An Introduction to the GNU Compiler

The GNU Compiler, part of the GNU Tools software suite, is the C compiler used in the Digital Systems Laboratory to compile C source code[1] into binary object files and executables. This compiler is extensively documented in *Using the GNU Compiler Collection;* you can find this document on your CD-ROM in the *gnutools/doc* directory. This Introduction restricts itself to describing how to use the GNU Compiler in the Laboratory: it does *not* attempt to teach you how to program in C.

Invoking the Compiler

You can use the GNU Compiler for the ARM by executing the **`arm-elf-gcc`** program. This can be invoked by a command line similar to the following:

```
arm-elf-gcc [stage-opt] [other-opts] -mcpu=arm7tdmi in-file -o out-file
```

You will need to replace *`stage-opt`* with one of the stage options listed later, *`other-opts`* with any other options that you need, *`in-file`* with the name of your input file, and *`out-file`* with the name of the output file. Both the input file and output filenames must have the appropriate extensions. Please note that you do *not* include the square brackets "[" and "]": these simply indicate that *`stage-opts`* and *`other-opts`* are optional and may be omitted.

Although all of this may sound rather complicated, in practice you would simply use "standard invocations" without modification. These are listed in the following summary with the briefest of explanations; the rest of this document deals with the various options and their usage.

To convert C source code to a binary object file:

```
arm-elf-gcc -c -O2 -g -mcpu=arm7tdmi filename.c -o filename.o
```

To convert multiple binary object files into an executable file (the general case):

```
arm-elf-ld filename1.o filename2.o … -o filename.elf
```

To convert C source code to an executable file (for use with Insight only, not Komodo):

```
arm-elf-gcc -O2 -g -mcpu=arm7tdmi filename.c -o filename.elf
```

To convert C source code to assembly language source code:

```
arm-elf-gcc -S -fverbose-asm -mcpu=arm7tdmi filename.c -o filename.s
```

Stages of Compilation

The GNU Compiler actually passes through four distinct stages to convert your C program into an executable file: it *preprocesses* your source code file, then *compiles* it, *assembles* it and finally *links* it. These stages are shown in the following diagram:

$$\begin{tikzcd}\boxed{\textbf{Preprocessing}} \arrow[r] & \boxed{\textbf{Compiling}} \arrow[r] & \boxed{\textbf{Assembling}} \arrow[r] & \boxed{\textbf{Linking}}\end{tikzcd}$$

Input:	C code (*.c*)	Preprocessed (*.i*)	Assembler (*.s*)	Object (*.o*)
Output:	Preprocessed (*.i*)	Assembler (*.s*)	Object (*.o*)	Executable (*.elf*)
***stage-opt*:**	`-E`	`-S`	`-c`	(none)

[1] The GNU C Compiler is technically and properly called the GNU *Compiler Collection:* it can compile not only C programs, but many other languages as well. The version in the Digital Systems Laboratory can compile C and C++ programs, although it is mainly used only as a C compiler.

You can *start* the GNU Compiler at any stage by supplying an input file with the required extension, as shown in the diagram. For example, specifying an input file ending in *.c* will make the compiler start in the Preprocessing stage, while specifying an input file with the *.s* extension will make the compiler start in the Assembling stage.

You can also *stop* the GNU Compiler at any stage by specifying the appropriate `stage-opt` stage option; you must also make sure that the output filename has the correct extension. For example, specifying **`-E`** will stop the compiler just after the Preprocessing stage, in which case the output filename must end with *.i*. As another example, specifying no option at all will make the compiler proceed right to the end of the Linking stage;[2] in this case, the output filename must have the extension *.elf*.

Preprocessing

The first stage of the GNU Compiler converts your input file into preprocessed output. Preprocessing a file converts all *preprocessing statements* (such as `#include`, `#define` and `#ifdef`) into true C code. The input filename must have the extension *.c* to signify C source code (in other words, the filename must end in *.c*, such as in *file.c*). You can stop the GNU Compiler at this stage by supplying **`-E`** as the stage option. If you specify **`-E`**, the output filename must have the extension *.i*.

In practice, you almost always want to start the GNU Compiler at the preprocessing stage, but almost never want to stop at that stage.

Compiling

The second stage of the compiler converts preprocessed input into assembly language output. This is where the bulk of the work in compiling a C program is done. You can start the compiler at this stage by specifying an input filename with the *.i* extension. You can also stop the compiler at this stage by specifying **`-S`**, in which case your output filename must end in *.s*.

In practice, you almost never want to start the GNU Compiler at this stage: you would normally start at the preprocessing stage by specifying a *.c* file as input. There are times, however, when you want to *stop* at this stage, usually to check the quality of the code produced by the compiler.

There is a plethora of options to control exactly how the compiler converts your C source code into assembly language (and, from there, into an object file or executable). Some of these are listed later in this document. The most useful are the debugging option **`-g`** and the optimisation options, such as **`-O2`**.

Assembling

The third stage of the compiler is to convert the assembly language code into relocatable binary object output. The GNU Compiler actually calls the GNU Assembler, **`arm-elf-as`**, to do the real work of assembly. For this reason, there is almost no reason to start the GNU Compiler from this stage (by specifying a file ending in *.s*).

On the other hand, the relocatable binary object that is generated by this stage *is* often desirable. For example, you can create a number of object files from C source code (by calling the compiler multiple times) which can then be linked together to form one executable. You can make the compiler stop at this stage by specifying **`-c`** as the stage option, in which case your output filename must have the extension *.o*.

[2] This assumes, of course, that there are no errors in your input files. If there are, the GNU Compiler will naturally stop much earlier!

Linking

The fourth and final stage of the compiler is to link the relocatable object file into an executable output file. The GNU Compiler uses the GNU Linker, **`arm-elf-ld`**, to do the real work. You can start the compiler at this stage by specifying an input filename with the extension *.o*. You can make the compiler stop at this stage (and not earlier) by *not* supplying a state option, in which case the output filename must end in *.elf*.

Important: Using the GNU Compiler to create your executable is *not* quite the same as using the GNU Linker, **`arm-elf-ld`**, yourself. The reason is that the GNU Compiler automatically links a number of *standard system libraries* into your executable. These libraries allow your program to interact with an operating system, to use the standard C library functions, to use certain language features and operations (such as division), and so on. If you wish to see exactly *which* libraries are being linked into the executable, you should pass the verbose flag **`-v`** to the compiler.

This has important implications for embedded systems! Such systems do not usually have an operating system. This means that linking in the system libraries is almost always meaningless: if there is no operating system, for example, then calling the standard `printf` function does not make much sense.

There are three possible solutions: either use the GNU Linker instead of the compiler to create the executable, or tailor the system libraries to the embedded system, or write an operating system that meets the requirements of the standard libraries. The third option is used by the GNU Debugger's simulator: it includes a "mini OS" that allows you to use `printf`, `scanf` and so on. The first option is by far the simplest, and is the one used in the Laboratory. In other words, when writing C programs for the DSLMU Microcontroller Board, you must *not* use the standard system libraries; instead, you must use **`arm-elf-ld`** to manually link relocatable object files that have been created specifically for the hardware environment.

Other Options

There are literally hundreds of options that you can pass to the GNU Compiler! Fortunately, you will only need to use a handful of these on a routine basis; these more useful options are listed in this document. If you want a brief look at the other available options, you are encouraged to look at the `arm-elf-gcc`(1) manual page. This rather cryptic notation simply means that you type "**`man 1 arm-elf-gcc`**" on the command line.[3]

Debugging Options

The debugging options instruct the GNU Compiler to produce information that will help you debug your programs. The most common debugging options are:

`-g`	Include debugging information in the relocatable binary object or executable output file. This information allows you to use the GNU Debugger to debug your program at the source code level. This option does *not* modify your program in any way, and thus should almost always be used.
`-Wall`	Warn about potential bugs and questionable constructs that may exist in your C code. Highly useful; use often!
`-fverbose-asm`	You should include this option if you are converting your C source code into assembly language (**`-S`** stage option). This causes the compiler to insert comments into the output to help you understand the resulting assembly language code.

[3] The notation may be rather cryptic, but it is hard to change well over thirty years of Unix history! In general, the notation "*`command`* (*`n`*)" simply means you type "**`man`** ***`n command`***" at the shell prompt.

-v	Be verbose: make the GNU Compiler display the actual command lines that it uses to perform the different stages of compilation, along with additional information. Useful if you want to see the exact command-line options and system libraries used to create the final executable.

Optimisation Options

The optimisation options directly affect the quality and size of the output code produced by the GNU Compiler. In particular, they direct the compiler to try harder at producing better-quality code. The most common optimisation options are:

-O0	Do not optimise the code at all. This option makes the compiler produce code that exactly matches, statement by statement, what was specified in the original C program. This makes assembly-language-level debugging easier, since each C statement is translated into independent blocks of code. Please note that the second character in all of these options is the upper-case letter O, *not* the number zero. The third character in this particular option is the number zero.
-O1	Try moderately hard to produce assembly language code that is faster and smaller. This makes the compiler take a little longer to run.
-O2	Try much harder to produce optimal assembly language code (which can then be assembled and linked, as usual). This can make the compiler take much longer to do its work.
-O3	Try hardest of all to produce fast assembly code. Note that the emphasis is on *fast:* the resulting code may take much more room in memory. This is because certain functions may be placed "in-line" and loops may be unrolled (eg, as if each iteration in a `for` loop was written out independently).
-Os	Try to produce code that is *small:* the emphasis is on size, not speed. This option uses many of the same optimisation algorithms as **-O2**, but with a different emphasis.

Notice that, given the same C program as input, output code that has been optimised for space often takes *longer* to run than output code that has been optimised for time and is consequently much larger. This is called the *space-time trade-off*, and has been one of the key problems facing the computer industry since its inception. These days, with vast amounts of memory often available, you would usually choose to optimise for speed (ie, to produce larger code that is faster). However, an embedded system is one place where you just might select **-Os** after all, especially if it means the difference between your code being able to fit into a Flash ROM or not!

The best way to see the effect of using these options on your C program is to stop the GNU Compiler at the assembly stage, then to examine the resulting assembly language code. The most useful of these options is **- O2**.

Machine-specific Options

The GNU Compiler in the Laboratory, **arm-elf-gcc**, produces code that is targeted at the ARM architecture. However, this architecture encompasses a great number of variants, both in hardware and in software. The machine-specific options direct the compiler to produce a particular style of code. The only option useful in the Laboratory is:

-mcpu=arm7tdmi	Produce code that is specific to the ARM7TDMI architecture, as used on the DSLMU Microcontroller Board. This architecture is also known as ARMv4T. You should always specify this option.

Disassembling Binary Files

The GNU Tools software suite includes a number of useful tools that can be used to investigate and manipulate the relocatable binary object files and executables. One of the more useful of these is the **`arm-elf-objdump`** utility.

The **`arm-elf-objdump`** utility displays information contained in a relocatable binary object file or executable. In particular, you can use this tool to disassemble object files, which allows you to determine the exact instructions being used by the compiler and assembler. You can also obtain information on the memory addresses that will be used when loading the executable file. The tool can be invoked as:

```
arm-elf-objdump option filename | more
```

Simply replace *`filename`* with the name of your object file or executable, such as *file.o* or *file.elf*, and *`option`* with one of the following:

`-d`	Disassemble any sections in the object file or executable that contain ARM machine code, along with the address and hexadecimal bit-pattern of each instruction. Please note that any address listed in relocatable binary object files is *not* the address in memory at which that instruction will appear. It is the linker's job to convert relocatable binary object files into an executable file: a file that *can* be loaded into memory.
`-x`	Display information about the relocatable binary object file or executable, such as the symbol table, any relocations that are needed and the location of each section of code. This option is most useful with executable files, as it allows you to see exactly where each program section will be loaded into memory. For example, most executables contain a `.text` section that loads at address 0x8000; using this option will confirm that the VMA (Virtual Memory Address) and LMA (Load Memory Address) is indeed 0x8000 (or close to it).

You can find more information about **`arm-elf-objdump`** in the `arm-elf-objdump`(1) manual page and in the *GNU Binary Utilities* reference manual; you can find this manual on your CD-ROM in the *gnutools/doc* directory.

More Information

You can find more information about the GNU Compiler in *Using the GNU Compiler Collection.* This 400-odd-page document is well-written and quite comprehensive. You can find it on your CD-ROM in the *gnutools/doc* directory.

The definitive reference is the actual source code to the GNU Compiler. You can find this on your CD-ROM in the *gnutools/src* directory. After unpacking the GNU Compiler archive files in that directory (and after applying the appropriate patch files), try browsing the source code files in the *gcc* directory.