# An Introduction to the GNU Compiler and Linker

William Gatliff

## Table of Contents

## Overview

Despite their reputations as workstation application development tools, the GNU compiler and linker excel at producing high-quality executables for embedded targets. The reason is only partly because an increasing number of embedded systems are based on the same 32-bit processors found in some desktop workstations; it is mostly because the diverse, high-end workstation environment demands flexible and powerful tools, and such tools can also be used to make great embedded systems.

This paper describes the features of the GNU compiler and linker that are most important for embedded developers. It begins with a brief overview of the tools themselves and some of their most useful command line options, then covers the compiler's syntax extensions, in particular its inline assembly language support. It then introduces the linker's command language, and concludes with the procedure used to build the tools from source code.

## The GNU Compiler Collection (gcc)

The GNU Compiler Collection, **gcc**, can compile programs written in C, C++, Java and several other languages. It provides many useful command line options and syntax extensions, and also serves as a powerful frontend for the GNU linker, **ld**.

## Command line options

Gcc supports a large list of command line options. In fact, there are no less than thirteen *categories* of options to choose from! The following is a list of the most important and immediately useful ones for embedded development; see gcc's online documentation for the rest.

### The `-v` option

This option tells gcc to print all the commands it runs during compilation. It also causes gcc to emit internal version data, and other useful troubleshooting information.

### The `-g` option

This command tells gcc to include debugging information in its output files. It is necessary if you intend to use the GNU debugger, **gdb**, to debug the application.

### The `-c` option

This command tells gcc to stop after creating an object file.

### The `-S` and `-Wa` options

The `-S` option tells gcc to stop after translating a source file into assembly language, before the assembler is invoked. The output file is called `<filename>.s`.

To get an annotated assembly language listing, with intermixed source and assembly language code, use the following commands instead of `-S`:

```
$ <targetname>-gcc -g -c -Wa,-alh,-L <filename>
```

This statement uses the `-Wa` command to pass two options to the assembler: `-alh` (produce an annotated source listing), and `-L` (retain information about local variables in the listing). For more information on assembler options, see the assembler's online documentation.

Annotated assembly listings can be confusing when aggressive optimizations are requested, because sections of code may get moved or deleted during the optimization process. Work with unoptimized listings whenever possible.

## The `-Wall` option

Gcc supports a lot of options for warning message generation. In fact, its syntax checking is so complete that in many cases a separate source code scanning utility like **lint** is unnecessary. The `-Wall` option turns on all of gcc's most popular warning settings, of which there are many.

## The `-Wcast-align` option

This option causes gcc to issue a warning whenever a pointer cast can create alignment issues. This option is not enabled by `-Wall`.

Many microprocessors can only access multibyte values on specific address boundaries. Casting a character pointer to an integer pointer in such architectures risks a misaligned data access when the pointer is dereferenced, because a character isn't necessarily placed on the same alignment boundaries as an integer. The `-Wcast-align` option causes gcc to issue a warning when a cast that risks an alignment issue is detected.

## The `-Wsign-compare` option

This option causes gcc to issue a warning when a comparison between signed and unsigned values could produce an incorrect result when the signed value is converted to unsigned. ANSI C requires such transformations during comparisons. This option is not enabled by `-Wall`.

## The `-Wconversion` option

Similar to the `-Wsign-compare` option, `-Wconversion` tells gcc to issue a warning if a negative integer constant expression is implicitly converted to an unsigned type. Statements that produce such transformations can be difficult to spot, and the result of such a conversion is almost always incorrect. This option is not enabled by `-Wall`.

## The `-fverbose-asm` option

Often used in conjunction with the `-S` option, and occasionally in place of the assembler's `-alh` option, `-fverbose-asm` tells gcc to put extra comments into generated assembly files. The extra information includes the memory or register locations of local variables, stack sizes, and argument passing conventions. This information is extremely useful for troubleshooting binary interface or code generation issues.

### The `-ffixed-<reg>` option

This option tells gcc to leave register `reg` alone. This is useful when you want to reserve a register for exclusive RTOS use, for example. In most cases, use of this option will require recompilation of runtime libraries and other code that depends on a binary interface.

### The `-fpack-enum` and `-fpack-struct` options

These options tell gcc to use the smallest representations possible for enumerations and structures, rather than using their default sizes. The resulting objects may be expressed in less space, but the assembly code required to access them will usually be slower and more complicated.

### `-O0`, `-O`, `-O1`, `-O2`, and `-O3`

These options tell gcc to perform varying levels of optimization on output files. The `-O` option produces the least optimization, while `-O3` produces aggressively optimized code. The `-O0` option tells gcc to not perform any optimizations at all (the default if no optimization level is specified). The `-Os` option tells gcc to only perform optimizations that don't negatively affect program size.

The `-O[x]` options affect the inclusion or exclusion of *sets* of optimization strategies. Gcc's online documentation lists the flags to control the use of specific optimization algorithms.

### The `-fomit-frame-pointer` option

This option tells gcc to omit frame pointer creation in functions that don't need it--- functions that have no local variables, for example.

### The `-Wa,<command>` and `-Wl,<command>` options

These options pass command line switches to the GNU assembler and linker, respectively.

## Syntax extensions

Gcc provides several language syntax extensions, including *inline assembly language*, assembly language generation controls, object *attributes*, and a long long object type.

## Inline assembly language

The most basic inline assembly statement supported by gcc is shown in Figure 1. This statement simply jams an assembly language instruction into the compiler's output stream when it is encountered.

**Figure 1. A basic inline assembly language example.**

```
printf ("Hello, world!\n");
__asm__( "mov r1, r2" );
printf ("R1 moved to R2.\n");
```

Obviously, this instruction will have disastrous results when the values of `r1` and `r2`--- perhaps the values of two local variables--- aren't what the programmer or compiler was expecting. This would be the likely case if the compiler's optimization levels changed after the code was written, or if the code was ported to a variant of the same processor family.

In these situations, gcc's *operand constraints* syntax comes to the rescue. Figure 2 is an example that uses the m68k's *fsinx* instruction to compute the value of `result` from `angle`. The "=f" and "f" tell gcc that it must use floating-point registers for the operands, and that the *fsinx* instruction modifies the register assigned to `result`.

**Figure 2. An example using operand constraints.**

```
float fsin (float angle)
{
  float result;
  __asm__( "fsinx %1, %0" : "=f" (result) : "f" (angle));
  return result;
}
```

Because operand constraints express concisely how gcc must construct the arguments to the opcode, gcc can properly structure any prologue or epilogue to the instruction needed to move its arguments into position. If gcc allocates the storage for `result` on the stack, for example, then the constraints tell gcc that it must move the contents of the register chosen for the %0 argument back to the stack after the opcode runs.

Gcc's operand constraint syntax prevents inline assembly language from disrupting the compiler's normal optimization processes. Consider the code in Figure 3. The assembly language is copying `b` to `a`, and so the return value is always `10`. With optimizations turned off, **gcc** emits code that does that explicitly, as shown in Figure 4. With optimizations turned on, however, (**-O3 -fomit-frame-pointer**), the code looks very different, as you can see in Figure 5.

During optimization, gcc apparently detects the constant return value created by the inline assembly language, and emits code to capitalize on that. Gcc doesn't move `#10` into `r0` directly, however, because gcc *never* eliminates code emitted via inline assembly language statements.

**Figure 3. A simple, optimizable inline assembly language example.**

```
int foo (int a)
{
    int b = 10;

    a = 20;
    __asm__("mov %1, %0" : "=r" (a) : "r" (b) ); /* sets a = b */
    return a;
}
```

**Figure 4. The output from Figure 3, compiled without optimizations. The code emits the requested opcode without modification. (Comments added by author.)**

```
_foo:

    /* frame pointer setup */
    mov.l r14,@-r15
    add #-8,r15
    mov r15,r14
    mov.l r4,@r14

    /* set b to 10 */
    mov #10,r1
    mov.l r1,@(4,r14)

    /* set a to 20 */
    mov #20,r1
    mov.l r1,@r14

    /* get b into a register
       per the operand constraints */
    mov.l @(4,r14),r2

    /* copy the b into a
       (redundant, but the inline asm
       we supplied requires this) */
    mov r2, r2

    /* write the result back to a */
    mov.l r2,@r14

    /* return a */
    mov.l @r14,r1
    mov r1,r0
    bra L1
    nop

    /* clean up the stack, and return */
    .align 2
L1: add #8,r14
    mov r14,r15
    mov.l @r15+,r14
    rts
```

**Figure 5. Output for Figure 3, compiled with optimizations. The optimizer exploits the constant return value created by the inline assembly language.**

```
_foo:
    mov #10,r1
    mov r1, r0
    rts
```

Operand constraints are documented in the *Extended Asm* section of the gcc user's manual.

## Controlling names used in assembler code

Occasionally the need arises to have C access to an assembly language object, but the object of interest isn't named in a C-friendly fashion. The code in Figure 6 creates the C object `foo_in_C` as an alias for the pathogenically-named assembly language symbol `foo_data`, which lacks a leading underscore and therefore can't normally be accessed in C.

**Figure 6. Declaring a C-friendly alias for an assembly language object.**

```
extern int foo_in_C __asm__("foo_data");
```

The same syntax is used to declare objects. In Figure 7, a C symbol called `bar` is assigned the name `bar_none` in the emitted assembly language. In Figure 8, whenever a function called `foo()` is referenced or declared, it gets the name `assembly_foo` in the emitted assembly code.

**Figure 7. Declaring a C object with a different name in assembly language.**

```
int bar __asm__("bar_none");
```

**Figure 8. Declaring a C function with a different name in assembly language.**

```
extern int foo (void) __asm__("assembly_foo");
```

## Object attributes

Gcc provides the `__attribute__` keyword to control object-specific settings. Some attributes that can be specified are alignment, packing, and section names. The code in Figure 9 emits the object `x` on a sixteen-byte boundary. The declaration in Figure 10 tells gcc to pack structure y as tightly as possible.

**Figure 9. Allocating a variable to an aligned address.**

```
int x __attribute__((aligned(16))) = 0;
```

**Figure 10. Packing a structure.**

```
struct { char a; int b; } my_struct __attribute__((packed));
```

The statement in Figure 11 allocates `z` to the `section_name` section. With the corresponding statements in a linker command file, `z` can be assigned to a specific memory address.

**Figure 11. Assigning an object to a named memory section.**

```
int x __attribute__((section("section_name")));
```

## long long object

Gcc provides a 64-bit object type for most targets. This type can be used just about anywhere any other integer type is valid. For targets where this type is not natively supported, gcc's runtime library provides an emulation.

```
long long a_big_value = 10;
```

# The GNU linker, ld

The GNU linker is a powerful application, but in many cases there is no need to invoke ld directly-- gcc invokes it automatically unless you use the `-c` (compile only) option.

Like many commercial linkers, ld's functionality is controlled using a combination of command line options and linker command files.

## Command line options

### --oformat=<format>

The GNU linker supports several output formats. Some choices are: srec (Motorola S records), coff-sh, and coff-m68k (COFF formats for SH and m68k targets, respectively). Once you've installed gcc, you can find out what formats it supports for your target using the following command:

```
$ <targetname>-objdump -i
```

### --output=<filename>

This option tells the linker what name to use for the output file.

### --print-map

This command tells the linker to create a map file.

### --cref

This tells the linker to add cross-reference information to the map file. This information is useful for determining why a particular module or object was linked into the executable.

### -T<filename>

This command tells ld to use the command file `<filename>`.

## Linker command scripts

Linker command scripts are by far the most effective way to control ld’s behavior. Figure 12 is an example linker command script that I will discuss in detail over the next several paragraphs. In summary, this script defines four memory regions called `vect`, `rom`, `ram` and `cache`, and five output sections called `vect`, `text`, `bss`, `init`, and `stack`.

**Figure 12. An example linker command script.**

```
/* a list of files to link
   (others may be supplied on the command line) */
INPUT(libc.a libg.a libgcc.a libc.a libgcc.a)

/* output format
   (can be overridden on command line) */
OUTPUT_FORMAT("coff-sh")

/* output filename
   (can be overridden on command line) */
OUTPUT_FILENAME("main.out")

/* our program’s entry point; not useful
   for much except to make sure the S7 record
   is proper, because the reset vector actually
   defines the "entrypoint" in most embedded systems */
ENTRY(_start)

/* list of our memory sections */
MEMORY
{
  vect  : o = 0, l = 1k
  rom   : o = 0x400, l = 127k
  ram   : o = 0x400000, l = 128k
  cache : o = 0xfffff000, l = 4k
}

/* how we’re organizing memory sections
   defined in each module */
SECTIONS
{
  /* the interrupt vector table */
  .vect :
  {
    __vect_start = .;
    *(.vect);
    __vect_end = .;
  } > vect

  /* code and constants */
  .text :
  {
    __text_start = .;
    *(.text)
    *(.strings)
     __text_end = .;
  }  > rom

  /* uninitialized data */
  .bss :
  {
     __bss_start = . ;
    *(.bss)
    *(COMMON)
     __bss_end = . ;
  }  > ram

  /* initialized data */
  .init : AT (__text_end)
  {
    __data_start = .;
    *(.data)
    __data_end = .;
  } > ram

  /* application stack */
  .stack :
  {
     __stack_start = .;
    *(.stack)
     __stack_end = .;
  }  > ram
}
```

Note that ld will use a default command file unless you tell it to do otherwise. To instruct ld to use your command file instead of its own, give gcc a `-Wl,T<filename>` command during compilation.

## The `OUTPUT_FORMAT` command

This command controls the format of the output file. A variety of formats are supported, including S-records (srec), binary (binary), Intel Hex (ihex), and several debug-aware formats, like COFF (coff-sh for SH-2 targets, coff-m68k for CPU32, etc.). Use the **objdump** utility to find out which formats are supported by your target's version of the linker. For more on **objdump**, see the section on building the tools from source code.

## The `MEMORY` command

The MEMORY command describes the target system's memory map. These memory spaces are then used as targets for statements in the SECTIONS comand.

The typical syntax is simple:

```
MEMORY {
    name : o = origin, l = length
    name : o = origin, l = length
    ...
}
```

Usually, there is a one-to-one relationship between statements in the MEMORY command and the number of uniform, contiguous memory regions supported by the target hardware. A frequent exception, however, is the processor's reset vector, along with the entire interrupt vector table in some cases. The reset vector is usually declared as an independent section so that its location in the output image can be strictly controlled.

## The `SECTIONS` command

Statements in a SECTIONS command describe the placement of each named output section, and specify which input sections go into them. You are only allowed one SECTIONS statement per command file, but it can have as many statements in it as necessary.

In the example, the statement:

```
/* code and constants */
.text :
```

starts the definition for a section called `.text`. The statements inside the subsequent curly braces instruct the linker to:

- create a symbol called `__text_start`, and place it at the beginning of the section,
- merge all `.text` and `.strings` sections from the input files into this section, and,
- create a symbol called `__text_end`, and place it at the end of the section.

Finally, the statement:

```
} > rom
```

tells the linker to locate the section in the memory space called `rom` which, according to the MEMORY command, begins at address `0x400`.

The list of input sections can also be file-specific. For example, if you added a line like:

```
foo.o (.specialsection)
```

to the `.text` section definition, then the linker would also merge into `.text` the section named `.specialsection` from the file `foo.o`.

## The `AT` directive

The AT directive tells the linker to load a section's data to somewhere other than the address it is located at. This feature is designed specifically for generating ROM images, something that's obviously important for embedded systems.

The best way to understand the AT directive is by example. So, consider an application that has only one initialized global variable:

```
int a_global = 102;
```

During compilation, gcc will declare an integer object `a_global` with the value `102` in the module's `.data` section. By supplying an AT directive for `.data` sections during linking, we tell the linker to assign `a_global` an address in one location (typically RAM), but place its initial value somewhere else (i.e. __text_end, typically in ROM).

With the initial value successfully stored in ROM, the question arises as to how to get it into the proper place in RAM. The code in Figure 13 initializes `a_global`, along with any other initialized global data in an application. This code uses the symbols `_text_end`, `_data_start` and `_data_end` to find the initial value, determine its size, and place it at its proper place in RAM.

**Figure 13. Code for initializing global data.**

```
extern const char _text_start, _text_end;
extern char _data_start, _data_end;
void __main (void)
{
  memcpy(&_data_start, &_text_end, &_data_end - &_data_start);
  return;
}
```

For most targets, the best place to put the code in Figure 13 is in a function called `__main()`. Gcc usually emits a silent call to `__main()` in the prologue for an application's `main()` function. (Use gcc's `-S` to see if this is the case for your target.)

## Putting It All Together

The following command line tells the ARM version of gcc to compile a file `main.c`, and then link it using the command file `main.cmd`. The output file will use the ELF debugging format.

```
$ arm-elf-gcc -g -Wl,-Tmain.cmd main.c
```

## Getting GNU

In contrast to other vendors, GNU development tools are all shipped as source code. To use the tools, you must first *build* them for the intended host and target.

The build process for GNU tools presumes the existence of a native C compiler and linker, so the best way to get started with GNU tools is to buy and install a GNU/Linux distribution CD. With this approach you get a working Linux environment with a preinstalled native GNU compiler and linker, plus a debugger and other tools. The Cygwin environment is also an option, if you intend to use a Win32 development host.

The next section, *The build script*, contains the basic commands for building a GNU crosscompiler and linker. The most important part of the process is deciding what target to build the tools for. The example uses the `arm-elf` target, which supports various flavors of the ARM microprocessor family and outputs images using the ELF file format by default. Some other target options are:

- m68k-elf (68k and CPU32 family)
- powerpc-elf (PowerPC family)
- sh-elf (Hitachi SH family)

Note that this is not a comprehensive list, by far. See the mailing list archives for the latest information on targets supported by the GNU tools.

GNU tools are distributed from the GNU website, at http://www.gnu.org (http://www.gnu.org/). A popular C runtime library used in the example script is newlib, which is available from http://sources.redhat.com/newlib. The build script has been tested with the following files:

- binutils-2.11.2.tar.gz
- gcc-2.95.3.tar.gz
- newlib-1.9.0.tar.gz

## The build script

First, log in as the root user (if necessary) and set up a directory structure in which to build the tools. Copy the `tar.gz` files into it, then set up some environment variables that will save typing later and make sure that the arguments used during the rest of the process are consistent--- a source of common errors. Figure 14 describes the commands, just type them in as they appear there (omit the leading ’$’, which represents the Cygwin or unix command prompt).

**Figure 14. Setting up the build environment.**

```
$ cd
$ mkdir build-crossgcc && cd crossgcc
$ cp ~/*.tar.gz .
$ mkdir build-binutils
$ mkdir build-gcc
$ mkdir build-newlib
$ export TARGET=arm-elf
$ export PREFIX=/usr/local
$ export PATH=${PREFIX}/bin:${PATH}
```

The value of `TARGET` is an argument that specifies the ARM microprocessor version of the tools, using the ELF debugging format. `PREFIX` specifies the topmost directory under which the GNU tools will be installed. The last command adds the GNU installation directory to the search path, so that we can run the tools after they are built and installed.

Decompress the source code for the tools themselves.

```
$ tar xzvf binutils-2.11.2.tar.gz
$ tar xzvf gcc-2.95.3.tar.gz
$ tar xzvf newlib-1.9.0.tar.gz
```

## Building the binutils package

The commands to configure and install the assembler and linker are shown Figure 15. These commands do the following:

- Run the **configure** script included with the binutils package, to set up the source code to build for the ARM target.
- Invoke the **make** program to actually compile and install binutils. The output from **make** is captured and stored in the file `make.log`.

When this process finishes, you will see a number of programs in `${PREFIX}/bin`. The assembler is **arm-elf-as**, and the linker is **arm-elf-ld**. **arm-elf-objcopy** is a utility that translates files to different formats (from ELF to S Record, for example), and **arm-elf-objdump** is a utility that can dissect the components of a file, to show you things like symbol addresses and a disassembly of the file's object code.

**Figure 15. Instructions to build binutils**

```
$ cd build-binutils
$ ../binutils-2.11.2/configure --target=$TARGET --prefix=$PREFIX
$ make all install 2>&1 | tee make.log
$ cd ..
```

## Building a bootstrap cross compiler

Now that we have an assembler and linker, we can build and install the GNU C/C++ compiler. The initial step in the procedure yields a *bootstrap* compiler that can only be used to build runtime libraries and header files. We will use this compiler to build the arm-elf version of the newlib C runtime library. Once that's done, we will rebuild the compiler completely, including internal libraries that need target-specific header files from newlib in order to be compiled properly.

The commands to make the boostrap compiler are shown in Figure 16.

**Figure 16. Instructions to build a bootstrap gcc.**

```
$ cd build-gcc
$ ../gcc-2.95.3/configure --target=$TARGET --prefix=$PREFIX \
   --with-newlib --without-headers --with-gnu-as \
   --with-gnu-ld --disable-shared --enable-languages=c
$ make all-gcc install-gcc 2>&1 | tee make.log
$ cd ..
```

At this point we have a bootstrap compiler called **arm-elf-gcc**, located in `${PREFIX}/bin`.

## Building the newlib C runtime library

The procedure, shown in Figure 17, should seem pretty familiar by now.

**Figure 17. Instructions to build newlib.**

```
$ cd build-newlib
$ ../newlib-1.9.0/configure --target=$TARGET --prefix=$PREFIX
$ make all install 2>&1 | tee make.log
$ cd ..
```

## Building a complete cross compiler

Now that we have ARM header files and libraries from newlib, we can build a complete cross compiler setup for C/C++ development. The steps are shown in Figure 18.

**Figure 18. Instructions to build gcc.**

```
$ cd build-gcc && rm -rf *
$ ../gcc-2.95.3/configure --target=$TARGET --prefix=$PREFIX \
   --with-gnu-as --with-gnu-ld --enable-languages=c,c++
$ make all install 2>&1 | tee make.log
$ cd ..
```

## Wrapup

As you can see, the GNU tools have a lot to offer for embedded development. If you give them a try, you are likely to find that their power and flexibility makes them the perfect choice for your next embedded product. I did!

## Resources

- http://www.billgatliff.com

  Additional information on GNU programming for embedded systems.

- http://sources.redhat.com/ml/crossgcc

  The CrossGCC mailing list archives.

- http://sources.redhat.com/cygwin

  The Cygwin unix emulation package.

- http://sources.redhat.com/binutils

  The Binutils homepage.

- http://gcc.gnu.org

  The GNU Compiler Collection homepage.

- http://sources.redhat.com/gdb

  The GNU debugger homepage.

- http://sources.redhat.com/newlib

  The Newlib homepage.

- http://www.gnu.org

The Free Software Foundation's GNU website.

- http://sourceforge.net/projects/gdbstubs

  Homepage for the gdbstubs project.

## Copyright

## About the Author

Bill Gatliff is an independent consultant with almost ten years of embedded development and training experience. He specializes GNU-based embedded development, and in using and adapting GNU tools to meet the needs of difficult development problems. He welcomes the opportunity to participate in projects of all types.

Bill is a Contributing Editor for Embedded Systems Programming Magazine (http://www.embedded.com/), a member of the Advisory Panel for the Embedded Systems Conference (http://www.esconline.com/), maintainer of the Crossgcc FAQ, creator of the gdbstubs (http://sourceforge.net/projects/gdbstubs) project, and a noted author and speaker.

Bill welcomes feedback and suggestions. Contact information is on his website, at http://www.billgatliff.com.