CrossGCC Frequently Asked Questions

Maintainer: Scott Howard, scott@objsw.com Version: 1.01, Last Updated December 3, 1999

Contents

1 Introduction

1.1 What is crossgcc?

'crossgcc' is the name of a mailing list for discussing issues regarding using GCC (plus the various ancilliary pieces) as a cross-compiler. Its main focus is using GCC and GDB to compile and debug code for embedded applications. Messages may be posted to this list by sending e-mail to `crossgcc@sourceware.cygnus.com`

1.2 How can I subscribe/unsubscribe to the list?

Subscription requests should be sent to ‘ crossgcc-subscribe@sourceware.cygnus.com’ . No subject line or message body is required; the mailing list software will add your 'From:' address to the list of subscribers. To unsubscribe, send a message to‘ crossgcc-unsubscribe@sourceware.cygnus.com’ , again with no subject line or message body. The email address you use to unsubscribe must be identical, character for character, to one used to subscribe, and case is important too. This list is run by the ‘ezmlm’ software.

1.3 Where can I find an archive of the list?

Cygnus maintains an archive at http://sourceware.cygnus.com/ml/crossgcc Also, Elijah Laboratories have an archive of the CrossGCC mailing list on their web site, at http://www.elilabs.com/xgccarc/index.html.

1.4 Where can I find copies of this FAQ?

www:

http://www.objsw.com/CrossGCC

ftp:

ftp://ftp.objsw.com/pub/crossgcc

1.5 How can I submit suggestions/improvements/additions for this FAQ?

This FAQ is unfinished (in the sense that there are some obvious sections it needs, some included below). Please send contributions to scott@objsw.com.

2 List of the pieces needed and where to find them

For a complete toolchain you need a compiler (GCC), assembler (GAS), linker (GLD), various utilities (Binutils), debugger (GDB), and a library (GLIBC or NEWLIB). For c++ programming you'll also want libstdc++; this is now packaged with the GCC compiler as of gcc 2.95.

The master repository of GNU software is ftp://ftp.gnu.org/pub/gnu. It is mirrored all over the world so please try to use a site closer to you to avoid overloading prep. A list of sites may be found in the file ftp://ftp.gnu.org/pub/gnu/GNUinfo/FTP.

Compiler:

GCC 2.95.2, file gcc-2.95.2.tar.gz

Assembler,Linker,Utilites:

GAS 2.9.1/GLD 2.9.1/binutils 2.9.1, file binutils-2.9.tar.gz

Debugger:

GDB 4.17, file gdb-4.17.tar.gz

Library:

There are currently two choices for a library, the GNU libc (Glibc) and a library put together by Cygnus (Newlib). GLIBC is generally better for larger (Unix-based) systems, while newlib is more appropriate for embedded applications.

- GLIBC 2.0.6, file `glibc-2.0.6.tar.gz`
- Newlib 1.8.2, file `ftp://sourceware.cygnus.com:/pub/newlib/newlib-1.8.2.tar.gz`

3 Patches

This section is intended to document patches that are available from various sources.

3.1 linux-x-djgpp.diff

This patch makes a small change to the way local labels are generated when you build gcc to run on MS-DOS. This patch is only used when cross-compiling to MS-DOS, it is not needed when cross-compiling to other targets.

4 How to configure, build, and install GCC as a cross-compiler

4.1 Unix Systems

Notes:

1. The author has more familiarity with Newlib than Glibc so Newlib will be used in all examples below.
2. GAS and GLD are now distributed in the 'binutils' distribution, so all further references to GAS and GLD in this section will use Binutils instead.
3. If you have a 'make' that understands VPATH (like GNU make), it is highly recommended that you build the pieces in a different directory from the sources. The examples below will assume this. Also, if you do decide to build in a directory different from the source, GNU make is generally the only make that can handle this correctly.

   The pieces have to be built and installed in a particular order. Why? Clearly the cross-compiler is needed to build the libraries, so GCC must be built before Newlib. Also, GCC has its own library (called libgcc) so Binutils must be built before GCC.

Here is the recommended order:

1. Binutils
2. GCC
3. Newlib

4. GDB

Suppose

- you've unpacked all the sources into `/home/foo`,
- you want to build a sparc-solaris2 hosted m68k-coff cross-compiler,
- the cross-compiler is to be installed in `/bar`.

The build procedure would then look like this.

```
cd /home/foo
host=sparc-sun-solaris2
target=m68k-coff
prefix=/bar
i=$prefix/bin

mkdir build-binutils build-gcc build-newlib build-gdb

# Configure, build and install binutils
 cd build-binutils
../binutils-2.9.1/configure --target=$target --prefix=$prefix -v
make all install

# Configure, build and install gcc
cd ../build-gcc
../gcc-2.8.1/configure --target=$target --prefix=$prefix -v
make all install

# Configure, build and install newlib
cd ../build-newlib
../newlib-1.8.1/configure --target=$target --prefix=$prefix -v
# The settings for FOO_FOR_TARGET aren't necessary if you put $prefix/bin
# in your path before running this.
make all install \
        CC_FOR_TARGET=$i/${target}-gcc \
        AS_FOR_TARGET=$i/${target}-as \
        LD_FOR_TARGET=$i/${target}-ld \
        AR_FOR_TARGET=$i/${target}-ar \
        RANLIB_FOR_TARGET=$i/${target}-ranlib

# Configure, build and install gdb
cd ../build-gdb
../gdb-4.17/configure --target=$target --prefix=$prefix -v
make all install
```

4.2 One Pass Installation

Is there an easier way? Yes!

If you study the top-level Makefile that comes with binutils-2.9.1, newlib-1.8.1, or gdb-4.17, you'll see that they're all quite similar (not surprising since they're essentially the same file). You'll also discover that it is capable of building and installing the entire toolchain in one pass. The catch is that is assumes a directory layout different than what would normally appear when you unpack the sources.

What we now need to do is turn this directory layout

```
./binutils-2.9.1/
               bfd/
               binutils/
               config/
               gas/
               include/
               ld/
               libiberty/
               opcodes/
               texinfo/
./gcc-2.95.2/
./newlib-1.8.2/
               config/
               libgloss/
               newlib/
               texinfo/
```

into

```
./src/
      bfd/
      binutils/
      config/
      gas/
      gcc/
      include/
      ld/
      libgloss/
      libiberty/
      newlib/
      opcodes/
      texinfo/
```

Where's GDB? GDB is left out because it shares sources with Binutils (e.g. bfd, include, libiberty, and opcodes). One can't 'mix and match' them, they each need there own copies (since they were tested and released with their own copies). GDB can be built separately afterwards.

One might well ask what's the point of this pseudo one-pass installation and that would be a good question. It simplifies the installation a little, and in particular the 'Canadian Cross' installation (see 4.9 (What is a Canadian Cross?). Binutils and Newlib share 'config' and 'texinfo' directories; we can use Binutils' copies.

A script exists to reorganize the above source tree using symlinks. It assumes:

- you're using a version of make that understands VPATH (e.g. GNU make)
- `./binutils-2.9.1` exists
- `./gcc-2.95.2` exists
- `./newlib-1.8.2` exists

The script is one-tree-1.6.sh (`ftp://ftp.objsw.com/pub/crossgcc/one-tree-1.6.sh`). It will create a subdirectory called `src`.

If one or more of the subdirecties listed above does not exist, the script will attempt to un-pack a tar file from a subdirectory called `tars` to create the missing source files. Finally, the `src` directory is created and the source files are symlinked into it.

After running the script, do this:

```
mkdir build
cd build
../src/configure --target=$target --prefix=$prefix -v
make all install

mkdir ../build-gdb
cd ../build-gdb
../gdb-4.17/configure --target=$target --prefix=$prefix -v
make all install
```

4.3 Can I build in a directory different from the source tree?

Yes.

However, it requires a version of make that understands VPATH. SunOS make may or may not be sufficient: its VPATH support has problems which require extra effort to support, and developers don't always adhere to the restrictions. If you don't use GNU make, you're on your own as far as getting the build to work.

4.4 Can I use a non-GNU make?

Yes.

However, there are a few things to be wary of:

- you must build in the source tree
- testing generally isn't done with non-GNU makes, so one is more likely to run into build problems. These problems can be fixed of course, but it may require effort on your part.

4.5 Is there a script available to automate the build process?

Glad you asked. Yes, there is. It is called 'build-cross.sh' (`ftp://ftp.objsw.com/pub/crossgcc/build-cross.sh`) and it builds a cross-compiler and runtime libraries on your Unix/Linux machine that targets your favourite embedded processor.

The script takes two command line parameters, with the following syntax:

```
sh build-cross.sh <targetname> [install]
```

where <targetname> is the 'configure' name for the embedded target system, and [install] is an optional flag which causes the script to install the cross-compiler tools after they have been built.

The typical way to use this script is to run it once under your normal user account, without the 'install' flag. This causes the tools to be built in your working directory, but not installed into the system's execution directory (normally /usr/<targetname> on Linux systems). Once the build is done, log in now as 'root' and run the script again, this time with the 'install' flag on the command line. This time, the tools will be installed into their final resting place, from which point they will be available for use to build programs for your embedded target.

For example, say you wanted to build a cross compiler targeting embedded PowerPC systems. The configure name to use is powerpc-eabi. A typical session might look like this:

```
[scotth] $ ./build-cross.sh powerpc-eabi
(tons of messages go by while the tools are being built.)
[scotth] $ su
Password:
[root] # ./build-cross.sh powerpc-eabi install
(Somewhat fewer messages this time, while the tools are installed.)
[root] # exit
[scotth] $
```

You can build and install in one command by always specifying the 'install' flag; however this requires that the build be performed with root privileges, since this privilege level is usually required in order to install the tools to their execution directories, and performing long builds like this with 'root' privilege can be a security risk. But if security is not an issue, this can be more convenient.

4.6 Cygnus Releases

4.6.1 Where are the info files?

Cygnus releases differ from FSF releases in that files that are not really source files but are built from other files (like yacc files and texinfo files) are not included. Instead, they are built with the rest of the toolchain.

4.6.2 How do I build a Cygnus release?

Cygnus releases are essentially the 'one-pass installation' tree, except that a lot more tools are included (e.g. bison, flex, expect, gdb, make, tcl, texinfo). To build a toolchain from a Cygnus release, you should consult

the documentation that came with it (there may be last minute release notes, or this FAQ may be out of date, etc.).

But for those who happen to come upon a Cygnus release, here is a quick introduction. Suppose you happen upon a Cygnus release and want to build a `sparc-sun-solaris2` cross `m32r-elf` compiler. Do this:

```
src=/path/to/source/tree
rel=/path/to/install/tree
host=sparc-sun-solaris2
target=m32r-elf
mkdir build
cd build
$src/configure --target=$target --prefix=$rel
make all info install install-info
PATH=$PATH:$rel/bin ; export PATH
```

You can also run dejagnu on the build tree at this point with

```
make -k check
```

The gcc and g++ execute tests won't do much in this particular example unless dejagnu has been configured to use the simulator or target board for the execute tests.

4.7 Win32 hosted cross-compilers

There is work in progress that will let the GNU tools be used as a native compiler in the win32 environment and also as a cross-compiler (either win32 hosted or win32 targeted). Join the `gnu-win32@cygnus.com`mailing list if you wish to help out. The release is kept in `ftp://ftp.cygnus.com/pub/gnu-win32`. The configuration for this project is `i386-cygwin32`.

Subscription requests should be sent to `gnu-win32-request@cygnus.com`with a body of 1 line containing '`subscribe gnu-win32 <your-email-address>`' and nothing else (and no <>'s around your email address).

4.8 Gnu CC and MS-DOS

The primary MSDOS port is version 2 of 'djgpp', a port of the GNU tools to MSDOS by dj Delorie, using 'go32-v2', a 32 bit extender. See the djgpp FAQ for more details (`http://www.delorie.com/djgpp/v2faq/`).

Using the djgpp runtime system, Gnu CC can run on MS-DOS as a 32-bit DOS application. It can also run on another type of system (say, Linux) and cross-compile DOS programs that will run on MS-DOS.

4.8.1 How do I build a cross-compiler with DJGPP?

The procedure is basically this.

- Use gunzip to decompress the various archives.

- Use djtarx to untar them and resolve 8.3 conflicts
- Run 'configure' in all the right places to configure for a native compiler.
- Manually (read the INSTALL file) copy the target-specific files around to reconfigure the system for the given target. Edit the Makefiles as appropriate to include the target snippets.
- Run make.

4.8.2 How do I create a cross-compiler for DOS on my Unix/Linux machine ?

You need the following pieces:

- The source archives for gcc and the binary utilities, as listed in section 2
- The patch file `linux-x-djgpp.diff`, available at `ftp://ftp.objsw.com/pub/crossgcc/linux-x-djgpp.diff`
- The djgpp cross-compiler archive `djcrx202.zip`, available from `ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2/djcrx202.zip`.

The script assumes that the zip file is in a subdirectory named `tars`. It installs libraries and header files from the zip file into their final installation directory (normally /usr/i386-pc-msdosdjgpp); it then configures, compiles and installs the binary utilities, and then compiler. The script must be run with 'root' privileges.

After you have run the script, you will be able to compile DOS programs using the compiler `i386-pc-msdosdjgpp-gcc`. For example, if you wanted to compile the program `hello.c` to run on DOS:

```
[scotth] $ i386-pc-msdosdjgpp-gcc -o hello.exe hello.c
```

4.9 Canadian Crosses

4.9.1 What is a Canadian Cross?

One cool thing about the GNU tools is that they support building a cross compiler for one host on another host (i.e. building a cross-compiler with a cross-compiler). This is called a 'Canadian Cross' because at the time a name was needed, Canada had three national parties.

4.9.2 How do I build an MSDOS hosted cross compiler without using MSDOS?

Suppose one wanted to build an MSDOS cross `m68k-coff` cross compiler on a `sparc-sunos4` machine. The procedure is as follows. Note that it is quite lengthy; that's because four compilers are involved (three to build the one we want).

Why four? Remember, we're building the tools completely on a Unix box, therefore all the programs that run during the build process must obviously run on the Unix box. We can't skip over to an MSDOS machine, run something there, and come back - the entire toolchain is built from scratch on the Unix box.

The first compiler that is needed is a `sunos4` cross `m68k-coff` compiler to build the m68k-coff libraries. But in order to build that we need a `sunos4` native compiler. That's two. We also need a `sunos4`

cross `i386-pc-msdosdjgpp` compiler to build the programs that run on MSDOS . Finally, we need an `i386-pc-msdosdjgpp` cross `m68k-coff` compiler: our final goal.

Four compilers. However, the process is quite straightforward once you understand all the pieces that are needed.

Previous versions of this FAQ featured a script that would build the `sunos4` cross `i386-pc-msdosdjgpp` as part of the script. This worked because they targeted version 1 of the djgpp port of Gnu CC, which used the newlib run-time library. In order to take advantage of the considerable benefits of djgpp version 2 (such as virtual memory and better debugging support), we have elected to build the `sunos4` cross `i386-pc-msdosdjgpp` as a separate step, using the `build-djgpp.sh` script file. See the section on building a Linux to djgpp cross compiler 4.8.2 (here).

Assume the source tree has been created with the 'one-tree' script. The following is the list of steps, written so that it can be copied into a shell script and run.

```
#!/bin/sh
# This script is build-3way.sh from crossgcc FAQ 1.0
# Before using this script it is a good idea to check with the most recent
# version of the FAQ to see if any changes have been made.  The FAQ can be
# obtained from http://www.objsw.com/CrossGCC.
#
# This script assumes that the source tree (in directory 'src') contains binutils, gcc,
# and newlib, constructed with the one-tree script
#
# Syntax: sh build-3way.sh targetname [install]
#
# The default is to configure and build, but not install.
#
# The recommended way to use this script is to modify the variables
# build,host,target,src,rel,relexec as necessary, then run:
#
# build-3way.sh targetname
# build-3way.sh targetname install
#
# The process is rather involved as there are a lot of steps.
# On the other hand, it is really rather straightforward.
# The goal is to build a $host cross $target toolchain.  Some hosts aren't
# well suited to program development (eg: msdos) and other hosts may not have
# complete GNU support yet.  Both of these cases are ideally handled by a
# script like this where we build everything in a more familiar and
# comfortable environment (eg: unix).
#
# The toolchain we are building is composed of basically two pieces:
#
# 1) programs that run in the host environment.
#    These include gcc, cpp, cc1, as, ld, objdump, etc.  These critters
#    are built with a $build cross $host cross-compiler.
#
```

```
# 2) libraries of functions that run in the target environment.
#    These include libgcc.a, libc.a, libm.a, etc.  These critters are
#    built with a $build cross $target cross-compiler.
#
# We end up building 3 complete toolchains: $build cross $host (done in a separate script),
# $build cross $target, and ultimately $host cross $target.
# Remember, the only environment in which we can run programs is $build:
# that is the reason for the complexity.
#
# The cool thing is that this can be done at all!

set -e

here=‘pwd‘

target=$1
action=$2

build=‘src/config.guess‘
host=i386-pc-msdosdjgpp
src=$here/src
rel=/tmp/test
relexec=$rel/H-${host}

# Build directory for the $build cross $target toolchain.
b2t=$here/b-${build}-x-${target}
# Build directory for the $host cross $target toolchain.
h2t=$here/b-${host}-x-${target}

# Bail if no target given.
if [ x"$target" = x ]
then
        echo "usage: ’$0 targetname’ [install], where:"
        echo " - ’targetname’ specifies the target configuration to build"
        echo " - ’install’ (optional) directs the script to install the tools (in $rel)."
        exit 1
fi

######################################################################
#
# The first step is to build a $build cross $host cross-compiler.
#
# Previous versions of this script built this cross-compiler from the
# same source tree as was used for the embedded target. However, this
# doesn’t really work anymore because our MS-DOS host configuration
# no longer uses newlib as its runtime library... so now you must do
# this step separately by running the script ’build-djgpp.sh’.
```

```
#
# See the CrossGCC FAQ for more information.

#######################################################################
#
# Now build a $build cross $target toolchain.
# The value for --prefix we give here is /tmp/junk as we don't intend
# to install this toolchain.

if [ ! -f $b2t/configure.done ] ; then
    [ -d $b2t ] || mkdir $b2t
    (cd $b2t ; CC=gcc $src/configure --host=${build} --target=${target} --prefix=/tmp/junk --with-gnu-l
--with-gnu-as --with-newlib -v)
    touch $b2t/configure.done
fi
(cd $b2t ; make -w all-gcc CC=gcc CFLAGS=-g LANGUAGES="c c++")

#######################################################################
#
# Now that we've built the tools that we need, we can finally build
# our $host cross $target toolchain.

# Both configure and make need to be told where to find the various pieces.
# Define several variables of the things we need to pass to configure and make.

# These are for building programs that run on $build.
CC_FOR_BUILD=gcc
CXX_FOR_BUILD=gcc

# These are for building programs and libraries that run on $host.
CC=$host-gcc
AR=$host-ar
RANLIB=$host-ranlib

# These are for building libraries that run on $target.
CC_FOR_TARGET="$b2t/gcc/xgcc -B$b2t/gcc/ -isystem $src/winsup/include -isystem $b2t/${target}/newlib/ta
GCC_FOR_TARGET="$CC_FOR_TARGET"
CC_FOR_TARGET="$CC_FOR_TARGET"
CXX_FOR_TARGET="$CC_FOR_TARGET"
# use the correct names for as, ld, and nm.
if [ -f $b2t/gas/as-new ]
then
        AS_FOR_TARGET=$b2t/gas/as-new
        LD_FOR_TARGET=$b2t/ld/ld-new
        NM_FOR_TARGET=$b2t/binutils/nm-new
else
        AS_FOR_TARGET=$b2t/gas/as.new
```

```
        LD_FOR_TARGET=$b2t/ld/ld.new
        NM_FOR_TARGET=$b2t/binutils/nm.new
fi

AR_FOR_TARGET=$b2t/binutils/ar
RANLIB_FOR_TARGET=$b2t/binutils/ranlib
# $DLLTOOL_FOR_TARGET is only needed for win32 hosted systems, but
# it doesn't hurt to always pass it.
DLLTOOL_FOR_TARGET=$b2t/binutils/dlltool
# For go32 cannot use -g because it can overflow coff debug info tables.
CFLAGS=-O
CXXFLAGS=-O

# Ready.  Configure and build.
if [ ! -f $h2t/configure.done ] ; then
    [ -d $h2t ] || mkdir $h2t
    (cd $h2t ; CC="$CC" AR="$AR" RANLIB="$RANLIB" $src/configure --build=${build} --host=${host}
--target=${target} --prefix=$rel --exec-prefix=$relexec --with-gnu-ld --with-gnu-as -v)
    touch $h2t/configure.done
fi

cd $h2t
make -w all \
        LANGUAGES="c c++" \
        CFLAGS="$CFLAGS" \
        CXXFLAGS="$CXXFLAGS" \
        CC_FOR_BUILD="$CC_FOR_BUILD" \
        CXX_FOR_BUILD="$CXX_FOR_BUILD" \
        CC="$CC" \
        AR="$AR" \
        RANLIB="$RANLIB" \
        GCC_FOR_TARGET="$CC_FOR_TARGET" \
        CC_FOR_TARGET="$CC_FOR_TARGET" \
        CXX_FOR_TARGET="$CC_FOR_TARGET" \
        AS_FOR_TARGET=$AS_FOR_TARGET \
        LD_FOR_TARGET=$LD_FOR_TARGET \
        AR_FOR_TARGET=$AR_FOR_TARGET \
        NM_FOR_TARGET=$NM_FOR_TARGET \
        RANLIB_FOR_TARGET=$RANLIB_FOR_TARGET \
        DLLTOOL_FOR_TARGET="$DLLTOOL_FOR_TARGET"
# All done, install if asked to.
if [ x"$action" = xinstall ] ; then
    make -w install \
        LANGUAGES="c c++" \
        CFLAGS="$CFLAGS" \
        CXXFLAGS="$CXXFLAGS" \
        CC_FOR_BUILD="$CC_FOR_BUILD" \
```

```
        CXX_FOR_BUILD="$CXX_FOR_BUILD" \
        CC="$CC" \
        AR="$AR" \
        RANLIB="$RANLIB" \
        GCC_FOR_TARGET="$CC_FOR_TARGET" \
        CC_FOR_TARGET="$CC_FOR_TARGET" \
        CXX_FOR_TARGET="$CC_FOR_TARGET" \
        AS_FOR_TARGET=$AS_FOR_TARGET \
        LD_FOR_TARGET=$LD_FOR_TARGET \
        AR_FOR_TARGET=$AR_FOR_TARGET \
        NM_FOR_TARGET=$NM_FOR_TARGET \
        RANLIB_FOR_TARGET=$RANLIB_FOR_TARGET \
        DLLTOOL_FOR_TARGET="$DLLTOOL_FOR_TARGET"
fi

# Almost done.  Before the toolchain is usable we need to
# - convert the coff files to .exe's,
# - convert file names to follow MSDOS's 8.3 rules,
# - Change \n to \r\n in text files (like headres).
# The package dosrel-1.0 is set up to do all this.
# See ftp://ftp.objsw.com/pub/crossgcc/dosrel-1.0.tar.gz

exit $?
```

Before the tools are usable, a few things must be done:

- convert the binaries to .exe's
- cope with DOS's 8.3 file name restriction

`ftp://ftp.objsw.com:/pub/crossgcc/dosrel-1.0.tar.gz`contains a set of tools to do this. It works on the 'install tree' created by the above procedure and produces a tar/zip'able tree that is ready to install and use.

Modify the steps in dosrel-1.0/README as follows:

1. Build and install the DOS hosted cross-compiler using the Canadian-Cross method.
2. Configure this directory the same way you configured the DOS cross-compiler. We build in the source directory here. The argument to `--exec-prefix` violates DOS's 8.3 rules but the violation is harmless.eg: `/path/to/binutils-2.9.1/configure --srcdir=. --build=$build host=i386-go32 --target=$target --prefix=$prefix --exec-prefix=$prefix/H-i386-go32 -v`
3. Run '`make dos-tree GO32_STRIP=/path/to/b-sparc-sun-solaris2-x-i386-go32/binutils/strip-new`'.
4. Subdirectory '`dos-tree`' is now ready to copy over to DOS and use. You will need to edit '`set-env.bat`' in the '`bin`' directory to tell GCC and the various pieces where you installed them.

Once the tree is built and installed in MSDOS, you need to create a file called `DJGPP.ENV` that sets up several parameters for the djgpp runtime system. Then you need to set the environment variable `DJGPP` to point to

this file; and you need to set the PATH variable to include the directory where your newly-built compiler resides.

The following may be copied to the file `DJGPP.ENV`:

```
#= djgpp.env, modified for cross-compilation support
#= Don't edit this line unless you move djgpp.env outside
#= of the djgpp installation directory.  If you do move
#= it, set DJDIR to the directory you installed DJGPP in.
#=

DJDIR=%:/>DJGPP%
+USER=dosuser
+TMPDIR=%DJDIR%/tmp
+EMU387=%DJDIR%/bin/emu387.dxe
+LFN=n

[bison]
BISON_HAIRY=%DJDIR%/lib/bison.hai
BISON_SIMPLE=%DJDIR%/lib/bison.sim

[cpp]
CPLUS_INCLUDE_PATH=%/>;CPLUS_INCLUDE_PATH%%DJDIR%/lang/cxx;%DJDIR%/include;%DJDIR%/contrib/grx20/includ
#C_INCLUDE_PATH=%/>;C_INCLUDE_PATH%%DJDIR%/include;%DJDIR%/contrib/grx20/include
#OBJCPLUS_INCLUDE_PATH=%/>;OBJCPLUS_INCLUDE_PATH%%DJDIR%/include;%DJDIR%/lang/objc
#OBJC_INCLUDE_PATH=%/>;OBJC_INCLUDE_PATH%%DJDIR%/include;%DJDIR%/lang/objc

[gcc]
#COMPILER_PATH=%/>;COMPILER_PATH%%DJDIR%/bin
#LIBRARY_PATH=%/>;LIBRARY_PATH%%DJDIR%/lib;%DJDIR%/contrib/grx20/lib
GCC_EXEC_PREFIX=%/>;GCC_EXEC_PREFIX%%DJDIR%/

[gxx]
#COMPILER_PATH=%/>;COMPILER_PATH%%DJDIR%/bin
#LIBRARY_PATH=%/>;LIBRARY_PATH%%DJDIR%/lib;%DJDIR%/contrib/grx20/lib
GCC_EXEC_PREFIX=%/>;GCC_EXEC_PREFIX%%DJDIR%/
[info]
INFOPATH=%/>;INFOPATH%%DJDIR%/info;%DJDIR%/gnu/emacs/info
INFO_COLORS=0x1f.0x31

[emacs]
INFOPATH=%/>;INFOPATH%%DJDIR%/info;%DJDIR%/gnu/emacs/info

[less]
LESSBINFMT=*k<%X>
LESSCHARDEF=8bcccbcc12bc5b95.b127.b
LESS=%LESS% -h5.0.0.7.0$
```

```
[locate]
+LOCATE_PATH=%DJDIR%/lib/locatedb.dat

[ls]
+LS_COLORS=no=00:fi=00:di=36:lb=37;07:cd=40;33;01:ex=32:*.cmd=32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.

[dir]
+LS_COLORS=no=00:fi=00:di=36:lb=37;07:cd=40;33;01:ex=32:*.cmd=32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.

[vdir]
+LS_COLORS=no=00:fi=00:di=36:lb=37;07:cd=40;33;01:ex=32:*.cmd=32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.
```

The following can be copied to a `.bat` file and run; replace '`c:\bar`' with the correct value.

```
@echo off
rem set up path and environment variable for cross-compilation
rem you may want to move this file to a directory that is on your path

rem this batch file assumes that the base directory is "c:\bar"
rem if you install it somewhere else, then modify this to suit.

set DJGPP=c:/bar/djgpp.env
path c:\bar\bin;%PATH%
```

4.10 Disk space requirements

How much disk space is required? Disk space requirements vary depending on the host and target. The source tree occupies about 80MB. Very roughly:

- binutils: 16MB
- gcc: 40MB
- gdb: 20MB
- newlib: 8MB

A `sparc-sunos4.1.4` cross `m68k-coff` toolchain requires about 100MB to build, and about 40MB installed. Note that the programs are not stripped by default. Stripping them will save you a considerable amount of space.

5 Frequently Encountered Problems

5.1 installation problem, cannot exec 'cpp': No such file or directory

This error message usually appears when GCC has been installed in a place other than the one it was configured for. There are two solutions:

- install GCC in the right place. You can find out where GCC was configured to be installed by running '`gcc --print-search-dirs`'. It will print something like this:

```
install: /calvin/dje/rel/H-sparc-sun-solaris2/lib/gcc-lib/sparc-sun-solaris2/2.95.2/
programs: [... omitted ...]
libraries: [... omitted ...]
```

The 'install' line tells you where GCC was configured to be installed (in this case `/calvin/dje/rel/H-sparc-sun-solaris2`).

- Set the environment variable `GCC_EXEC_PREFIX` to the directory where you installed GCC. For example, if you installed GCC in `/home/gcc` (and file `cc1` lives in `/home/gcc/lib/gcc-lib/$target/2.95.2` where `$target` is the `--target` argument that was passed to configure), then set `GCC_EXEC_PREFIX` to `/home/gcc/lib/gcc-lib/`. The trailing '/' is important!

See the GCC info page 'Environment Variables' for more information. e.g.

```
info -f gcc.info -n "Environment Variables"
```

5.2 Assembler errors while building GCC's `enquire` or `libgcc.a`

Assembler errors encountered while building `enquire` or `libgcc.a` are usually caused by GCC (the one you just built) not being able to find the right assembler. Have you installed it in a place where GCC can find it?

Were GCC and Binutils configured with the same `--prefix`/`--exec-prefix` arguments?

If the assembler hasn't been installed, the quickest solution is to create a symbolic link called '`as`' in the GCC build directory that points to the assembler to use.

5.3 Unresolved symbols during linking

The first thing to do is confirm you've included all the necessary libraries. When linking with the GNU linker directly, `libgcc.a` will not be included. `libgcc.a` contains various routines internal to GCC (for example software floating point support or a 32 bit integer multiply on systems without one). Add `-lgcc` as the last argument to GNU ld.

Note that it is not necessary to add `-lgcc` when linking with GCC as it will add the `-lgcc` automatically. To find out how GCC is invoking the linker, try to link with gcc and pass the `-v` option to gcc.

5.4 Where are open, read, write, close, etc. ?

The following is a typical situation people run into when linking their application.

```
/usr/local/m68k-coff/lib/libc.a(sbrkr.o): In function `_sbrk_r':
sbrkr.c:60: undefined reference to `sbrk'
/usr/local/m68k-coff/lib/libc.a(makebuf.o): In function `__smakebuf':
```

```
makebuf.c:93: undefined reference to 'isatty'
/usr/local/m68k-coff/lib/libc.a(filer.o): In function '_open_r':
filer.c:63: undefined reference to 'open'
/usr/local/m68k-coff/lib/libc.a(filer.o): In function '_close_r':
filer.c:100: undefined reference to 'close'
/usr/local/m68k-coff/lib/libc.a(filer.o): In function '_lseek_r':
filer.c:142: undefined reference to 'lseek'
/usr/local/m68k-coff/lib/libc.a(filer.o): In function '_read_r':
filer.c:184: undefined reference to 'read'
/usr/local/m68k-coff/lib/libc.a(filer.o): In function '_write_r':
filer.c:226: undefined reference to 'write'
/usr/local/m68k-coff/lib/libc.a(fstatr.o): In function '_fstat_r':
fstatr.c:61: undefined reference to 'fstat'
```

Depending upon the target, system calls are not built into newlib's `libc.a`. They are too dependent upon the particular target board in use. Libgloss (which comes with newlib net releases) is intended to be the repository of such routines and may either provide them in another library that you must link against or in an object file. For systems that don't have a need for such routines, just stub them out. e.g.

```
int open (char *f, int flags, ...) { errno = ENOSYS; return -1; }
```

etc.

5.5 How do I pass options from GCC to GAS or GLD?

Sometimes one wants to have GCC pass options to the assembler or linker. This is done with the `-Wa` and `-Wl` arguments to GCC respectively. For example, suppose one wanted to pass `-foo` to GAS. This is done by passing `-Wa,-foo` to 'gcc'. And for the linker, use `-Wl,-foo`. Multiple options may be specified either with multiple uses of `-Wa` or `-Wl`, or by separating the arguments with a comma (e.g.`-Wl,-foo,-bar`). If '`-foo`' takes an argument, use commas as in `-Wa,-foo,fooarg`.

5.6 How do I write an interrupt handler in C?

GCC doesn't support writing interrupt handlers (in a general way) because the FSF doesn't believe this is a useful addition to GCC. The frequency of use doesn't justify the additional complexity in GCC. Write a cover function in assembler that calls C code as necessary. Or, you could embed the necessary assembler at the top and bottom of a C function, but getting it right may be tricky, and a few bytes of ROM space will be wasted by the prologue/epilogue that are provided by GCC.

5.7 How do I write assembler code that works with `m68k-aout`'s leading '_' and `m68k-coff`'s lack of leading '_'?

See `config/m68k/lb1sf68.asm` in the GCC source tree. It uses macros that prepend (or leave off) the leading underscore as necessary (and leading '%' for registers).

6 Library Support

6.1 What is `libgcc.a`?

`libgcc.a` is a library internal to GCC. It exists to provide

- subroutines to replace missing hardware functionality (for example, say, 32 bit integer multiply)
- software floating point support for chips that don't have hardware floating point routines (see 6.4 (Software Floating Point Support))
- other routines needed to implement language functionality that GCC doesn't emit inline code for

6.2 How do I force GCC to emit inline code for `memcpy`?

You can't (in general).

Some language functionality requires copying a (pseudo-)arbitrary number of bytes from one place to another (for example, structure assignment in C). If the GCC port for the target doesn't provide the necessary support, GCC emits a call to `memcpy` (or `bcopy`, depending on the target) to do this, and currently there is no option to force GCC to emit inline code. If your C library doesn't provide the routine, you will need to write one yourself. GCC may also emit calls to `memcpy/bcopy` if the object is sufficiently large.

6.3 Why would I choose glibc over newlib (or vice versa)?

There are currently two sources for a C library: Glibc and Newlib. Both have their pluses and minuses.

6.3.1 Intent

Glibc is the GNU libc maintained by the FSF. It is intended to be a complete replacement library for native systems. It includes the routines required by C Standards as well as Posix and other routines.

Newlib is a collection of software from several sources, and was put together by Cygnus for embedded systems. It contains the routines required by the C Standard as well as a math library and misc. other routines. It is not intended to be a complete replacement library for Unix systems (however, newlib is the native library for go32 and cygwin32 systems).

6.3.2 Licensing

Glibc is covered by the LGPL (the GNU Library General Public License). Newlib is a collection of software from several sources, each with their own copyrights, but basically it's a Berkeley style copyright.

6.3.3 Resource Utilization

Glibc, being intended for native Unix environments, does not need to worry about memory usage as much. It is designed to work most efficiently in demand-page-loaded shared library situations. Newlib, being intended for embedded systems, does worry about memory usage (and is more memory-efficient than glibc).

6.4 What if I need software floating point support?

GCC now comes with generic software floating point support for almost all embedded targets.

6.5 Why are several copies of newlib built?

When building `m68k-coff` (and other embedded targets), one will notice several copies of `libgcc.a`, `libc.a`, and `libm.a` being built. Several copies exist so that the correct one can be used when a given option is used. For example, when compiling for the m68000, you do not want to link in a library compiled for the m68020. And so on.

6.5.1 Is there any way to build only the libraries I need?

Yes. Depending upon the target, there should be `--enable` options to allow you to select which libraries you want to build. This varies over time so listing them here might not be useful. The best way to find out is to study the `config-ml.in` script at the top of the newlib source tree.

7 Using GDB for Remote Debugging

The gdb manual has information on how to do remote debugging with the targets that it supports. That is the best place to look. If you have 'info' and the gdb 'info' files installed you can run '`info -f gdb.info -n Remote`' to view the relevant section.

7.1 BDM Support

7.1.1 What is BDM, and why do I want it?

BDM stands for 'Background Debug Mode', and it is a special debug interface on some of Motorola's embedded 68K and Power PC processors. BDM allows an external debug system to control and monitor the operation of theembedded processor through a 10-pin proprietary serial interface, without any software support in the target system and without the use of an In-Circuit Emulator (ICE). Using a low-cost interface cable to your PC's printer port, and the right software on your PC, you can perform most of your debugging directly on your target hardware at very low cost.

7.1.2 What processors have BDM capability?

Since BDM requires some special hardware on the embedded processor, you can't use it with just any old CPU. It is available only on a select set of Motorola's embedded microprocessors:

- all MC68300 devices except the 68302, 68306, 68307, 68322, 68328, and 68356 (specifically, any device that is based upon Motorola's CPU32 processor core)
- all Coldfire devices in the MCF52xx/MCF53xx series
- all MPC500 and MPC800 embedded Power PC processors

7.1.3 What are the capabilities of BDM?

BDM (as implemented in the original CPU32 version) executes a small set of commands over a high-speed, proprietary serial interface that allows your debug host to read and write memory locations; read and write CPU registers; and start, stop, and reset the processor. Breakpoints are implemented by replacing a normal program instruction in your code with a special opcode that forces the processor into BDM.

The ColdFire processors implement the same command set, but have some extra hardware support in the form of a hardware breakpoint register.

The Power PC implementation of BDM is yet more capable. Instead of using a command language, it allows the debug system to feed CPU opcodes directly to the processor core to be executed. In addition, dedicated hardware implements a small number of hardware breakpoints for code execution, and watchpoints for data accesses.

7.1.4 What are the tradeoffs in using BDM for in-circuit debug?

BDM plays off the good old 80/20 relationship; it gives you 80% of the functionality of a full-blown ICE for 20% of the cost.Its major limitations are:

- it cannot track execution in real time; you need a hardware logic analyzer to do this
- for all intents and purposes, you must run your code in RAM in order to be able to set breakpoints. Some derivatives have limited hardware breakpoint capability, but not nearly enough for typical debugging situations.
- On the 68k targets, it can only break on a fetch of a breakpoint instruction. The Power PC devices have some limited support for hardware data-fetch breakpoints.

Having said that, it should be noted that many developers regularly do the majority of their hardware and software debugging without using any other debug system, saving the expensive tools for those really difficult problems which cannot be tracked down without some real-time tools.

7.1.5 What else can BDM be used for?

Here are a couple of the most common applications:

- Flash EPROM programming in-system
- Production line testing and calibration

7.1.6 I am using a PC host system. How do I use BDM via gdb?

The first requirement is an interface cable. There are two options here:

1. Motorola (`http://www.mot-sps.com`) sells a software/hardware package manufactured by P&E Microsystems (`http://www.pemicro.com`) under part number M68ICD32 which includes a cable and a simple assembly-language debugger; list price is $150. This cable is pretty much the standard for the 68k devices.

2. The 'Wiggler' from Macraigor Systems is an interface cable which supports both 68k and Power PC. List price is around $500 (last time I checked). Check our `http://www.macraigor.com`for more info.

The other item is software support for the host system; a low-level driver is required to control these interface cables since the BDM protocol is not based on standard parallel or serial protocols.

7.1.7 683xx BDM resources

- At `ftp://ftp.lpr.e-technik.tu-muenchen.de/pub/bdm/`there is a device driver and patches for gdb to implement source-level debug via BDM for MC683xx processors.
- Another fairly comprehensive driver for linux at `ftp://skatter.usask.ca/pub/eric/BDM-Linux-gdb`
- `http://www.zeecube.com/bdm/index.htm`offers an kernel-mode driver for Windows NT and links to other useful sites relating to BDM drivers.

7.1.8 PowerPC (MPC5xx/MPC8xx) BDM resources

- If you are running on a Win32 operating system (for example with the cygwin system from Cygnus), gdb has built-in support for the Macraigor device driver `wigglers.dll`, which is available for download as part of Macraigor's 'OCD Commander' package.

7.1.9 I'm using a Unix host system. How do I use BDM via gdb?

If you are on an Intel Linux system, there is a driver to implement BDM for 683xx targets; check out 7.1.6 (BDM via gdb on PC host). For other systems, several vendors have reasonably low-cost ICE-type products which interface via BDM and communicate with the host system via RS-232 serial ports, or Ethernet.

7.1.10 Where can I learn more about BDM?

BDM is fully documented in Motorola's reference manuals, document no. CPU32RM/AD for the 68k devices and MPC860UM/AD for Power PC. These documents may be ordered for free from Motorola's web site, `http://design-net.com/home/lit\_ord.html`.

8 S Records, etc.

8.1 What are S Records?

From the `comp.sys.m68k`FAQ (`http://www.hitex.com/automation/FAQ/m68k`):

> S-Records are Ascii characters in a protocol developed by Motorola and is used to transfer data and program code to and from host computers or to store such information. Details of this protocol have been archived at `ftp://nyquist.ee.ualberta.ca/pub/motorola/general/s\_record.zip`, and also at `ftp://ftp.luth.se/pub/misc/motorola/faq/s\_record.gz`.

8.2 How do I use objcopy to generate S Records?

objcopy can be used to convert an existing file to S Records. Use `--output-target=srec`. or `-O srec`.

8.3 How do I use the linker to generate S Records?

Pass `-oformat=srec` to GLD.

9 Target Specific Info

9.1 What embedded targets are supported?

Here is a partial list. Targets are being added all the time so this list will always be out of date.

They are named by the argument one would use with the `--target=` option of '`configure`'.

- a29k-amd-udi
- arm-aout
- arm-coff
- h8300-hms
- hppa1.1-hp-proelf
- i386-aout
- i386-coff
- i386-elf
- i960-coff
- m32r-elf
- m68k-aout
- m68k-coff
- mips-ecoff
- mips-elf
- mips64-elf
- powerpc-eabi
- sh-hms
- sparc-aout
- sparclite-aout
- sparclite-coff
- sparc64-elf

10 Operating systems for embedded systems

10.1 RTEMS

RTEMS is a freely available real-time executive with multiprocessor capabilities. RTEMS was designed to provide performance and capabilities similar to those of the best commercial executives. The directive execution times and other critical performance measures such as interrupt latency are comparable to those of commercial executives. It was developed by On-Line Applications Research Corporation (OAR) under contract to the U.S. Army Missile Command.

RTEMS includes support for multiple APIs. Currently RTEMS supports an RTEID based API which is similar to pSOS+ and a POSIX threads API. RTEMS has a number of advanced real-time features including optional rate monotonic scheduling support, binary semapahores with priority inheritance, and watchdog timer functions.

RTEMS includes a port of the FreeBSD TCP/IP stack that has very high performance on very modest hardware. A remote debug server is supported on some targets that allows debugging across the network.

RTEMS provides a rich run-time environment with a reentrant C library, POSIX 1003.1b support, optional debug aids like stack overflow and heap integrity checking, a C++ interface, Ada95 bindings, and much more.

RTEMS is built using GNU autoconf and can be built on any host supporting the GNU tools including Linux, FreeBSD, NetBSD, Solaris, and MS-Windows to name but a few. The current RTEMS release supports a the following CPU families:

- Motorola m680x0, m683xx, and ColdFire
- Motorola and IBM PowerPC (4xx, 6xx, 7xx, and 8xx)
- Hitachi SH
- Intel i386, i486, Pentium, and i960
- SPARC
- MIPS
- HP PA-RISC
- AMD A29K

There are over 35 Board Support Packages included with RTEMS including BSPs for the SPARC and PowerPC simulators included with gdb. There are numerous BSPs for CPUs targetting the embedded market such as the Motorola m683xx and mpc8xx series which support the on-CPU peripherals.

WWW:

`http://www.OARcorp.com`

Email:

`rtems-info@OARcorp.com`or

`JoelSherrill(joel@OARcorp.com)`

10.2 eCOS

eCOS stands for 'Embedded Cygnus Operating System'. All of the following information is from the Cygnus web site.

> The embedded Cygnus operating system (eCos) is an open-source, configurable, portable, and royalty-free embedded RTOS. The system comes with everything necessary to develop eCos based applications, including tools, documentation and complete sources. All eCos runtime source code is distributed under the Cygnus eCos Public License (CEPL), a derivative of the Netscape Public License.
>
> This release is a 'Technology Release'. It provides all the functionality needed for a wide variety of embedded applications, and it has undergone extensive testing. However at this stage it is only available for a limited number of target architectures, and there is lots of additional functionality that can and will be added. We have chosen to release the system at this stage because it can already be of great benefit to many embedded system developers, and to encourage the development of an eCos open source user community on the net which will help to move the system forwards.

10.2.1 eCos Functionality

- Hardware Abstraction Layer (HAL)
- Real-time kernel
- Interrupt handling
- Exception handling
- Choice of schedulers
- Thread support
- Rich set of synchronization primitives
- Timers, counters and alarms
- Choice of memory allocators
- Debug and instrumentation support
- uITRON 3.0 compatible API layered on top of the basic kernel
- ISO C library

10.2.2 Supported Targets

- Matsushita MN10300 processor, stdeval1 evaluation board and simulator
- Toshiba TX39 processor, jmr3904 evaluation board and simulator
- PowerPC MPC860 processor, `Cogent`evaluation board and `psim`simulator. This target should be considered a beta.

10.2.3 Supported Hosts

The main supported host is Windows NT on an x86-based host. The system has also been tested on Red Hat 5.1 Linux using an x86-based host, and on both Windows 95 and 98, but these hosts should also be considered a beta. The system has not been tested on any other host platform.

10.2.4 Host Software

The system comes with full support for manual configuration of the system on all host platforms. In addition Windows users get a graphical configuration tool which simplifies the process of configuring a system and validating the resulting system. This tool is not yet available for Linux users. It is likely that new host tools, including future versions of the graphical configuration tool, will only be available as part of the Developer Kit product.

The system comes with a full set of tools needed for embedded systems development. This includes the compilers gcc and g++, the assembler and linker, the debugger gdb, and simulators for all the target architectures.

WWW:

`http://www.cygnus.com/ecos`and

`http://sourceware.cygnus.com/ecos`

E-mail:

`ecos-info@cygnus.com`

11 How to get help

11.1 General questions

If the topic is related to using the tools in a cross-compiler environment then this list is an excellent place to look for help. If you've found a bug in a particular piece of software, it is best to report the bug to the appropriate list (e.g. bug-gcc) rather than this list.

Be careful about reporting bugs associated with patches you have applied to GCC (or any GNU software). The FSF has not sanctioned these patches and shouldn't have to respond to bug reports for such patches. Sometimes it is hard to tell, in which case use your best judgement. When sending bug reports to bug-gcc, always mention all patches applied.

11.2 Online documentation

Documentation for the GNU tools as packaged by Cygnus may be found in `http://www.cygnus.com/pubs/gnupro`. Documentation for the FSF releases, along with other stuff useful to embedded developers, may be found at `http://www.objsw.com/docs`.

11.3 Web sites

Binutils:

`http://sourceware.cygnus.com/binutils`

GCC/G++:

`http://egcs.cygnus.com/`

Glibc:

`http://www.gnu.org/software/libc`

Gdb:

`http://sourceware.cygnus.com/gdb`

Newlib:

`http://sourceware.cygnus.com/newlib`

11.4 Bug reporting

Here is a list of the tools and their bug reporting addresses:

Binutils:

`bug-gnu-utils@gnu.org`

GCC:

`bug-gcc@gnu.org`

G++:

`bug-g++@gnu.org`

Glibc:

`bug-glibc@gnu.org`

GDB:

`bug-gdb@gnu.org`

Newlib:

`newlib@sourceware.cygnus.com`

When reporting a bug, please read the documentation that comes with the software to find out how to report a bug. If you've installed 'info' and the GCC info files, '`info -f gcc.info -n Bugs`' should zip you right to the page you need to read for GCC. Failing that, always include the following information:

- version number (e.g. gcc-2.95.2)
- configuration (the arguments you passed to '`configure`')
- type of system you are compiling on (e.g. `i386-linux`, `sparc-solaris2`); ideally this is the result of the '`config.guess`' program that is included in the source tree

- (in the case of gas, gcc, glibc, and newlib) preprocessed source that triggers the bug. Preprocessed source can either be obtained by passing `-E` to GCC and sending the output, or by passing `-save-temps` to GCC and sending the resulting .i file (or .ii file in the case of g++). It is generally much better to include the preprocessed source rather than say 'GCC can't compile file foo.c from program Bar version 4.2'.
- complete list of options passed to the program (e.g. `gcc -g -O2` ...).
- the text of the error message (as output by the program, do not edit it for brevity)

11.5 Other mailing lists, web sites, newsgroups, etc.

11.5.1 djgpp

Mailing list:

`djgpp@delorie.com`

Web site:

`http://www.delorie.com/djgpp/`

Newsgroup:

`comp.os.msdos.djgpp`

11.5.2 cygwin

Mailing list:

`cygwin@sourceware.cygnus.com`

Web site:

`http://sourceware.cygnus.com/cygwin`

11.5.3 miscellaneous newsgroups

- `comp.arch.embedded`
- `comp.sys.m68k`

FIXME: Others?

12 Glossary

Canadian Cross

This is a cross compiler where build machine != host machine. See 4.9 (What is a Canadian Cross?)for more information.

cross compiler

This is a compiler that builds programs for a machine different than the one the compiler is run on. For example, an `i386-linux` cross `m68k-coff` compiler is one where GCC runs on an `i386-linux` machine and compiles programs for running on `m68k-coff` machines.

build machine

This is the machine the toolchain is being built on. In the case of a Canadian Cross, it is not the same as the host machine.

host machine

This is the machine the toolchain programs (GCC, GAS, etc.) will run on. In the case of a cross compiler, this is not the same as the target machine.

native compiler

This is a compiler where host == target.

target machine

This is the machine the toolchain builds programs for.

FIXME: Others?

13 Contributors

The following people have contributed to this FAQ. Thanks!

- Steve Baldwin, `baldwins@stimpy.fp.co.nz`
- Brian Carlstrom, `bdc@ai.mit.edu`
- Steve Chamberlain, `sac@cygnus.com`
- DJ Delorie, `dj@delorie.com`
- Doug Evans, `devans@cygnus.com`
- John Fisher, `johnf@tsd.neca.nec.com.au`
- David Golombek, `daveg@maker.com`
- Larry Langerholc, `Larry.Langerholc@icn.siemens.com`
- Curt Moffitt, `cmoffitt@turnstone.com`
- Jason Molenda, `crash@cygnus.com`
- Josh Pincus, `pncu\_ss@bridge.cc.rochester.edu`
- K. Richard Pixley, `rich@ono.noir.com`
- Ken Raeburn, `raeburn@cygnus.com`
- Tim Mooney, `mooney@dogbert.cc.ndsu.NoDak.edu`

- Rob Savoye, `rob@cygnus.com`
- Steven Schram, `schram@invocon.com`
- Stan Shebs, `shebs@cygnus.com`
- Joel Sherrill, `joel@oarcorp.com`
- Ian Taylor, `ian@cygnus.com`

If I've left anyone out, please let me know.