++filt

The **c++filt** program does the inverse mapping: it decodes (demangles) low-level names into user-level names so that the linker can keep these overloaded functions from clashing.

gasp

Gnu Assembler Macro Preprocessor.

ld

The GNU linker **ld** combines a number of object and archive files, relocates their data and ties up symbol references. Often the last step in building a new compiled program to run is a call to ld.

nm

GNU **nm** lists the symbols from object files.

objcopy

The GNU **objcopy** utility copies the contents of an object file to another. **objcopy** uses the GNU BFD library to read and write the object files. It can write the destination object file in a format different from that of the source object file. The exact behavior of **objcopy** is controlled by command-line options.

objdump

objdump displays information about one or more object files. The options control what particular information to display.

ranlib

ranlib generates an index to the contents of an archive, and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file. You may use ‘nm -s’ or ‘nm --print-armap’ to list this index.

readelf

readelf Interprets headers on elf files.

size

The GNU **size** utility lists the section sizes and the total size for each of the object files objfile in its argument list. By default, one line of output is generated for each object file or each module in an archive.

strings

GNU **strings** prints the printable character sequences that are at least 4 characters long (or the number given with the options below) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.

strip

GNU **strip** discards all symbols from the target object file(s). The list of object files may include archives. At least one object file must be given. **strip** /modifies the files named in its argument, rather than writing modified copies under different names.

3.3.2. Downloading, unpacking and patching

The first thing you need to build is GNU binutils. 2.9.5 versions have proved stable but generally the latest release is recommended (2.10.1 at the time of writing). No-one should be using 2.9.1 anymore.

Download the latest version you can find from any of these sites:

- ftp://ftp.gnu.org/gnu/binutils/" - the official release site (US); (ftp://ftp.gnu.org/gnu/binutils/)
- H. J. Lu's own site -- this has the very latest stuff (US); (ftp://ftp.varesearch.com/pub/support/hjl/binutils)
- src.doc.ic.ac.uk (UK). (ftp://src.doc.ic.ac.uk/Mirrors/sunsite.unc.edu/pub/Linux/GCC)

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

```
cd /usr/src
```

tar -xzf /.../binutils-2.10.1.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. However ARM binutils are now quite stable (at the time of writing) and integrated with the main tree, so extra patches are no longer normally required. The best place to get up to date information is the armlinux-toolchain mailing list.

The place to check if there are any currently recommended patches is here:

- ftp://ftp.armlinux.org/pub/toolchain (UK). (ftp://ftp.armlinux.org/pub/toolchain)

3.3.3. Configuring and compiling

Essentially, you want to follow the instructions provided in the file called `INSTALL`. In practice you'll probably use one of the followingexamples.

If you're building a native toolchain, i.e. you're building on an ARM machine for an ARM machine, then you should just do this from inside the binutils directory:

./configure --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 e.g. **./configure --target=arm-linux --prefix=/usr/arm_tools**

This should succeed (i.e. proceed without stopping with anything that looks like an obvious error message), and you can then actually start the build.

Invoke **make** in the binutils directory:

make

This should proceed without incident.

If it works, you can then install your new binutils tools, making sure you've read the overwriting warning below (refer to Section 3.6.2):

make install

You'll notice your new set of tools in PREFIX/TARGET-NAME/. One likely location may be in `/usr/arm_tools/arm-linux/`.

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

3.4. gcc

3.4.1. Kernel headers

> **Note:** When building a native compiler, most likely a set of kernel headers for your platform will be available and you don't need to be concerned with headers. For cross-compiling, a set of kernel headers from a source tree configured for your target must be available.

The overwhelming chances are that KERNEL-SOURCE-LOCATION for a native compiler build will be `/usr/src/linux`. Now skip the rest of this section.

However if you are compiling for a different type of ARM machine, or compiling a different version of the kernel or cross-compiling, then you need a different set of headers. First off we need to get hold of a current Linux/ARM kernel. Download the latest kernel archive you can find (version 2.4.1 at the time of writing):

- ftp.uk.kernel.org (UK) (ftp://ftp.uk.kernel.org/pub/linux/kernel/)
- ftp.kernel.org (US) (ftp://ftp.kernel.org/pub/linux/kernel)

We recommend you use a version 2.4 kernel (i.e. one in the v2.4 directory). There are several reasons for this; many newer ARM architectures are only really properly supported in kernel 2.4 and development on version 2.0 has ceased, whilst 2.2 is now in maintainence mode. However, 2.2 kernels are significantly smaller, so if it has the functionality you need it may make sense to use one, but you will be running against the flow to an increasing extent.

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. We suggest /usr/PREFIX/ (e.g. /usr/arm-linux/).

There are a wide variety of patches you can apply to Linux kernel source for ARM. Applying the kernel patches tends to be mandatory. The two basic patches we recommend are:

- the latest patch you can find on ftp.arm.linux.org.uk (ftp://ftp.arm.linux.org.uk/pub/armlinux/source/kernel-patches) for your version of the kernel (currently `patch-2.4.1-rmk1.gz for version 2.4`);
- the latest patch you can find in Nicolas Pitre's StrongARM patches (ftp://ftp.netwinder.org/users/n/nico/) for your version of the kernel (currently `diff-2.4.1-rmk1-np2.gz` for version 2.4).

You may possibly require patches for specific hardware (e.g. iPAQ, Psion), 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 in sequence (assuming you want to use both of them).

Now you may need to tweak the make file, to ensure that 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.*/arm/ -e s/sa110/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 arm-linux with a StrongARM that doesn’t need to support arm v3, select ’LART’. In practice, the toolchain only takes a couple of items from the kernel headers so it doesn’t actually matter much what you select, so long as you select something and the correct links are made so that the files needed are visible in the right places.

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

make dep

This command actually makes the links (linking /linux/include/asm/ to /linux/include/asm-arm etc) and ensures your kernel headers are in tip-top condition for the toolchain.

Having patched up your kernel appropriately, you are ready to go if you are building a cross-development tool chain. If you are doing a native toolchain build, however, you will have to copy the headers across into your new toolchain’s directory:

mkdir TARGET-PREFIX/include

cp -dR KERNEL-SOURCE-LOCATION/include/asm-arm TARGET-PREFIX/include/asm

cp -dR LINUX-SOURCE-LOCATION/include/linux TARGET-PREFIX/include/linux

Now gcc will have its headers, and compile happily.

3.4.2. Downloading, unpacking and patching gcc

Download the latest version (2.95.3-prerelease at the time of writing), unless you need thumb support or are feeling brave, in which case you can try a CVS snapshot of the forthcoming v3.0 from any of these sites:

- gcc.gnu.org (US) (ftp:///gcc.gnu.org/)

- sourceware.cygnus.com Mirror (UK) (ftp://sunsite.doc.ic.ac.uk/Mirrors/sourceware.cygnus.com/pub/gcc/)

We suggest you grab `gcc-core-2.95.3.tar.bz2` (8MB) (in the `gcc-2.95.3` directory on the gcc site), or even `gcc-2.95.3.tar.bz2` (12MB) if you're feeling masochistic and 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; As of gcc 2.95.2 patches for ARM are not generally required. Check out the armlinux-toolchain list archives for the current state of play or look here to see if there are any current patches for your version:

- ftp://ftp.armlinux.org/pub/toolchain (UK). (ftp://ftp.armlinux.org/pub/toolchain)

3.4.3. Configuring and compiling

You can now configure gcc in a similar way that you did for binutils (reading `INSTALL` as you go).

Most people will have arm-linux as the target name they're configuring for, which builds a toolchain suitable for running on any ARM. For a native compiler do this:

./configure --prefix=PREFIX

For a cross-compiler, do this::

./configure --target=TARGET-NAME --prefix=PREFIX --with-headers=LINUX-SOURCE-LOCATION/include

e.g. **./configure --target=arm-linux --prefix=/usr --with-headers=/usr/src/linux/include**

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

This is probably the trickiest stage to get right; There are several factors to consider:

- Do you have a fully-working and installed version of glibc *for the same ABI* as that for which you are building gcc? (i.e is your existing glibc for the same processor-type and binary format etc). 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 other than C, because all the other front-ends depend on libc (i.e. the final gcc binary would expect to link with libc), so if this is your first build, or you changed to a different target, then you must add the switches **--enable-languages=c --disable-threads** to the gcc configurations listed above.

- 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 (refer to Section 3.6.4); however, it's worth attempting a build first to see whether you're affected or not. (Most likely you will be if this is your first cross-compile attempt.)

Invoke make inside the top-level gcc directory, with your chosen parameters. 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 (refer to Section 3.6.4) -- follow through the instructions in the hack, and then restart at the configure stage.

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

make install

If the make terminated normally, congratulations. You (probably) have a compilation environment capable of compiling kernel-mode software, such as the Linux kernel itself. Note that when building a cross-compiler you will probably see error messages in the transcript saying that the compiler that was built doesn't work. This is because the test that's performed creates a small executable in the target, not the host format, and as a result will fail to run and generate the error. 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.

3.5. glibc

glibc is the C library. Almost all userland applications will link to this library. Only things like the kernel, bootloaders and other things that avoid using any C library functions can be compiled without it. There are alternatives to glibc for small and embedded systems (it's a big library) but this is the standard and it's what you should use for compiling the rest of gcc.

3.5.1. Downloading and unpacking

glibc is split into bits (called add-ons):

- the *linuxthreads* code which is in a separate archive;
- the crypto stuff (which used to be an add-on) is now included in the latest release (glibc-2.2.2).

Fetch the main glibc archive (currently `glibc-2.2.2.tar.gz`) and the corresponding linuxthreads archive from one of the following:

- ftp.gnu.org/gnu/glibc (US) (ftp://ftp.gnu.org/gnu/glibc);
- ftp.funet.fi (Finland) (ftp://ftp.funet.fi/pub/gnu/funet);

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.

3.5.2. Configuring and compiling

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

CC=TARGET-NAME-gcc

Be sure to add the path to **TARGET-NAME-gcc** to your PATH environment variable as well. Create a new directory next to the top level source directory for gcc. Go into this directory, and configure and build glibc here. It is a very bad idea to configure in the glibc source directory (see the README file for further warnings). We won't detail the reasons here. 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:

../glibc-2.2.2/configure arm-TARGET-NAME --build=NATIVE-TARGET --prefix=TARGET-PREFIX --enable-add-ons

So what do all the variables mean? 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.

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, but with the target name component on the end as well.

Warning

Don’t 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 - however it can only compile C programs. To be able to compile anything else, you need to do a bit more work.

Go back and re-configure gcc but this time either add the languages you want (e.g. **--enable-languages=c,c++** or else leave off the switch altogether to build everything. You must also remove the "Dinhibit_libc" hack if you had to apply it before. WARNING: be sure to unset the ’CC’ environment variable when cross-compiling so the native compiler will be used to finish building your cross-development tool chain.

You can now try compiling things by using TARGET-NAME-gcc (e.g arm-linux-gcc) as your compiler; just set CC as above (e.g CC=arm-linux-gcc) before compiling any given package, and it should work. For any package using configure, you can normally set CC in the makefile, rather than as a local system variable. Setting the local system variable can be a problem if later on you need to compile something natively and you forget to unset it. Be sure that a path to all of the toolchain binaries exists in your PATH environment variable.

3.6. Notes

3.6.1. libgcc

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, called `libgcc.a`. This is the thing that will differ between different target names, and this is what makes different toolchains binary incompatible.

> **Note:** Exactly the same incompatibilities apply to glibc as well.

3.6.2. Overwriting an existing toolchain

If you're building a native compiler, with a significantly different target from the current one, 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. older Linux/ARM systems running on Acorn hardware). This is no longer a significant issue, but the example remains valid. The crucial breaking point is the **make install** command which installs binutils. In the example scenario, 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? A 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, you will need to go and fetch a suitable toolchain for your existing installation.

3.6.3. Issues with older version of binutils and gcc

There are some significant differences between binutils 2.9.1 and 2.9.5 and between gcc 2.95.1 and 2.95.2. If you use an old binutils with a new compiler or vice versa then things will go wrong. The indications of this areas follows: the error ’unrecognised emulation: armelf_linux’ means your toolchain is too old for your compiler. Conversely ’unrecognised emulation: elf32arm’ means your compiler is too old for your toolchain.

3.6.4. 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_posix_h`.

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

4. Links

Other useful sources of information include:

- Phil Blundell's web site (http://www.tazenda.demon.co.uk/phil/). This web site is a good place to check for information that isn't out of date.
- The crossgcc FAQ is an excellent source of information on building cross-compilers, and should be read.
- Chris Sawer on kernel-compiling hints (http://members.xoom.com/chrissawer/armlinux.html)
- The main ARMLinux pages (http://www.arm.linux.org.uk/) often have useful information on them, and are a good starting point.