Embedding with GNU: The GNU compiler and linker

By Bill Gatliff
Embedded Consultant and Senior Design Engineer
Komatsu Mining Systems

A quality compiler, linker, and debugger are critical for the success of any embedded project. In two articles last year, I discussed the benefits of the GNU debugger, gdb. [1,2] But unlike debuggers, I find it difficult to get excited about cross compilers and linkers. They're excruciatingly technical products. To make matters worse, I tend to stick to the ones that quietly and reliably churn out code, instead of the ones that don't always get it right but put on a big show in the process.

On the other hand, although the GNU compiler and linker are the strong, silent types, they also sport some nifty features that can really ease the burden of writing solid, portable code for embedded systems. If you stick with me here for a few minutes, I think you will agree.

gcc: the GNU C compiler
In addition to its popularity on the desktop, gcc is perfectly capable of producing high-quality code for embedded systems of all types. A complete introduction to all of its features isn't possible in the space I have available here, so I'll only mention the ones that are most useful for embedded systems. See the gcc on-line help files for the rest.

Syntax and potential run-time error checking
The GNU compiler provides several command-line options that control how much trust it will place in your code. The "-Wall" setting is my favourite because it causes gcc to warn you about nearly everything that looks suspicious.

Some other useful options include: "-Wformat" (checks arguments to printf() calls); "-Wcast-align" (warns if a cast can cause alignment issues); and "-Wconversion" (warns if a negative integer constant expression is implicitly converted to an unsigned type).

Inline assembly code
This compiler provides a powerful syntax for embedding assembly language statements in C/C++ source code. In the most basic case, you can insert assembly language instruction(s) into a block of C code as follows:

```
void set_imask_to_6( void )
{
printf("switching to interrupt
mask level 6.\n");
__asm__(" andi #0xf8, sr");
__asm__(" ori #6, sr");
printf("Interrupt mask level
is now 6.\n");
   }
```

But that's just the tip of this feature iceberg. The compiler also provides a way to safely refer to C objects in assembly statements, instead of requiring you to predict in advance which register will be used to store the value of interest.

For example, if you wanted to use the 68881's fsinx instruction in as robust and portable a way

Installing the GNU compiler and linker

The GNU compiler is the only application that can reliably build a cross-platform version of itself, so you'll need to do all of the following on a machine that already has a native gcc installed. Get Linux, or use Cygwin (*www.cygnus.com*) if you need to build and/or run under Win32.

To build and install the GNU compiler and linker, you first must get the source code. The files you need are:

- gcc: ftp://ftp.gnu.org/gnu/gcc/gcc-2.95.1.tar.gz
- linker, assembler, and other utilities: ftp://ftp.gnu.org/gnu/binutils/binutils-2.9.1.tar.gz
- run-time library: ftp://sourceware.cygnus.com/pub/newlib/newlib-1.8.1.tar.gz

Untar all the files, login as root, and abide by the following script. If you don't have root privileges on your machine, then add **-prefix=<dir>** to the configure statements, replacing **<dir>** with a pathname that you have permission to write to.

Build the assembler and linker first:

```
cd binutils-2.9.1
configure
-target=your_target_name
make all install
```

Example target names are sh-hitachi-hms for the SH, m68k-unknown-coff for CPU32, and so on. See configure.sub for information on supported target/host combinations, but beware: the list is long!

Now build the compiler and run-time libraries:

```
cd gcc-2.95.1
rm -rf libf2c
ln -s newlib ../newlib-1.8.1/newlib
configure —target=your_target_name —with-newlib
—with-headers=
/newlib/libc/
include5

make cross LANGUAGES="c c++" install
```

as possible, you could do it like this: 3

```
__asm__("fsinx %1,%0" : "=f"
(result) : "f" (angle));
```

The "=f" and "f" are called operand constraints, and they're used to tell gcc both how it must generate the %0 and %1 expressions to make the opcode function properly, and what side effects the operand produces. In this example, they tell gcc that it must use floating-point registers for the values of the variables angle and result, and that the fsinx instruction returns the answer in variable result.

Operand constraints allow mixed assembly and C/C++ statements to work properly even when changes in optimisation levels and other compiler settings might cause them to do otherwise. A variety of operands and constraints are available, and several examples are provided in the "Extended Asm" section of the gcc manual.

One other nice thing about gcc's inline assembly language feature is that it doesn't disrupt the compiler's normal optimisation processes nearly as much as it seems to with other compilers. To illustrate this, consider the following simple function:

```
int foo( int a )
{
int b = 10;
a = 20;
__asm__ ("mov %1, %0" : "=r"
(a) : "r" (b) ); /* a = b */
return a;
}
```

The assembly language is simply copying b to a, and so the return value is always 10. With optimisations turned off, gcc emits code that does that explicitly (Hitachi SH-2 code, in this case):

```
_foo:
mov.l r14,@-r15
add #-8,r15
mov r15,r14
mov.l r4,@r14
mov #10,r1
mov.l r1,@(4,r14)
mov #20,r1
mov.l r1,@r14
mov.l @(4,r14),r2
mov r2, r2
mov.l r2,@r14
mov.l @r14,r1
mov r1,r0
bra L1
nop
.align 2
L1: add #8,r14
mov r14,r15
mov.l @r15+,r14
rts
```

Crank up the optimisation level (-O3 -fomit-frame-pointer), however, and the code looks very different:

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

In the latter case, gcc's optimiser concluded that the only possible return value was 10, which means that it "understood" the assembly code we supplied and used it to its advantage. Pretty spiffy behaviour, and not something I've come to expect from the commercial compilers I've used.

Why didn't gcc just put the 10 into r0 in the first place? Because gcc will optimise around user-supplied assembly code, but it won't omit it. In other words, the **mov r1, r0** is there because of our inline assembly statement, not because gcc put it there.

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 following code allows C statements to use a symbol named **foo_in_C** to refer to the pathogenically named assembly language symbol **foo_data** (which lacks a leading underscore, and therefore can't normally be accessed in C):

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

A similar syntax is used for declarations as well:

```
int bar asm ("bar_none");
extern int foo( void ) asm
("assembly_foo");
```

The first statement causes gcc to create a C symbol called bar, but emit assembly code that calls it **'** instead of the usual **_bar** . The second statement causes gcc to use the name **assembly_foo** (instead of **_foo**) whenever the C function foo() is either called or defined.

Section specification

Many commercial cross compilers allow you to specify the target memory section for declarations by using a command-line option. For example, to place all of a module's constant global declarations into a section called **myconsts** , you often use a command similar to the following:
compiler -C"myconst" main.c

In my opinion, the problem with this approach is that it invisibly changes the effect of the const keyword, which leaves open the possibility that future **const** additions to the module will accidentally end up in the wrong memory space. Furthermore, it forces you to split **myconst** and generic constant declarations into separate modules, which creates a real maintenance headache as a project matures.

In contrast, gcc's approach is to specify allocation sections on a per-declaration basis, using its section attribute language extension:

```
const int put_this_in_rom
__attribute__((section("myconst"
)));
const int put_this_in_flash
__attribute__((section("myflash"
)));
```

In contrast to the command-line approach, this technique allows you to put all of the declarations related to a piece of functionality into the same source module, regardless of their destinations in the target's memory map.

Section attributes can also be used to construct data tables. For example, eCos (the open-source Embedded Cygnus Operating System), uses section attributes to place all device information structures (one of which is allocated in each device driver's source module) into a section called **devtab** , which the operating system then **analys**es at run time. Because of this approach, adding or removing a driver is as simple as adding or removing a module from the application—there is no "master device driver list" to modify.

Interrupt service routines

An interrupt_handler attribute isn't provided by gcc for most target processors. This means that you can't write an entire ISR (interrupt service routine) in C, because gcc won't return from the function using a return-from-exception opcode.

This limitation is not as significant as it sounds. A common workaround is to write a small snippet of assembly language code that saves registers, calls the C code, and then exits with an **RTE** , like this:

```
void isr_C( void )
{
/* whatever we do in C */
...
}
__asm__("
.global _isr
_isr:
/* push scratch registers—C
* preserves the rest
*/
push r0
push r1
...
/* call the workhorse */
jsr _isr_C
/* clean up, return */
...
pop r1
pop r0
rte
");
```

Why doesn't gcc make things easier for ISR writers, you ask? The reason appears to be that gcc's primary mission (although this is changing quickly) is to produce code for desktop workstations, which have no need for interrupt service routines. Further-more, modifying gcc's stack frame and register management implementation is nontrivial, and so most serious GNU users seem to prefer to write a few lines of assembly language here and there, rather than risk additional bugs.

Reserving registers

Sometimes it's nice if you can tell a compiler never to use a particular register, perhaps because you have assembly language functions that require a register value be preserved across C function calls.

As you might have guessed, gcc can do this. Simply put a -ffixed-REG on the command line, that is:

```
gcc -ffixed-a7 myprogram.c
```

This capability is also useful if your RTOS reserves a register or two for its own use, or your application has assembly code that needs a high-speed global variable.

Function names as strings

The standard C preprocessor macros provide minimal information useful for display output at run time. For example, you can describe the **__DATE__** and **__TIME__** of a **__FILE__** , but not much else.

Not only does gcc support these terms, it also adds two of its own: **__FUNCTION__** and **__PRETTY_FUNCTION__** . The two produce the same output in C, but the latter provides more information when it appears in a C++ module. In either case, the information they provide can be useful, especially for diagnostic outputs caused by failed assertions.

For example, this statement:

```
printf ("The function %s in file
%s, was compiled on: %s.\n",
__PRETTY_FUNCTION, __FILE__,
__DATE__ );
```

yields either:

```
The function foo, in file foo.c,
was compiled on: Feb 10
1999.
```

or:

```
The function int c::foo, in file
foo.cpp, was compiled on:
Feb 10
1999.
```

Debugging information

You always have to tell gcc to include debugging information in output files, using the **-g** flag:

```
gcc -g main.c
```

Without the **-g** , the application will run but you won't be able to debug it.

ld: the GNU linker

The GNU linker is a powerful application as well, 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, most of ld's functionality is controlled using linker command files, which are text files that describe things like the final output file's memory organisation. **Listing 1** contains 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 the following output sections: **vect, text, bss, init, and stack** .

As with gcc, I don't have the space to discuss every ld feature or supported command, but they're all described in ld's online documentation. Just type **info ld** after installation.

The linker 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.

Listing 1 An example linker command script

```
/* a list of files to link
(others are 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 organising
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
}
```

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, and so on).

The GNU linker derives its output file formatting capabilities from a library known as . Use **objdump -i** to find out which formats are supported by your target's version of the linker.

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
  ...
}
```

A one-to-one relationship usually exists between statements in the **MEMORY** command and the number of uniform, contiguous memory regions supported by the target hardware. A typical exception, however, is the processor's reset vector—and in some cases, the entire interrupt vector table—which is usually declared as an independent section so that its final location can be strictly controlled.

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 named .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
- 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 entire section in the memory space called rom which, according to the MEMORY command, begins at address 0x400. 4

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, the linker would also merge into .text the section named **.specialsection** from the file **foo.o** .

AT directive

The **AT** directive tells the linker to load a section's data somewhere other than the address at which it's actually located. 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 initialised 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. But 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 (**__text_end** , usually **ROM**).

The following code initialises **a_global** (and any other initialised 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** :

```
extern const char _text_start,
 _text_end;
extern char _data_start,
 _data_end;
memcpy(        &_data_start,
&_text_end,
 &_data_end - &_data_start );
```

Now to put it all together. The following command line tells gcc to compile a file, main.c, and then link it using the linker command file **main.cmd** :

```
gcc -g -Wl,-Tmain.cmd main.c
```

Issues specific to systems built using GNU tools

Although GNU and other open source tools are available at no cost, they aren't necessarily free for unrestricted use. For example, if you link your application with a library released under the terms of the GPL (GNU Public License), then, according to the terms of the licence, you must make your application available in source form as well. Unfortunately, this is the case for the C run-time library (the code for **printf(), malloc()** , and so forth), most often used with gcc: glibc.

A suitable C run-time alternative that isn't restricted in this fashion is a library called newlib, available from the Cygnus Solutions archives. According to the terms of this library's licence, you may build proprietary applications without disclosing your source code, so long as you acknowledge in a manner appropriate to your application that you're using Cygnus' newlib.

Carefully read and understand the licences for any open source software (or any other software, for that matter) that you use to create or include in your embedded application. If you can't abide by the terms, you can't use the code.

What have you got to lose?

Don't take my word for it—the beauty of GNU is that you can download and try out the tools yourself at no charge. I encourage you to do so and to think seriously about using GNU tools for your next embedded project.

References

1. Gatliff, Bill, "Embedding with GNU: The GNU Debugger," *Embedded Systems Programming* , September 1999, p. 80.
2. Gatliff, Bill, "Embedding with GNU: The gdb Remote Serial Protocol," *Embedded Systems Programming* , November 1999, p. 108.
3. This example, along with several others, is included in the gcc documentation. 4. The name **.text** is traditional. With **gcc** , the text section actually includes the application's instruction code, constant declarations, and text strings, subject to section attributes supplied in individual modules. 5. Replace **<ABSOLUTE_PATH_TO_NEWLIB>** with where you untarred the newlib sources. for example: **/home/me/crossgcc-test/newlib-1.8.1]**)

Email   Send inquiry