GCC for Embedded Engineers

Read along to understand how GCC works, find out what all those other programs in the toolchain directory do, and learn some tips and tricks to become more comfortable with most indispensable tool in your project.

GENE SALLY

GCC, the GNU Compiler Collection, is a tool used by nearly every embedded engineer, even those who don't target Linux. In release since 1987, supporting every processor known to man, GCC is a juggernaut of software engineering that, because of its ubiquity and ease of use, doesn't get the admiration it deserves.

When used in an embedded project, GCC capably does another trick, cross-compilation, without complaint. Simply invoke the compiler and the right things will happen. Under the covers, GCC is a very complex tool with lots of knobs to turn to fine-tune the compilation and linking process; this article looks at how to build a GCC cross-compiler, examines the process that GCC uses to compile a program and shares some productivity-boosting tips and tricks.

Building a Cross-Compiler

When starting an embedded project, the first tool needed is a cross-compiler, a compiler that generates code intended to work on a machine different from the one on which the code generation occurred. Sometimes, it's possible to obtain a prebuilt cross-compiler (from a commercial or noncommercial source), short-circuiting the need to build from source; however, some projects require that all tools must be re-creatable from source. No matter why GCC needs to be built, there are several different approaches to building a cross-compiler.

Quite possibly the easiest way is by using the crosstool Project, created by Dan Kegel and hosted at **www.kegel.com/crosstool**. Using this project involves downloading the source code and making one of the presupplied files feed the right parameters into the script that builds the compiler. The matrix of supported platforms and software versions can be found at **www.kegel.com/crosstool/crosstool-0.43/buildlogs**, and choosing something that's marked as working will yield a compiler in a few hours. crosstool will download the right software, even the patches, necessary to make the software work on the target platform. However, if the project requires support for an alternate C library, crosstool becomes more difficult to use.

Because many developers want to use uClibc, a smaller implementation of the C library, it's fortunate that this project has something similar to crosstool, called buildroot, located at **buildroot.uclibc.net**. As a bonus, buildroot, along with building a cross-compiler, also can be used to build a root filesystem for the board based on the related BusyBox Project. The user configures a buildroot run using a process similar to that of the kernel configuration to ready the build. This project doesn't have a chart of known working configurations like crosstool, so finding a working configuration can be difficult.

Finally, for the type of person who doesn't like the idea of wading through somebody else's build scripts when things don't work, building a cross-complier by hand isn't as daunting a process as one would expect. The following steps outline the process, where $TARGET is the target processor and $INSTALLAT is the directory where the compiler will reside after being built:

1. Download and build binutils:

```
$ tar xzf binutils-<version>.tar.gz
$./binutils-<version>/configure --target=$TARGET --prefix=$INSTALLAT
$ make ; make install
```

2. Copy the include and asm from the board's kernel to the installation directory:

```
$ mkdir $INSTALLAT/include
$ cp -rvL $KERNEL/include/linux $KERNEL/include/asm $INSTALLAT/include
```

3. Download and build bootstrap GCC. At this point, it's best to build the bootstrap GCC in its own directory and not the directory where it has been unpacked:

```
$ tar xzf gcc-<version>.tar.gz
$ mkdir ~/$TARGET-gcc ; cd ~/$TARGET-gcc
$../gcc-<version>/configure --target=$TARGET --prefix=$INSTALLAT
--with-headers=$INSTALLAT/include --enable-languages="c" -Dinhibit_libc
$ make all ; make install
```

4. Download and build glibc (or alternate libc) with the bootstrap compiler. Like GCC, the build of the library works best when you configure and make outside the source tree:

```
$ tar xzf glibc-<version> --target=$TARGET --prefix=$INSTALLAT
--enable-add-ons --disable-sanity-checks
$ CC=$INSTALLAT/bin/$TARGET-gcc make
$ CC=$INSTALLAT/bin/$TARGET-gcc make install
```

5. Build the final GCC. The bootstrap compiler was built to build the C library. Now, GCC can be built to use the cross-compiled C library when building its own programs:

```
$ cd ~/$TARGET-gcc
$../gcc-<version>/configure --target=$TARGET --prefix=$INSTALLAT
--with-headers=$INSTALLAT/include --enable-languages="c"
$ make all ; make install
```

At the end of this process, the newly built cross-compiler will be at $INSTALLAT/bin. An oft-used strategy for those needing a specially configured GCC is to use crosstool or buildroot to download and patch the source files and then interrupt the process. At this point, the user applies additional patches and builds the components with the desired configuration settings.

Before leaving this section, there's a frequently asked question from embedded engineers targeting Pentium machines doing development on desktops that are essentially same. In this case, is a cross-compiler necessary? The answer is yes. Building a cross-compiler for this configuration insulates the build environment and library dependencies from the development machine that happened to be used to build the source code. Because desktop systems can change revisions several times a year, and not all team members may be using the same version, having a consistent environment for compiling the embedded project is essential to eliminate the possibility of build configuration-related defects.

The Toolchain: More Than Just a Compiler

The collection of programs necessary to compile and link an application is called the toolchain, and GCC, the compiler, is only one part. A complete toolchain consists of three separate parts: binutils, language-specific standard libraries and the compiler. Notably absent is the debugger, which is frequently supplied with the toolchain but is not a necessary component.

binutils

binutils (binary utilities), performs the grunt work of manipulating files in a way that's appropriate for the target machine. Key parts of the toolchain, such as the linker and assembler, reside in the binutils Project and aren't part of the GCC Project.

Hidden inside the binutils Project is another nifty bit of software, the BFD library, which technically is a separate project. The BFD, Binary Descriptor Library (the actual acronym unpacks to something too bawdy for this publication), provides an abstract, consistent interface to object files, such as handling details like address relocation, symbol translation and byte order. Because of the features supplied by BFD, most tools that need to read or manipulate binaries for target reside in the binutils Project to best take advantage of what BFD has to offer.

For the record, binutils contains the following programs:

- addr2line: given a binary with debugging information and an address, returns the line and file of that address.
- ar: a program for creating code archives that are a collection of object files.
- c++filt: demangles symbols. With classes and overloading, the linker can't depend on the underlying language to provide unique symbol names. c++filt will turn _ZN5pointC1ERKS_ into something readable. A godsend when debugging.
- gprof: produces reports based on data collected when running code with profiling enabled.
- nlmconv: converts an object file into a Netware Loadable Module (NLM). If you've ever worked with NLMs, you probably did so with your collar turned up and cringed when seeing ABEND on your terminal. It's noted here because nlmconv is rarely, if ever, distributed with a toolchain.
- nm: given an object file, lists symbols such as those in the public section.
- objcopy: translates a file from one format to another, used in the embedded file to generate S-Records from ELF binaries.
- objdump/readelf: reads and prints out information from a binary file. readelf performs the same function; however, it can work only with ELF-formatted files.
- ranlib: a complement to ar. Generates an index of the public symbols in an archive to speed link time. Users can get the same effect by using ar -s.
- size: prints out the size of various components of a binary file.
- strings: extracts the strings from a binary, performing correct target host byte order translation. It's frequently used as the slacker's way of seeing what libraries a binary links to, as ldd doesn't work for cross-compiled programs: `strings <binary> | grep lib`.
- strip: removes symbols or sections, typically debugging information, from files.

Table 1. Pros and Cons of Most Frequently Used C Libraries

Library	Pros	Cons
glibc	The canonical C library; contains the greatest amount of support for all C features; very portable; support for the widest number of architectures.	Size; configurability; can be hard to cross-build.
uClibc	Small (but not the smallest); very configurable; widely used; active development team and community.	Not well supported on all architectures; handles only UTF-8 multibyte characters.
DietLibC	Small, small, small; excellent support for ARM and MIPS.	Least functionality; no dynamic linking; documentation.
NewLib	Well supported by Red Hat; best support for math functions; great documentation.	Smallish community; not updated frequently.

Finally, for the type of person who doesn't like the idea of wading through somebody else's build scripts when things don't work, building a cross-complier by hand isn't as daunting a process as one would expect.

C Library

The C language specification contains only 32 keywords, give or take a few, depending on the compiler's implementation of the language. Like C, most languages have the concept of a standard library supplying common operations, such as string manipulation, and an interface to the filesystem and memory. The majority of the programming that happens in C involves interacting with the C library. As a result, much of the code in the project isn't written by the engineers, but rather is supplied by the standard libraries. Picking a standard library that has been designed to be small can have a drastic impact on the final size of the project.

Most embedded engineers opt for using a C library other than the standard GNU C Library, otherwise known as glibc, to conserve resources. glibc was designed for portability and compatibility, and as such, it contains code for cases not encountered or that can be sacrificed on an embedded system. One example is the lack of binary compatibility between releases of the library. Although glibc rarely breaks an interface once published, embedded standard libraries do so without any qualms.

Table 1 outlines the most frequently used C libraries, with the pros and cons of each.

Preprocessor and Compiler

These components perform only a small slice of the work necessary to produce an executable. The preprocessor, for languages that support such a concept, runs before the compiler proper, performing text transformations before the compiler transforms the input into machine code for the target. During the compilation process, the compiler performs optimizations as specified by the user and produces a parse tree. The parse tree is translated into assembler code, and the assembler uses that input to make an object file. If the user wants to produce an executable binary, the object file is then passed to the linker to produce an executable.

How It All Fits Together

After looking at all the components in a toolchain, the following section steps through the process GCC takes when compiling C source files into a binary. The process starts by invoking GCC with the files to be compiled and a parameter specifying output to be stored to thebinary:

```
armv5l-linux-gcc file1.c file2.c -o thebinary
```

GCC is actually a driver program that invokes the underlying compiler and binutils to produce the final executable. By looking at the extension of the input file and using the rules built in to the compiler, GCC determines what programs to run in what order to build the output. To see what happens in order to compile the file, add the -### parameter:

```
armv5l-linux-gcc -### file1.c file2.c -o thebinary
```

This produces virtual reams of output on the console. Much of the output has been clipped, saving untold virtual trees, to make it more readable for this example. The first information that appears describes the version of the compiler and how it was built—very important information when queried "was GCC built with thumb-interworking disabled?"

```
Target: armv5l-linux
Configured with: <the contents of a autoconf command line>
Thread model: posix
gcc version 4.1.0 20060304 (TimeSys 4.1.0-3)
```

After outputting the state of the tool, the compilation process starts. Each source file is compiled with the cc1 compiler, the "real" compiler for the target architecture. When GCC was compiled, it was configured to pass certain parameters to cc1:

```
"/opt/timesys/toolchains/armv5l-linux/libexec/gcc/
➥armv5l-linux/4.1.0/cc1.exe" "-quiet" "file1.c"
 ➥"-quiet" "-dumpbase" "file1.c" "-mcpu=xscale"
 ➥"-mfloat-abi=soft" "-auxbase" "file1" "-o"
 ➥"/cygdrive/c/DOCUME~1/GENESA~1.TIM/LOCALS~1/Temp/ccC39DVR.s"
```

Now the assembler takes over and turns the file into object code:

```
"/opt/timesys/toolchains/armv5l-linux/lib/gcc/
➥armv5l-linux/4.1.0/../../../../armv5l-linux/bin/as.exe"
 ➥"-mcpu=xscale" "-mfloat-abi=soft" "-o"
 ➥"/cygdrive/c/DOCUME~1/GENESA~1.TIM/LOCALS~1/Temp/ccm4aB3B.o"
 ➥"/cygdrive/c/DOCUME~1/GENESA~1.TIM/LOCALS~1/Temp/ccC39DVR.s"
```

The same thing happens for the next file on the command line, file2.c. The command lines are the same as those for file1.c, but with different input and output filenames.

After compilation, collect2 performs a linking step and looks for initialization functions (called constructor functions, but not in the object-oriented sense) called before the "main" section of the program. collect2 gathers these functions together, creates a temporary source file, compiles it and links that to the rest of the program:

```
"/opt/timesys/toolchains/armv5l-linux/libexec/gcc/
➥armv5l-linux/4.1.0/collect2.exe" "--eh-frame-hdr"
 ➥"-dynamic-linker" "/lib/ld-linux.so.2" "-X" "-m"
 ➥"armelf_linux" "-p" "-o" "binary" "/opt/timesys/
➥toolchains/armv5l-linux/lib/gcc/armv5l-linux/
➥4.1.0/../../../../armv5l-linux/lib/crt1.o"
 ➥"/opt/timesys/toolchains/armv5l-linux/lib/gcc/
➥armv5l-linux/4.1.0/../../../../armv5l-linux/lib/crti.o"
 ➥"/opt/timesys/toolchains/armv5l-linux/lib/gcc/
```

```
➥armv5l-linux/4.1.0/crtbegin.o"
 ➥"-L/opt/timesys/toolchains/armv5l-linux/lib/
➥gcc/armv5l-linux/4.1.0" "-L/opt/timesys/
➥toolchains/armv5l-linux/lib/gcc/armv5l-linux/
➥4.1.0/../../../../armv5l-linux/lib"
 ➥"/cygdrive/c/DOCUME~1/GENESA~1.TIM/LOCALS~1/
➥Temp/ccm4aB3B.o" "/cygdrive/c/DOCUME~1/
➥GENESA~1.TIM/LOCALS~1/Temp/cc60Td3s.o"
 ➥"-lgcc" "--as-needed" "-lgcc_s" "--no-as-needed"
 ➥"-lc" "-lgcc" "--as-needed" "-lgcc_s" "--no-as-needed"
 ➥"/opt/timesys/toolchains/armv5l-linux/lib/
➥gcc/armv5l-linux/4.1.0/crtend.o" "/opt/timesys/
➥toolchains/armv5l-linux/lib/gcc/armv5l-linux/
➥4.1.0/../../../../armv5l-linux/lib/crtn.o"
```

There are some other nifty things in here that warrant pointing out:

1. Here's the option that specifies the dynamic linker to invoke when running the program on the target platform:

```
"-dynamic-linker" "/lib/ld-linux.so.2"
```

On Linux platforms, dynamically linked programs actually load by running a dynamic loader, making themselves a parameter of the linker, which does the work of loading the libraries into memory and fixing up the references. If this program isn't in the same place on the target machine, the program will fail to run with an "unable to execute program" error message. A misplaced linker on the target ensnares every embedded developer at least once.

2. These files contain the code before the programmer's entry point (typically main, but you can change that too) and handle things like initialization of globals, opening the standard file handles, making that nice array of parameters and other housekeeping functions:

- crtbegin.o
- crt1.o
- crti.o

3. Likewise, these files contain the code after the last return, such as closing files and other housekeeping work. Like the prior items, these are cross-compiled during the GCC build:

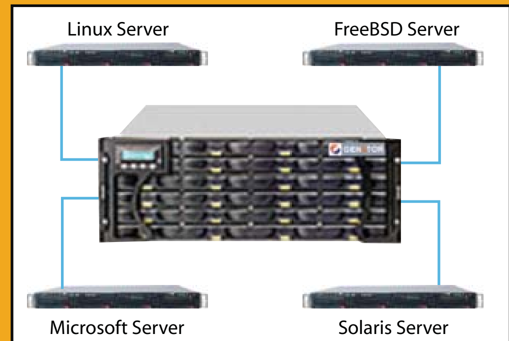
- crtend.o
- crtn.o

And, that's it! At the end of this process, the output is a program ready for execution on the target platform.

The spec File

Recall that GCC is a driver program that knows what program to invoke to build a certain output, which begs the question, "How does it know that?" This information that was built in to GCC when it was built is kept in the "specs". To see the specs, run GCC with the -dumpspecs parameters:

Linux - FreeBSD - x86 Solaris - MS etc.



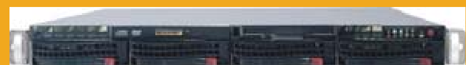
GENSTOR STORAGE SOLUTIONS:

- Storage options - FC to SATA/SAS, FC to FC SAS to SAS/SATA, SCSI to SATA, SCSI to SCSI
- Exceptional Performance with Proven Reliability
- 24 TB in 4U with easy upgrade path
- Host Servers and Storage comes Pre-Configured with heterogeneous OS- Linux, * BSD, Solaris Microsoft etc.
- Fully redundant Storage solutions

Proven technology. Proven reliability

When you can't afford to take chances with your business data or productivity, rely on a GS-1245 Server powered by the Intel® Xeon® Processors.

Quad Core Woodcrest



2 Nodes & Up to 16 Cores - in 1U

Ideal for high density clustering in standard 1U form factor. Upto 16 Cores for high CPU needs. Easy to configure failover nodes.

Features:

- 1U rack-optimized chassis (1.75in.)
- Up to 2 Quad Core Intel ® Xeon ® Woodcrest per Node with 1333 MHz system bus
- Up to 16 Woodcrest Cores Per 1U rackspace

Servers : : Storage : : Appliances

Genstor Systems, Inc.
780 Montague Express. #604
San Jose, CA 95131
www.genstor.com
Email: sales@genstor.com
Phone: 1-877-25 SERVER
1-408-383-0120

Intel®, Intel® Xeon® Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries

```
armv5l-linux-gcc -dumpspecs
```

The console will fill with a few hundred lines of output. The spec file format evolved over years of development, and it's easier for the computer to read than for a person. Each line contains instructions for what parameters to use for a given tool. From the prior example, consider the command line for the assembler (with the path names removed for readability):

```
"<path>/as.exe" "-mcpu=xscale" "-mfloat-abi=soft"
➥"-o" "<temppath>/ccm4aB3B.o" "<temppath>/ccC39DVR.s"
```

The compiler has the following in the specs for the assembler:

```
*asm:
%{mbig-endian:-EB} %{mlittle-endian:-EL} %{mcpu=*:-mcpu=%*}
➥%{march=*:-march=%*} %{mapcs-*:-mapcs-%*}
➥%(subtarget_asm_float_spec)
➥%{mthumb-interwork:-mthumb-interwork}
➥%{msoft-float:-mfloat-abi=soft}
➥%{mhard-float:-mfloat-abi=hard} %{mfloat-abi=*}
➥%{mfpu=*} %(subtarget_extra_asm_spec)
```

This line uses some familiar constructs explained below. Adequately discussing the minutiae of the spec file would require an article series in itself.

- *asm: this line tells GCC the following line will override the internal specification for the asm tool.
- %{mbig-endian:-EB}: the pattern %{symbol:parameter} means if a symbol was passed to GCC, replace it with parameter; otherwise, this expands to a null string. In our example, the parameter -mfloat-abi=soft was added this way.
- %(subtarget_extra_asm_spec): evaluate the spec string %(specname). This may result in an empty string, as it did in our case.

Most users don't need to modify the spec file for their compiler; however, frequently engineers who inherit a project need to have GCC recognize nonstandard extensions for files. For example, assembler source files may have the extension, .arm; in this case, GCC won't know what to execute, as it doesn't have a rule for that file extension. In this case, you can create a spec file containing the following:

```
.arm:
@asm
```

and use the -specs=<file> to pass that to GCC, so that it will know how to handle files with the .arm extension. The spec file on the command line will be added to the internal spec file after it has been processed.

Tips and Tricks of the Trade

The following tips and tricks should be, if they haven't already, stashed on the crib sheet of engineers who work with GCC.

Force GCC to use an alternate C library:

```
armv5l-linux-gcc -nostdlib -nostdinc -isystem
➥<path to header files> -L<path to c library>
➥-l <c library file>
```

This tells GCC to ignore everything it knows about where to find header files and libraries and instead uses what you tell it. Most alternate C libraries provide a script that performs this function; however, some projects can't use the wrapper scripts, and other times, when experimenting with several versions of a library, the flexibility and control of specifying this information directly is necessary.

Mixed assembler/source output:

```
armv5l-linux-gcc -g program.c -o binary-program
armv5l-linux-objdump -S binary-program
```

This is the best way to see exactly what GCC generated in relation to the input code. Doing the compilation with several different optimization settings shows what the compiler did for the given optimization. Because embedded development pushes the processor-support envelope, being able to see the generated assembler code can be instrumental in proving a defect in GCC's support for that processor. In addition, engineers can use this to validate that the proper instructions are generated when specifying processor-specific optimizations.

List predefined macros:

```
armv5l-linux-gcc -E -dM - < /dev/null
```

An invaluable tool for doing a port, this makes clear what GCC

Resources

uClibc, a replacement for the GNU C Library, optimized for size: **www.uclibc.org**.

dietlibc, another replacement for GNU C, the smallest of the group: **www.fefe.de/dietlibc**.

NewLib, a Red Hat-supported project for a minimal C library: **sourceware.org/newlib**.

GCC Internals—information about the guts and construction of GCC; it's very well written and a great guide for those curious about how GCC works: **gcc.gnu.org/onlinedocs/gccint**.

binutils—architecture-specific tools that smooth the way for development: **www.gnu.org/software/binutils**.

`info gcc`, from your command line, provides in-depth information about end-user-related aspects of GCC.

crosstool, a tool for building GCC cross-compilers, now the canonical way for doing so, is very easy to use: **www.uclibc.org**.

The Definitive Guide to GCC by Bill von Hagen—a great book covering all aspects of how to use GCC.

macros will be set automatically and the value. This will show not only the standard macros, but also all the ones set for the target architecture. Keeping this output and comparing it to a newer version of GCC can save hours of work when code fails to compile or run due to changes.

List dependencies:

```
armv5l-linux-gcc -M program.c
```

Formally, this command creates a separate make rule for each file on the command line showing all dependencies. The output is indispensable when trying to track down problems related to what header files a source file is using and tracking down problems related to forcing GCC to use an alternate C library. Deeply nested header files are both unavoidable and incredibly useful in any nontrivial C project and can consume hours when trying to debug. Using -MM instead of -M will show only nonsystem dependencies—useful noise reduction when the problem resides in the project files alone.

Show internal steps:

```
armv5l-linux-gcc -### program.c
```

This article already uses this command to make GCC show what steps occur internally to build a program. When a program isn't compiling or linking properly, using -### is the fastest route to see what GCC is doing. Each command is on its own line and can be run individually, so:

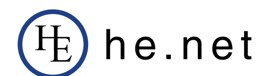
```
armv5l-linux-gcc -### program.c &> compile-commands
```

will produce a file compile—commands that the user can mark as executable and run a line at a time to pinpoint the exact cause of a problem.

Wrapping Up

GCC is a deceptively powerful, complex tool. The developers have created software that "does the right thing" with minimal information from the user. Because it works so well, users frequently forget to spend the time to learn about GCC's capabilities. This article scratches the surface; the best advice is to read the documentation and invest a little time each day to learn how this tool always can do more than expected.■

Gene Sally has been working with all facets of embedded Linux for the last seven years and is cohost of LinuxLink Radio, the most popular embedded Linux podcast. Gene can be reached at gene.sally@timesys.com.



IP Transit
Gigabit Ethernet

Run BGP+IPv6+IPv4

$5/Mbps

Colocation Full Cabinet

Holds up to 42 1U servers

$400/month

Full 100 Mbps Port

Full Duplex

$2,000/month

Order Today!

email sales@he.net or call 510.580.4190

he.net/ip_transit.html