Orlando
2005

Freescale
Technology
Forum

Network. Connect. Explore.

Building GNU tool chains for PowerPC target systems

Phil Brownfield
Systems Enablement Manager, CPD

freescale semiconductor

Overview

In this session we'll discuss

Where to obtain current GNU tool chain source code

Options for library source code, and where to obtain it

Building a native GNU tool chain for PowerPC Linux

Building a cross-GNU tool chain for an embedded PowerPC target

The GNU Compiler Collection

The GNU Compiler Collection (GCC) is a project of the Free Software Foundation

The GCC supports C, C++, Fortran, ADA, Java and additional languages, along with necessary supporting software libraries

It supports software development for various native and cross target systems

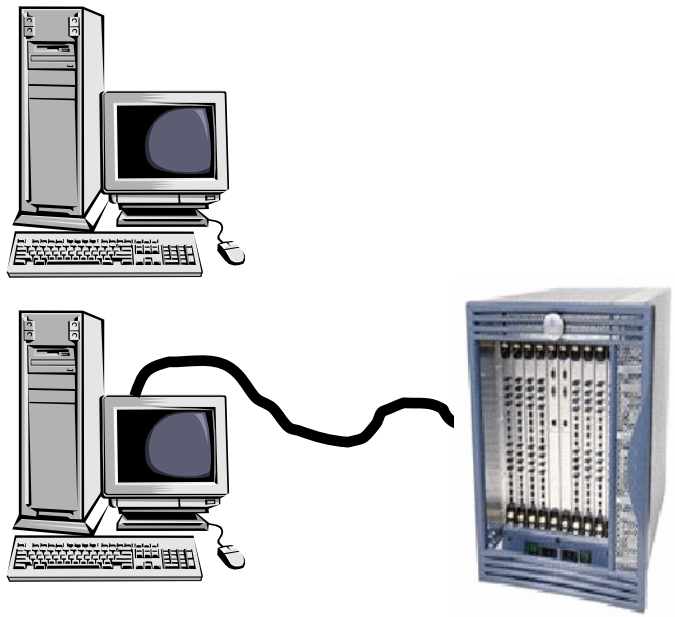
The GCC is "free software" released under the GNU Public License

Native versus cross program development

Native, "self-hosted" development creates programs on the system where they will run

Cross development creates programs on a "host" system, and runs them on a different "target" system

Linux ("GNU/Linux" if you prefer) and *BSD are the only embedded PowerPC OSes supporting native development today



GCC compile flow

Assembler, Linker

- binutils

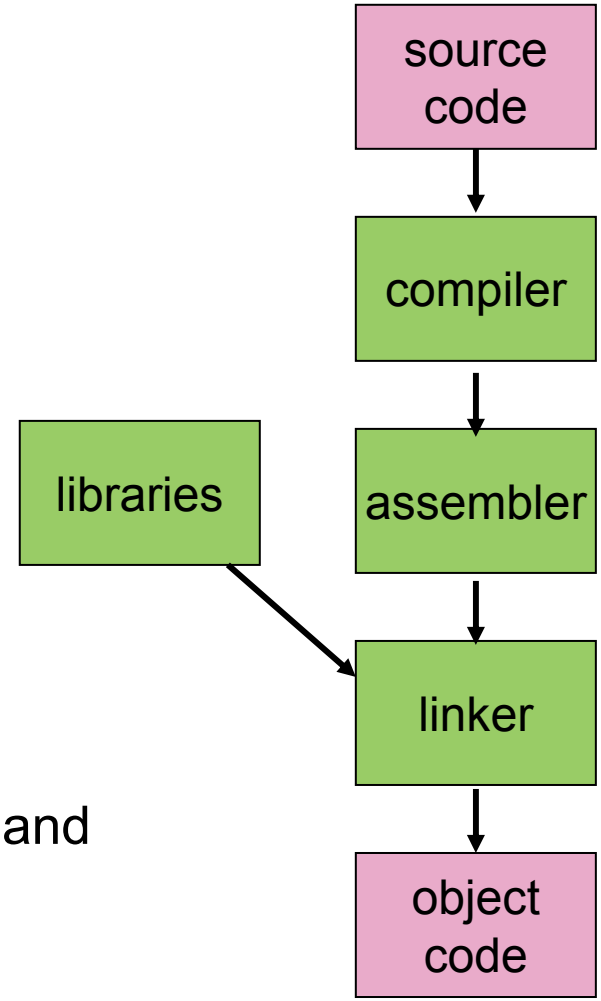
Compiler

- gcc

Runtime Libraries

- glibc
- or newlib

GNU tools support native or cross program development and debugging



Building a native compiler

Almost all UNIX and Linux hosts have native GNU tools available, and under Windows the Cygwin environment enables GNU tools.

But perhaps you'll find you need a feature newer in GCC than versions supported by your vendor. It's possible to install your own:

- Verify host tools prerequisites exist
- Build binutils
- Build compiler
- Build runtime libraries

You can bootstrap up a complete GNU toolset on a system with just an ANSI C90 compiler and a POSIX environment, but it's a much lengthier process that we won't cover

Prerequisites for Building GNU tool chains

Compiler, assembler, linker

- For full bootstrapping to a new machine type, an ANSI C90 compiler
- For other languages/simple cross targets/rebuilds, gcc v2.95 or later
  - > For gcc 3.3 or later, gcc 3.2 or later is needed

A POSIX-compatible shell, eg bash

- For command scripting

GNU make v3.79.1 or later

Tar, gzip and/or bzip2

binutils

A suite of tools for creating and manipulating binary programs

Includes the GNU assembler (as) and GNU linker (ld)

Also includes addr2line, ar, c++filt, gprof, nlmconv, nm, objcopy, objdump, ranlib, readelf, size, strings, strip, windres

Licensed under the GNU Public License (GPL)

Current version is binutils 2.16

Home page: http://sourceware.org/binutils/

Download available via the home page or a via a mirror listed at http://sourceware.org/mirrors.html

gcc

"A full-featured ANSI C compiler with support for K&R C, as well as C++, Objective C, Java, and Fortran"

Licensed under the GNU Public License (GPL)

Most recent versions are gcc 3.3.6, gcc 3.4.4, gcc 4.0.0

Home page: http://gcc.gnu.org/

Download available via the home page or via a mirror listed at http://gcc.gnu.org/mirrors.html or http://sourceware.org/mirrors.html

gcc 4 new features

The GCC 4.0.0 release brings a long list of new features

Of note is the new "Tree SSA" framework for optimizer improvements

- Supports autovectorization for AltiVec
  - Driven by –ftree-vectorize –maltivec flags during application compilation
  - Vectorizes some forms of program loops
- Supports future optimization enhancements

Also of note is support for Fortran 90 and Fortran 95

glibc

A very complete standard library suite for ANSI C

GCC is required to build glibc

Licensed under the Lesser GNU Public License (LGPL)

Current version is glibc 2.3.5

Home page: http://www.gnu.org/software/libc/libc.html

Download available via the home page or a via a mirror listed at http://sourceware.org/mirrors.html

newlib

More memory-conscious than glibc

Contains required C standard library functions, but not all for a full UNIX system

Licensed under various free software licenses, generally BSD or BSD-ish

Current version is newlib 1.13

Home page: http://sourceware.org/newlib

Download available via the home page or a via a mirror listed at http://sourceware.org/mirrors.html

1) Getting started: set up your directories

```
pkb@pegasos2 pkb $ mkdir ~/GNUsrc ~/GNU
pkb@pegasos2 pkb $ cd GNUsrc
pkb@pegasos2 GNUsrc $ tar jxf ../GNUarchive/binutils2.16.tar.bz2
pkb@pegasos2 GNUsrc $ tar jxf ../GNUarchive/gcc3.4.4.tar.bz2
pkb@pegasos2 GNUsrc $ tar jxf ../GNUarchive/glibc-2.3.5.tar.bz2
pkb@pegasos2 GNUsrc $ tar jxf ../GNUarchive/glibc-linuxthreads-2.3.5.tar.bz2
pkb@pegasos2 GNUsrc $ ls
binutils-2.16      gcc-3.4.4 glibc-2.3.5         linuxthreads      linuxthreads.db
pkb@pegasos2 GNUsrc $ mv linuxthreads linuxthreads.db glibc-2.3.5
pkb@pegasos2 GNUsrc $ mkdir build_binutils build_gcc build_glibc
```

Create a working directory and an installation directory

Unpack GCC source tarballs into the working directory

- Merge linuxthread support package into glibc tree

Create a working subdirectory for each package you'll build

- DON'T build the packages in their source directories

2) Building PowerPC Linux native binutils

```
pkb@pegasos2 GNUsrc $ export INSTDIR=/home/pkb/GNU
pkb@pegasos2 GNUsrc $ cd build_binutils
pkb@pegasos2 build_binutils $ ../binutils-2.16/configure --prefix=$INSTDIR \
  --enable-languages=c,c++ >c.log 2>&1
pkb@pegasos2 build_binutils $ make >m.log 2>&1
pkb@pegasos2 build_binutils $ make install >i.log 2>&1
pkb@pegasos2 build_binutils $
```

Building any of these GNU packages follows the same basic three-step process

- configure
- make
- make install

--prefix=$INSTDIR specifies directory where binutils will be installed

There are many, many other parameters available with configure

3) Building PowerPC Linux native gcc

```
pkb@pegasos2 build_binutils $ export PATH=$INSTDIR/bin:$PATH
pkb@pegasos2 build_binutils $ cd ../build_gcc
pkb@pegasos2 build_gcc $ ../gcc-3.4.4/configure --prefix=$INSTDIR --disable-nls \
  --enable-languages=c,c++ --enable-threads=posix --enable-shared --with-system-zlib \
   --disable-checking --enable-cstdio=stdio --enable-__cxa_atexit >c.log 2>&1
pkb@pegasos2 build_gcc $ make >m.log 2>&1
pkb@pegasos2 build_gcc $ make install >i.log 2>&1
pkb@pegasos2 build_gcc $
```

Add $INSTDIR/bin to $PATH so that subsequent steps use the new binutils

--prefix=$INSTDIR specifies directory where binutils will be installed

--enable-languages specifies language front ends (C and C++ in this case)

Again, there are many, many parameters

gcc config parameters

```
pkb@pegasos2 build_gcc $ gcc -v
Reading specs from /usr/lib/gcc/powerpc-unknown-linux-gnu/3.4.1/specs
Configured with: /var/tmp/portage/gcc-3.4.1-r3/work/gcc-3.4.1/configure --prefix=/usr
--bindir=/usr/powerpc-unknown-linux-gnu/gcc-bin/3.4 --includedir=/usr/lib/gcc/powerpc-
unknown-linux-gnu/3.4.1/include --datadir=/usr/share/gcc-data/powerpc-unknown-linux-gnu/3.4
--mandir=/usr/share/gcc-data/powerpc-unknown-linux-gnu/3.4/man --infodir=/usr/share/gcc-
data/powerpc-unknown-linux-gnu/3.4/info --enable-shared --host=powerpc-unknown-linux-gnu --
target=powerpc-unknown-linux-gnu --with-system-zlib --enable-languages=c,c++,f77 --enable-
threads=posix --enable-long-long --disable-checking --disable-libunwind-exceptions --enable-
cstdio=stdio --enable-version-specific-runtime-libs --with-gxx-include-
dir=/usr/lib/gcc/powerpc-unknown-linux-gnu/3.4.1/include/g++-v3 --with-local-
prefix=/usr/local --disable-werror --enable-shared --enable-nls --without-included-gettext -
-disable-multilib --enable-__cxa_atexit --enable-clocale=gnu
Thread model: posix
gcc version 3.4.1 20040803 (Gentoo Linux 3.4.1-r3, ssp-3.4-2, pie-8.7.6.5)
pkb@pegasos2 build_gcc $ ../gcc-3.4.4/configure --prefix=$INSTDIR --disable-nls \
--enable-languages=c,c++ --enable-threads=posix --enable-shared --with-system-zlib \
--disable-checking --enable-cstdio=stdio --enable-__cxa_atexit >c.log 2>&1
```

"gcc –v" shows configuration an existing gcc was built with

It's a useful tool to determining parameters needed for a new gcc

4) Building PowerPC Linux native glibc

```
pkb@pegasos2 build_gcc $ cd ../build_glibc
pkb@pegasos2 build_glibc $ ../glibc-2.3.5/configure --prefix=/usr --disable-nls \
  --with-gd=no --without-cvs --disable-profile --enable-add-ons=yes >i.log 2>&1
pkb@pegasos2 build_glibc $ make >m.log 2>&1
pkb@pegasos2 build_glibc $ mkdir ~/GLIBC
pkb@pegasos2 build_glibc $ make install_root=~/GLIBC install >i.log 2>&1
pkb@pegasos2 build_glibc $
```

--prefix=/usr is required; install_root=foo overrides on “make install”

Installing glibc on a running Linux system requires extreme care

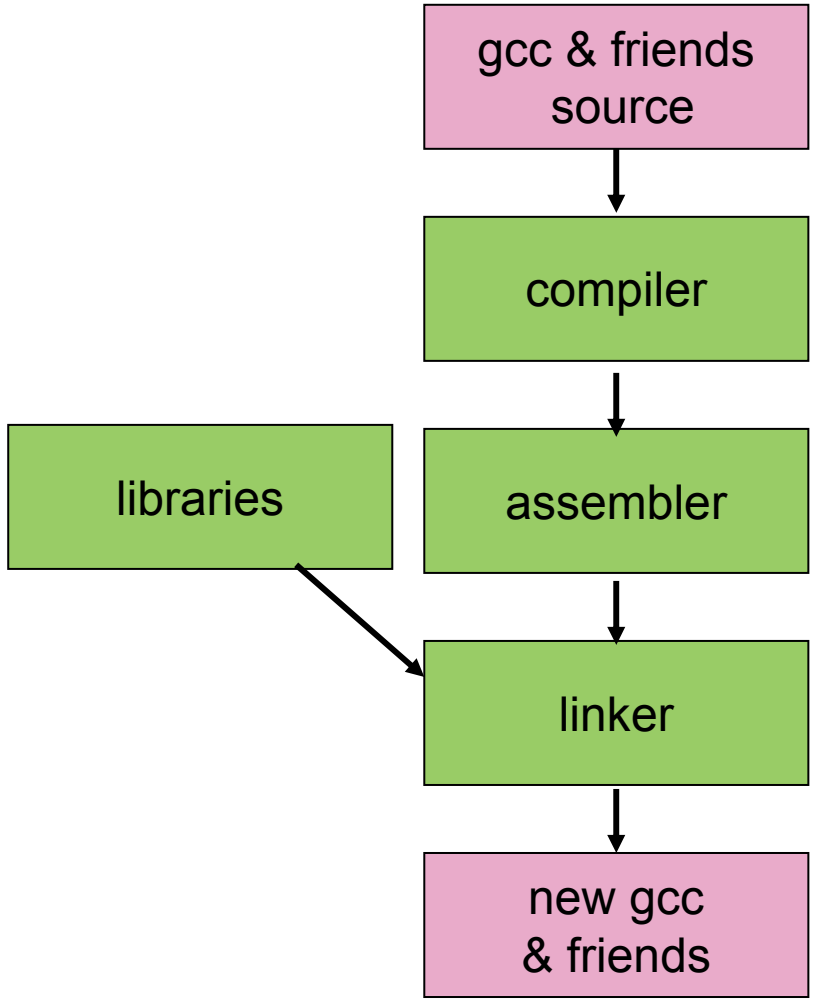
- Mistakes may leave the system in an inoperable state
- Updating to a new glibc revision may require application recompiles
- Study references, your Linux distribution’s methods very carefully!

Looking back

At this point, there's a new gcc, as, ld, in the $INSTDIR, and glibc elsewhere

Adding $INSTDIR/bin to the front of $PATH will make the new gcc available

Paths to binaries are hardcoded, so it's not possible to move $INSTDIR elsewhere without rebuilding the tools



Building a cross compiler

The steps to build a cross GNU compiler toolchain generally track those to build a native toolchain

glibc is big and heavyweight; newlib is frequently preferred for embedded runtime libraries

Let's jump right in . . .

Host, Target, Build

Let's look at the start of the output the native binutils configure generated:

```
pkb@pegasos2 build_binutils $ ../binutils-2.16/configure . . .
creating cache ./config.cache
checking host system type... powerpc-unknown-linux-gnu
checking target system type... powerpc-unknown-linux-gnu
checking build system type... powerpc-unknown-linux-gnu
checking for powerpc-unknown-linux-gnu-ar... no
```

"host system type" refers to the system where the tool will be run

"target system type" refers to the system where the output of the tool will be run

"build system type" refers to the system where the tool is being built

configure is pretty smart about determining build system type

"target system type" can be overridden in order to build cross tools

GNU tools PowerPC embedded target systems

There's many system types available for building GNU tools.

These are probably the most widely applicable embedded PowerPC targets:

powerpc-eabi
- PowerPC embedded ABI (aka EABI) target

powerpc-eabisim
- PowerPC embedded ABI (aka EABI) target with psim-compatible crt0.o

powerpc-eabispe
- PowerPC e500 ABI target

powerpc-unknown-linux-gnu
- PowerPC Linux ABI target

1) Getting started: set up your directories

```
pkb@ninja pkb $ mkdir ~/GNUsrc ~/GNU
pkb@ninja pkb $ cd GNUsrc
pkb@ninja GNUsrc $ ls ../GNUarchive
binutils-2.16.tar.bz2     gcc-3.4.4.tar.bz2newlib-1.13.0.tar.gz
pkb@ninja GNUsrc $ tar jxf ../GNUarchive/binutils2.16tar.bz2
pkb@ninja GNUsrc $ tar jxf ../GNUarchive/gcc3.4.4.tar.bz2
pkb@ninja GNUsrc $ tar zxf ../GNUarchive/newlib-1.13.0.tar.gz
pkb@ninja GNUsrc $ ls
binutils-2.16    gcc-3.4.4         newlib-1.13.0
pkb@ninja GNUsrc $ mkdir build_binutils build_gcc build_newlib
```

Create a working directory and an installation directory

Unpack GCC source tarballs into the working directory

- but this time with newlib in place of glibc

Create a working subdirectory for each package you'll build

- DON'T build the packages in their source directories

2) Building x86 to PowerPC cross binutils

```
pkb@ninja GNUsrc $ export INSTDIR=/home/pkb/GNU
pkb@ninja GNUsrc $ export TARGET=powerpc-eabi
pkb@ninja GNUsrc $ cd build_binutils
pkb@ninja build_binutils $ ../binutils-2.16/configure --prefix=$INSTDIR \
  --target=$TARGET --enable-languages=c,c++ >c.log 2>&1
pkb@ninja build_binutils $ make >m.log 2>&1
pkb@ninja build_binutils $ make install >i.log 2>&1
pkb@ninja build_binutils $ export PATH=$INSTDIR/bin:$PATH
pkb@ninja build_binutils $
```

set $TARGET to target system type

--prefix=$INSTDIR specifies directory where binutils will be installed, as before

Add $INSTDIR/bin to $PATH so that subsequent steps use the new binutils

Cross Host, Target, Build

Now let's look at the start of the output the cross binutils configure generated:

```
pkb@ninja build_binutils $ ../binutils-2.16/configure …
creating cache ./config.cache
checking host system type... i686-pc-linux-gnu
checking target system type... powerpc-unknown-eabi
checking build system type... i686-pc-linux-gnu
…
```

"target system type" differs from host, build when building a cross compiler

3) Building x86 to PowerPC cross gcc

```
pkb@ninja build_binutils $ cd ../build_gcc
pkb@ninja build_gcc $ ../gcc-3.4.4/configure --prefix=$INSTDIR --target=$TARGET \
  --enable-languages=c,c++ --with-newlib \
  --with-headers=../newlib-1.13.0/newlib/libc/include >c.log 2>&1
pkb@ninja build_gcc $ make >m.log 2>&1
pkb@ninja build_gcc $ make install >i.log 2>&1
pkb@ninja build_gcc $
```

--prefix=$INSTDIR specifies directory where binutils will be installed

--enable-languages specifies language front ends

--with-newlib causes certain gcc library routine calls to rely on newlib instead

--with-headers as shown assures that the cross-gcc will use C library header files compatible with the yet-to-be built newlib C standard libraries

- This is a way around compiling gcc again after building newlib

4) Building x86 to PowerPC cross newlib

```
pkb@ninja build_binutils $ cd ../build_newlib
pkb@ninja build_newlib $ ../newlib-1.13.0/configure --prefix=$INSTDIR \
  --target=$TARGET >c.log 2>&1
pkb@ninja build_newlib $ make >m.log 2>&1
pkb@ninja build_newlib $ make install >i.log 2>&1
pkb@ninja build_newlib $
```

Building newlib can be very straightforward, but time consuming

--enable parameters can be used to reduce the sets of libraries built

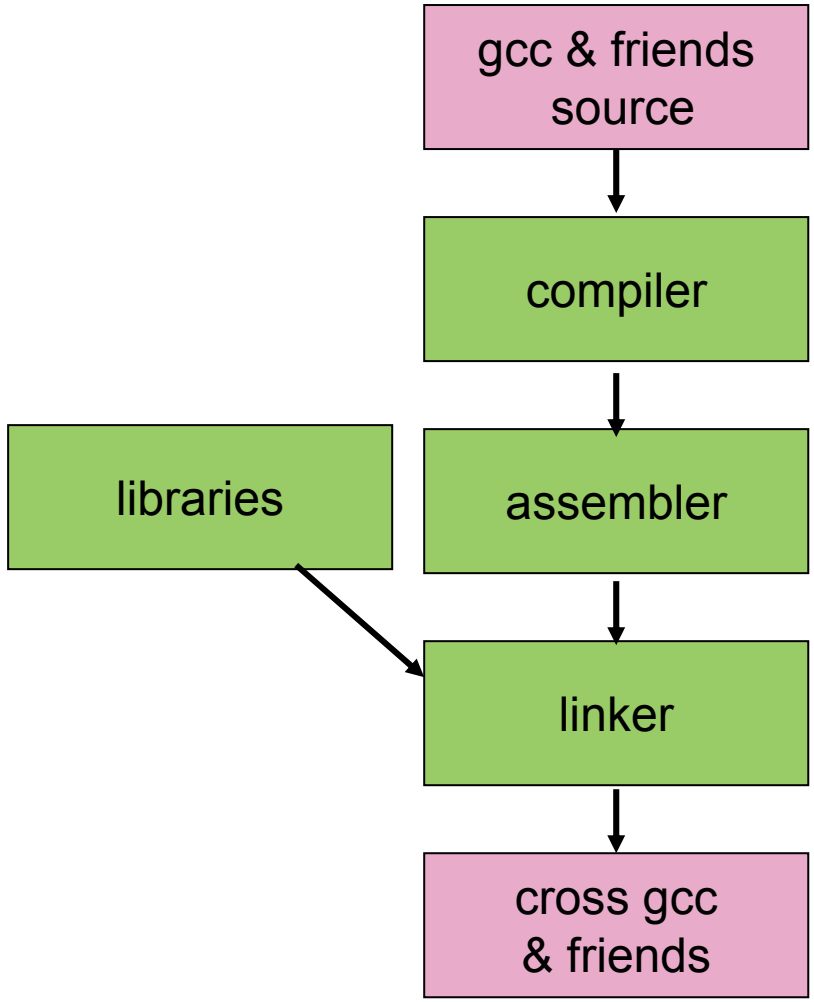
Note that newlib is built with the cross-gcc built in the previous step

Looking back again

Now, there's a cross gcc, as, ld, newlib in the $INSTDIR

Adding $INSTDIR/bin to the front of $PATH will make them available

As for the native tools, paths to binaries and library directories are hardcoded, so it's not possible to move $INSTDIR elsewhere without rebuilding the tools



crosstool

Building a cross GCC with glibc instead of newlib is a bit more complicated

crosstool, updated and maintained by Dan Kegel, simplifies the process by automating most steps

Licensed under the GNU Public License (GPL)

Current version is crosstool 0.35

Home page: http://www.kegel.com/crosstool

Download available via the home page

1) Getting started

```
pkb@ninja pkb $ mkdir ~/GNUsrc
pkb@ninja pkb $ cd GNUsrc
pkb@ninja GNUsrc $ tar zxf ../crosstool-0.35.tar.gz
pkb@ninja crosstool-0.35 $ cd crosstool-0.35
pkb@ninja crosstool-0.35 $ ls demo-ppc*
demo-ppc405.dat  demo-ppc604.dat    demo-ppc750.dat  demo-ppc970.dat
demo-ppc440.dat  demo-ppc7450.dat  demo-ppc860.dat
pkb@ninja crosstool-0.35 $ cat powerpc-7450.dat
TARGET=powerpc-7450-linux-gnu
TARGET_CFLAGS="-O"
GCC_EXTRA_CONFIG="--with-cpu=7450 --enable-altivec --enable-cxx-flags=-mcpu=7450"
pkb@ninja crosstool-0.35 $ vi demo-ppc7450.sh
```

Create a working directory and untar crosstool

*.dat files contain processor-specific configuration variables

There's a corresponding demo*.sh script

- These are good shortcuts that don't require learning all of crosstool's powers
- There are a few choices you may want to edit

2) Editing demo-ppc7450.sh

```
#!/bin/sh
set -ex
TARBALLS_DIR=$HOME/downloads
RESULT_TOP=/opt/crosstool
export TARBALLS_DIR RESULT_TOP
GCC_LANGUAGES="c,c++"
export GCC_LANGUAGES

# Really, you should do the mkdir before running this,
# and chown /opt/crosstool to yourself so you don't need to run as root.
mkdir -p $RESULT_TOP

# Build the toolchain.  Takes a couple hours and a couple gigabytes.
#eval `cat powerpc-7450.dat gcc-2.95.3-glibc-2.2.2.dat` sh all.sh --notest
#eval `cat powerpc-7450.dat gcc-3.2.3-glibc-2.2.5.dat` sh all.sh --notest
#eval `cat powerpc-7450.dat gcc-3.3-glibc-2.2.5.dat`  sh all.sh --notest
#eval `cat powerpc-7450.dat gcc-3.3-glibc-2.3.2.dat`  sh all.sh --notest
#eval `cat powerpc-7450.dat gcc-3.3.1-glibc-2.3.2.dat` sh all.sh --notest
#eval `cat powerpc-7450.dat gcc-3.3.2-glibc-2.3.2.dat` sh all.sh --notest
#eval `cat powerpc-7450.dat gcc-3.3.3-glibc-2.3.2.dat` sh all.sh --notest
#eval `cat powerpc-7450.dat gcc-3.3.4-20040530-glibc-2.3.2.dat` sh all.sh --notest
#eval `cat powerpc-7450.dat gcc-3.4.0-glibc-2.3.2.dat` sh all.sh --notest
 eval `cat powerpc-7450.dat gcc-3.4.1-glibc-2.3.3.dat` sh all.sh --notest
#eval `cat powerpc-7450.dat gcc-3.4.1-glibc-20040827.dat` sh all.sh --notest

echo Done.
```

Installation directory

Supported languages

Uncomment a choice of gcc, glibc versions

3) Building the toolchain

```
pkb@ninja crosstool-0.35 $ mkdir /opt/crosstool
pkb@ninja crosstool-0.35 $ sh demo-ppc7540.sh >cross.log 2>&1
pkb@ninja crosstool-0.35 $ /opt/crosstool/gcc-3.4.1-glibc-2.3.3/powerpc-7450-linux/gnu/bin/powerpc7450-linux-
   gnu-gcc  -v
   Reading specs from  /opt/crosstool/gcc-3.4.1-glibc-2.3.3/powerpc-7450-linux-gnu/lib/gcc/powerpc-7450-linux-
   gnu/3.4.1/specs
   Configured with:
   /home/pkb/GNUsrc/crosstool-0.34/build/powerpc-7450-linux-gnu/gcc-3.4.1-glibc-2.3.3/gcc-3.4.1/configure
   --target=powerpc-7450-linux-gnu --host=i686-host_pc-linux-gnu
   --prefix=/opt/crosstool/gcc-3.4.1-glibc-2.3.5/powerpc-7450-linux-gnu
   --with-cpu=7450 --enable-altivec --enable-cxx-flags=-mcpu=7450
   --with-headers=/opt/crosstool/gcc-3.4.1-glibc-2.3.3/powerpc-7450-linux-gnu/powerpc-7450-linux-gnu/include
   --with-local-prefix=/opt/crosstool/gcc-3.4.1-glibc-2.3.3/powerpc-7450-linux-gnu/powerpc-7450-linux-gnu
   --disable-nls --enable-threads=posix--enable-symvers=gnu
   --enable-__cxa_atexit --enable-languages=c,c++ --enable-shared
   --enable-c99 --enable-long-long
   Thread model: posix
   gcc version 3.4.1
pkb@ninja crosstool-0.35 $
```

Make your installation directory

Execute your demo*.sh script

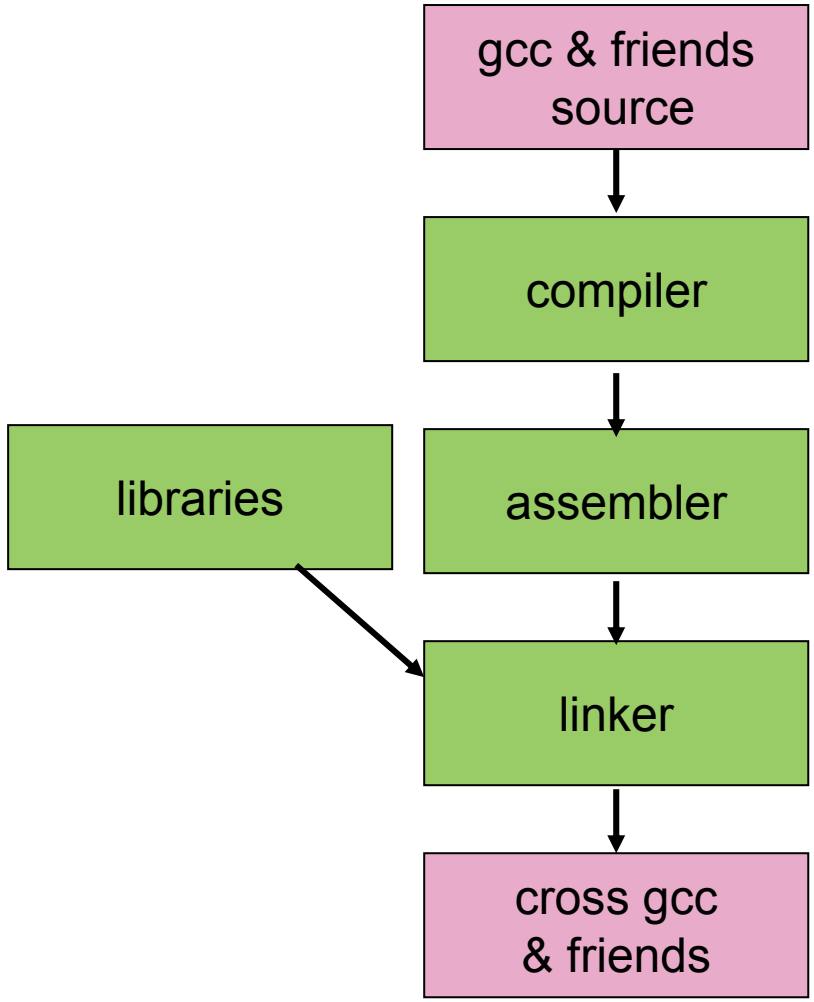
crosstool has more capabilities not explored here

Looking back yet again

Now, there's a cross gcc, as, ld, glibc deep inside /opt/crosstool

Adding /opt/crosstool/gcc-3.4.1-glibc2.3.3/powerpc-7450-linux-gnu/bin to the front of $PATH will make available the tools built in this example

As we've seen before, paths to binaries and library directories are hardcoded, so it's not possible to move GCC elsewhere without rebuilding the tools



"Canadian cross"

Yes, it's possible to build a cross GNU tool chain that runs on a different host than where it's built

- involves setting "host system type" and "build system type" differently
- eg under Solaris, build a powerpc-eabi targeted cross toolchain that runs under Cygwin

This is termed a "Canadian cross compiler" for obscure reasons

Possibly useful if you're a system administrator building tool chains for users with a variety of development hosts on their desks

We'll take a pass on techniques for building these, but you should know it is possible

When things don't work as expected

You may find that you have builds that fail or apparent bugs, especially if you are using bleeding edge releases

There's many resources on the net to help you out, and "Google is your friend"

Most GNU projects have FAQs and/or mailing lists

- http://mail.gnu.org/archive/html

Gentoo Linux forums

- http://forums.gentoo.org

Linuxfromscratch lfs-dev mailing list archives

- http://linuxfromscratch.org/pipermail/lfs-dev

CVS downloads

If you learn that you need a feature or bugfix that hasn't made it into a package release yet, you may choose to download a source code snapshot

sourceware.org hosts or links to CVS servers for binutils, gcc, glibc, newlib, gdb and other projects

For example, to download the current snapshot of newlib:

- Browse to their cvsweb interface (http://sourceware.org/cgi-bin/cvsweb.cgi?cvsroot=sourceware)
- Or invoke cvs directly:

```
cvs -z 9 -d :pserver:anoncvs@sourceware.org:/cvs/src login
   <password is "anoncvs">
cvs -z 9 -d :pserver:anoncvs@sourcware.org:/cvs/src co newlib
```

A few useful GCC-related references on the web

The Free Software Foundation home page - http://www.fsf.org

"Installing GCC" - http://gcc.gnu.org/install

"Cross GCC FAQ" - http://www.sthoward.com/CrossGCC/

"Building and Testing gcc/glibc cross toolchains" - http://kegel.com/crosstool/

In Closing

GNU tools are very portable and configurable, so building them can be a challenge

Studying working methods, plus a little patience, can reward you with your own tailored toolset

There's more than one way to build GNU tool chains; we've shown one for common scenarios to get you started


freescale™
semiconductor