XGCC

The Gnu C/C++ Language System for Embedded Development

Revision: Beta 1, 1/23/2000

Limits of Liability and Disclaimer of Warranty

Embedded Support Tools Corporation have used their best efforts in preparing the book and the programs incorporated in this product. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

Embedded Support Tools Corporation makes no warranty of any kind, expressed or implied, with regard to these programs, or the documentation contained in this book. It is entirely your responsibility to determine the suitability of these programs for your particular needs. Neither Embedded Support Tools Corporation nor its employees, officers, directors, or distributors shall be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of this book or these programs, even if they have been advised of the possibility of such damages.

Trademarks

XGCC and VisionClick are trademarks of Embedded Support Tools Corporation. PowerPC is a trademark of IBM Corporation, used under license therefrom. ColdFire and BDM are trademarks of Motorola Inc. Microsoft and Microsoft Windows are trademarks of Microsoft Corporation. All other trademarks are acknowledged to be the property of their respective owners.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 XGCC: Gnu CC from Embedded Support Tools Corporation

This manual documents XGCC, EST's release of Gnu CC which runs on the Microsoft Windows family of operating systems and generates code for a variety of embedded processor architectures. Much more than just a compiler, XGCC is a complete C/C++ language system that complements EST's premium C/C++ source-level debugger, VisionClick, providing a high-quality end-to-end solution for embedded development.

This manual is designed to be used together with the Gnu documentation that is installed on your computer with the compiler tools. In some cases it will fill in some of the gaps in the Gnu manuals, particularly on topics of interest to embedded developers; in other cases, it pulls together and summarizes information that may be spread out over several different manuals. Finally, it documents some of the enhancements and additions made to the Gnu tools by EST.

1.2 The Gnu project

An organization called the Free Software Foundation was created in 1984 to sponsor the development of (surprise…) free software. Since then the FSF have released dozens of programs that have received high praise for their quality and reliability. All of these programs

were released in source code form, freely accessible by anyone who wanted to download them.

The FSF define ‘free’ not in terms of *cost*, but in terms of *access*: the source code is always available, and if you add or change something and give the resulting program to somebody, you must also offer to give them the source code to your changed program, in order not to deny them any rights of access that were given to you by the program’s original author. The Gnu Public License (GPL) is the document that defines the legal license for the FSF programs, and it has since been adopted by many other individuals and organizations in releasing their own free software to the public.

The FSF’s Gnu project is an attempt to create a complete Unix work-alike operating system that is entirely made up of free software. Although original plans called for this system to be based upon the FSF’s own kernel (called the Hurd), this goal has now been largely attained through the Linux project, which is entirely based upon free (GPL’d) software and is now becoming a major force in the operating systems world.

This approach is radically different from the traditional approach of commercial program development. The FSF survives through corporate and private donations of time, money, computers, people, and office space. By releasing the code in source form with universal access, many thousands of motivated programmers end up making contributions to the programs, which ultimately results in very high-quality, feature-rich software. A different paradigm to be sure, but one that has proven to be successful in attaining its goals of high-quality, freely-available software.

1.3 Technical Support Information

EST Corporation provides free technical support for XGCC for a period of 90 days from date of purchase. After the initial 90 days, an Extended Support Agreement entitles you to additional free technical support. EST may be reached as follows:

Mailing address and telephone number

EST Corporation Headquarters
120 Royall St.
Canton, MA 02021
(781) 828-5588

EST Europe
12 Avenue De Pres
78180 Montigny Le Brettoneax
France
+33 (0) 1 3057 3200

For a complete listing of EST's worldwide sales offices, please consult the EST web site at http://www.estc.com/.

EST Technical Support Department Hours

Monday-Friday
8:30 A.M. – 6:00 P.M.
Eastern Standard Time

Internet (e-mail)
estsupp@estc.com

URL
http://www.estc.com

FTP server
ftp://estftp.estc.com

1.4 CD-ROM Contents

The XGCC CD-ROM distributed by EST contains everything you will need to get started quickly on your next embedded project. The following components are included:

1.4.1 Gnu C/C++ compiler for M68K and PowerPC embedded systems

EST have ported the Gnu CC compiler to run on the Win32 operating systems, cross-compiling to embedded systems. Currently, the Motorola M68K family and IBM/Motorola PowerPC families are supported. Over time, we will add support for other microprocessor families in future releases of the CD-ROM. The CD-ROM will be updated to track the new releases of the compiler.

The compiler comes with the latest version of the Silicon Graphics Inc. Standard Template Library (STL) implementation.

1.4.2 Gnu assembler (`as`), linker (`ld`) and binary utilities

Included with the compiler are the Gnu assembler and linker, again running on Win32 and cross-compiling to M68K and Power PC. Also included are the so-called binary utilities, which are a set of utility programs to manipulate object files in various formats. The most commonly-used binary utility programs are listed below:

- `objcopy`, a utility to copy object files between various different object and hex/ASCII formats
- `objdump`, a utility to examine the contents of object files
- `ar`, the Gnu object library (archive) manager
- `nm`, a utility to list symbols defined in object files
- `ranlib`, a utility to index object libraries for faster access
- `size`, which lists the individual and total sizes of the sections contained in a list of object files
- `strings`, which lists printable strings contained in an object file
- `strip`, a utility to remove debug information from object files

1.4.3 newlib C runtime library

newlib is a complete implementation of the standard C runtime library suitable for embedded applications. It is a collection of free software that was assembled by Cygnus Solutions to address two common issues in embedded applications:

- Most standard C library implementations are not appropriate for small- or medium-scale embedded systems, because of the amount of memory they require; and
- Some libraries have licensing restrictions that make it difficult to embed the software in a ROM-based product without also supplying source code to the end customer.

newlib is easy to adapt to embedded systems, and requires relatively small amounts of RAM and processor bandwidth. In addition, it is licensed under a BSD-style license, which means that there is no restriction against using the library in a commercial product.

1.4.4 Gnu iostream C++ library

This library implements iostreams on top of the standard C I/O library routines.

1.4.5 EST's librom.a I/O driver library

The newlib standard C library requires several supporting routines from the underlying operating system to link and run successfully. For embedded targets which do not use an operating system, we have provided the `librom` system of I/O libraries which implement a flexible and capable I/O subsystem for newlib while dramatically reducing the amount of programming required to adapt the library to a new hardware platform. Like newlib, the librom system is licensed under a Berkeley-style license that places no restrictions on commercial use of the software.

1.4.6 Gnu make

Gnu make automates the rebuilding of object files and executables based upon the rules specified by the programmer in a make file.

1.4.7 Gnu documentation in HTML

The Gnu manuals are provided as HTML files, making it simple to search for help information and navigate quickly between different topics. Manuals are provided for all the programs on the CDROM.

1.5 Minimum System Requirements

1.5.1 Processor

Since these are command-line compiler tools and not interactive applications, there is no particular minimum requirement for processor speed; any system capable of running Windows 95, Windows 98, or Windows NT will serve adequately as a platform for running these tools. Of course, faster is always better!

1.5.2 RAM

As a bare minimum, you should have at least 12 MB available under Windows 95/98. Under Windows NT, we suggest at least 16 MB. Making more RAM available will significantly improve the performance of the tools.

1.5.3 Disk Space

The tools themselves take little hard drive space, but the total space required will vary considerably depending upon how many sets of run-time libraries are installed. About 168 MB of space is needed for a typical installation of the PowerPC tools and libraries.

1.6 Installation

Installation is easy: just run `xgcc32.exe` from the root directory of the CD-ROM; it will ask you a few questions and then do all the work for you. The setup program will ask you to select a destination directory for the compiler tools, and also to select which target microprocessor families and other components you want to support. You can install the entire toolset, or just the pieces you will need immediately; if your needs change later on, you can always re-run the installation program to install additional components.

2 Hello, GCC: compiling your first program with Gnu CC

Figure 2.1 shows a compile session from start to finish, including (a) setting the compiler's environment variables, (b) running the compiler itself, and (c) converting the linked executable to a hex/ASCII file. We'll discuss each step in a little more detail.

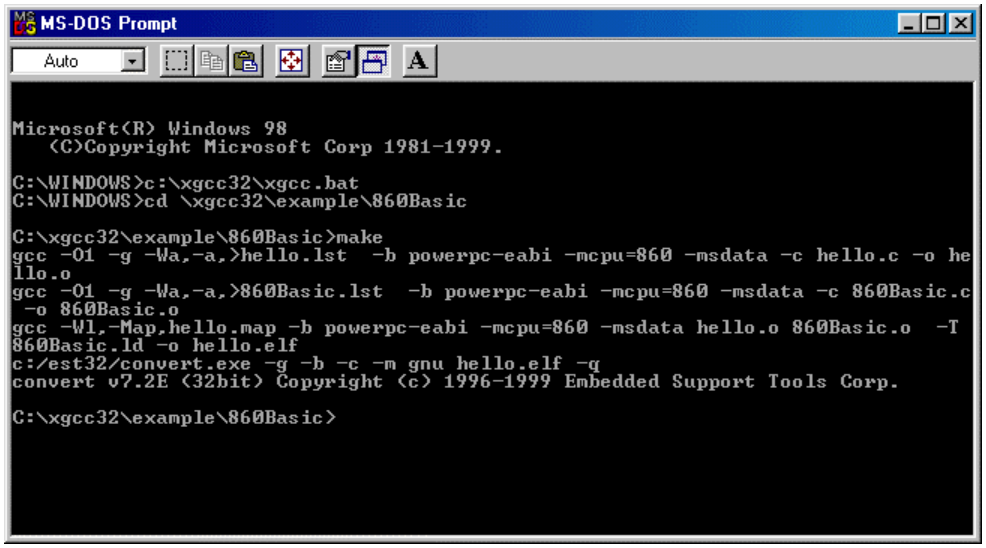

Figure 2.1: compiling 'Hello world' for the MDP860Basic board

2.1 Environment variables

Before any of the gnu tools may be run, the bin directory containing the gnu executables must be included on your PATH so COMMAND.COM (or CMD.EXE, for Windows NT) can find them. The batch file XGCC.BAT located in the root directory of the compiler installation (default c:\xgcc32) was provided for this purpose; open a console window and execute XGCC.BAT, and then you're ready to compile.

There are no other environment variables that must be set in order to run the tools. However, if you have other non-EST releases of the Gnu tools installed on your computer, your system may have environment variables defined for those tools. If these are set when the XGCC tools are executed, then they can cause problems where the EST tools may access the wrong directories for executables, libraries, header files, and so on. For this reason, the XGCC.BAT file provided with the EST release sets most of these non-essential environment variables to null strings to avoid these types of problems.

2.2 The command line

Now that the environment variables are set, you can compile one of the example programs included in the EXAMPLE subdirectory. We'll look at the source files hello.c and 860Basic.c in the directory C:\xgcc32\example\860Basic. This program will be run on the EST MPD860 Single-Board Computer:

Referring to Figure 2.1, sharp-eyed readers will see that we are building this project using the Gnu make utility, since this demo program is supplied with a makefile. Although make simplifies the task of building projects and keeping them up to date, in this section we want to look at the command lines executed by make to compile and link the program. Please refer to section 3.22 for more detail on how to use make.

```
C:\WINDOWS> cd \xgcc32\example\860Basic
C:\xgcc32\example\860Basic> gcc -O1 -g -Wa,-
a,>hello.lst  -b powerpc-eabi -mcpu=860 -msdata -c
hello.c -o hello.o
```

In the command line shown above, we have compiled the source file hello.c into an executable file hello.o. The program we invoked is gcc.exe, the Gnu CC driver

program. We gave `gcc` a bunch of options, and it ran several subprograms (the C preprocessor, the compiler proper, the assembler, and finally the gnu linker) to create the final output file.

Table 2.1 lists each of the options:

Command line Option	**Description**	**Detailed description**
`-O1`	Selects an optimization level of one (possible values are 0 through 4)	Section 3.6
`-g`	The compiler includes symbolic debugging information in the object file	Section 3.7
`-Wa,-a,>hello.lst`	Causes the assembler to create a listing and output it to a file named `hello.lst`	Section 3.18
`-b powerpc-eabi`	Tells `gcc` to use the PowerPC family compiler	Section 3.4
`-mcpu=860`	Instructs the compiler to use the CPU32 instruction set, and tells the linker to use run-time libraries compiled with this same option	Section 3.5
`-msdata`	Instructs the compiler to use the PowerPC EABI (Embedded Application Binary Interface) small data sections	Section 3.5
`-c`	Instructs `gcc.exe` to compile and assemble the C source file without linking it; this leaves the object file on disk for a subsequent link operation.	Section 3.9
`hello.c`	The source file we are compiling	
`-o hello.o`	The filename to give to the object file.	

Table 2.1: the components of the compiler command line

2.3 Linking an executable file

After the source files have been compiled into object files, the next step is to link them together into an executable. Linking the files resolves any external references between files into absolute address references. The linker command line looks like this:

```
C:\xgcc32\example\860Basic> gcc -Wl,-Map,hello.map -b powerpc-eabi -mcpu=860 -msdata hello.o 860Basic.o  -T 860Basic.ld -o hello.elf
```

Table 2.2 details each of the command line options passed during linking.

Command line Option	Description	Detailed description
`-Wl,-Map,hello.map`	Causes the linker to create a link map and output it to a file named `hello.map`	Section 3.18
`-b powerpc-eabi`	Tells `gcc` to use the PowerPC family compiler	Section 3.4
`-mcpu=860`	Instructs the compiler to use the CPU32 instruction set, and tells the linker to use run-time libraries compiled with this same option	Section 3.5
`-msdata`	Instructs the compiler to use the PowerPC EABI (Embedded Application Binary Interface) small data sections	Section 3.5
`hello.o 860Basic.o`	The object files we are linking	
`-T 860Basic.ld`	Passes the linker script `860Basic.ld` to the linker; this file lists any additional library files needed during linking, and specifies the location of code and data memory	Section 3.15
`-o hello.elf`	The filename to give to the ELF executable file.	

Table 2.2: Linker command line options

2.4 Creating a symbol file and download file for VisionClick

Once we have an executable program ready to debug, we must convert it to BDX format in order to load it into the VisionClick debugger. In addition, a symbol file must be created in order to be able to perform source-level debugging in VisionClick. Both these files are created with the `CONVERT` utility which is supplied with VisionClick. The command line for our example looks like this:

```
c:/est32/convert.exe -g -b -c -m gnu hello.elf –q
```

2.5 Downloading to the Target

To download the program to the target, we start VisionClick and create a new project file. We'll call this project `hello.prj`. Figure 2.2 shows the settings we entered for this project.

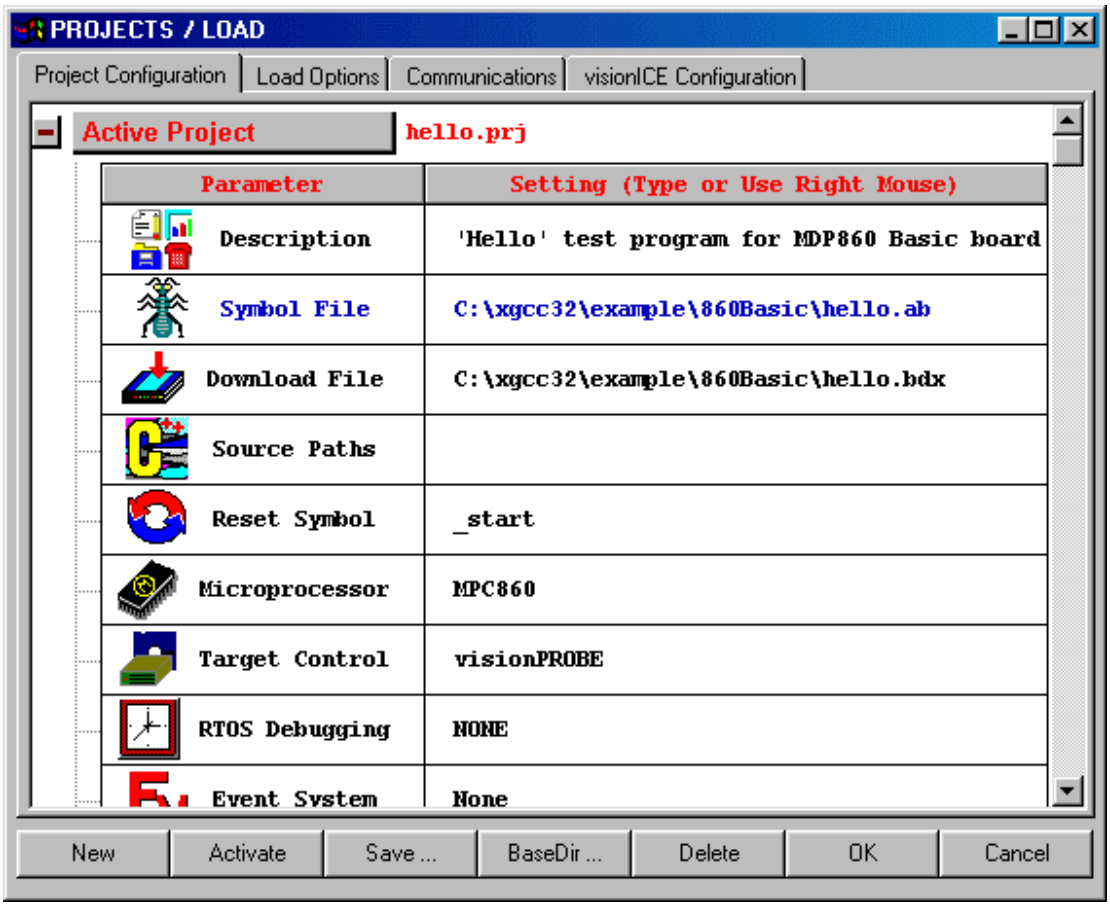

Figure 2.2:Project settings for hello.prj

We then download the program by clicking the OK button and pressing F11. The download dialog will confirm that the program was loaded into memory.

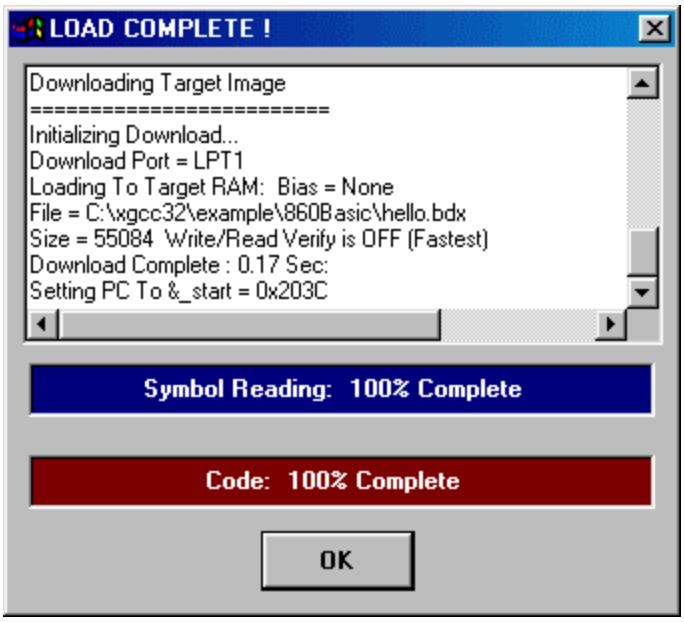

Figure 2.3: Downloading the BDX file to the target

Since this demo program interacts with a console on the SMC1 serial port, make sure to open VisionClick's I/O window and configure it for 9600 baud (right click on the I/O window to do this). In Figure 2.4 we are using COM1 on the PC; set this parameter for the COM port that is appropriate for your system. Also, ensure that the serial cable is connected properly to your PC.

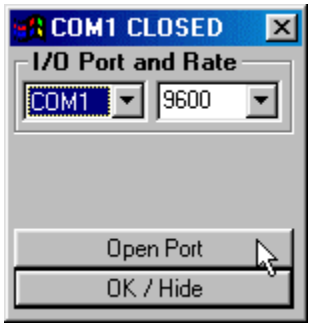

Figure 2.4: Select the appropriate COM port for your PC and set it to 9600 baud.

And now the moment of truth: we press F5 to run the program, and… the MPC860 says `Hello, world!` Success! We do a victory dance at the workbench, and then quickly compose ourselves and get back to work.

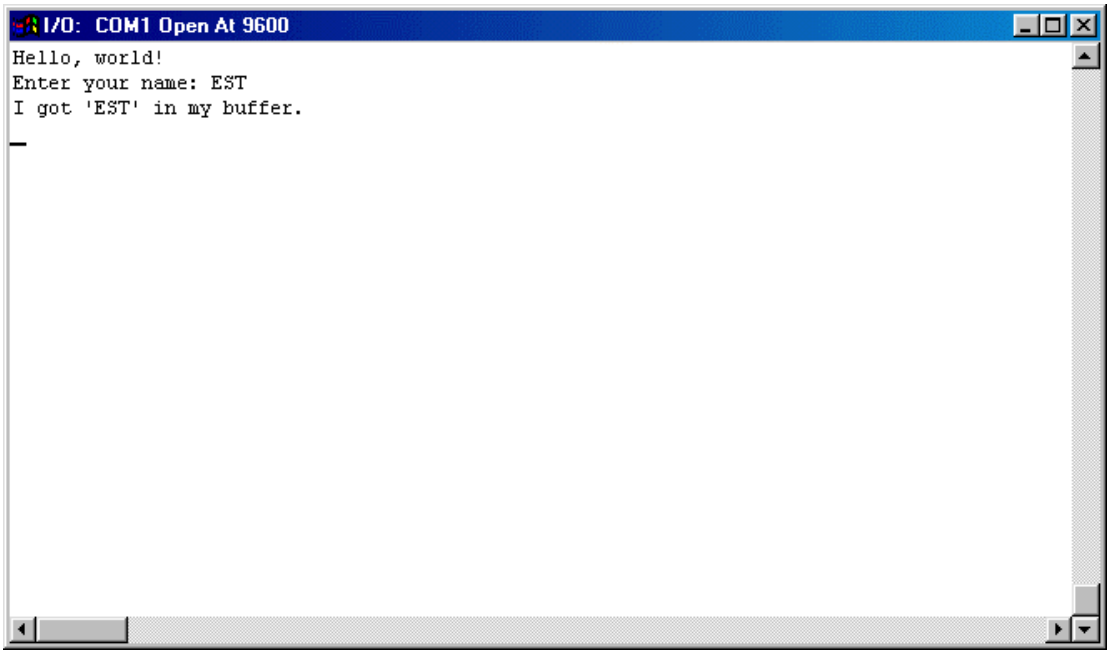

Figure 2.5: running the program on the target

We have moved very quickly through a trivial example in this section, in order that we could focus on the big picture. The next section goes into more detail on each of the steps, and will hopefully get you to the point where you can start work on developing your own embedded code with Gnu CC.

3 Running the Gnu tools

We now go into a little more detail on the most commonly-used command line options for the gnu tools. This is not intended as, nor could it be, a replacement for the Gnu manuals; rather we try to cover only the options that are most often used.

3.1 Program names

You can have multiple installations of the gnu tools on your hard drive at the same time, for example you can have one set which generates code for the Motorola 68K family along side another set for the PowerPC, Hitachi SH, etc. Each tool set comes with a gaggle of compilers, assemblers, linkers, utilities, and on and on. In order to easily separate them, each program is prepended with the configuration name of the target that it supports. For example, the linker, which is canonically named `ld`, resides in the file `m68k-elf-ld.exe` for the Motorola 68k version, `powerpc-eabi-ld.exe` for PowerPC, `sh-elf-ld.exe` for the SH-3, etc. Similiarly, the assembler is named `68k-as.exe`, `powerpc-eabi-as.exe`, `sh-elf-as.exe` etc.

3.2 How gcc controls the compilation and link process

`gcc.exe` is the program that you will almost always use to initiate a compile or link session. However gcc.exe does not actually do the work itself; it is actually a driver program which

examines the options and filenames passed on its command line, and then calls other programs to perform the requested operations.

Figure 3.1 shows the relationship between `gcc.exe` and the other programs, and how the various types of input files are processed by each tool. When `gcc.exe` calls these other tools, it will construct a command line that is based upon the contents of the original command line passed to `gcc.exe`, modified as dictated by a script in the `specs` file for the target processor. The `specs` file is located in the directory `c:\xgcc32\lib\gcc-lib\<target name>\<compiler version>`, where `<target name>` is the name of the target compiler configuration (eg `powerpc-eabi` for PowerPC), and `<compiler version>` is the version number of the compiler being used (eg. `2.95.2`).

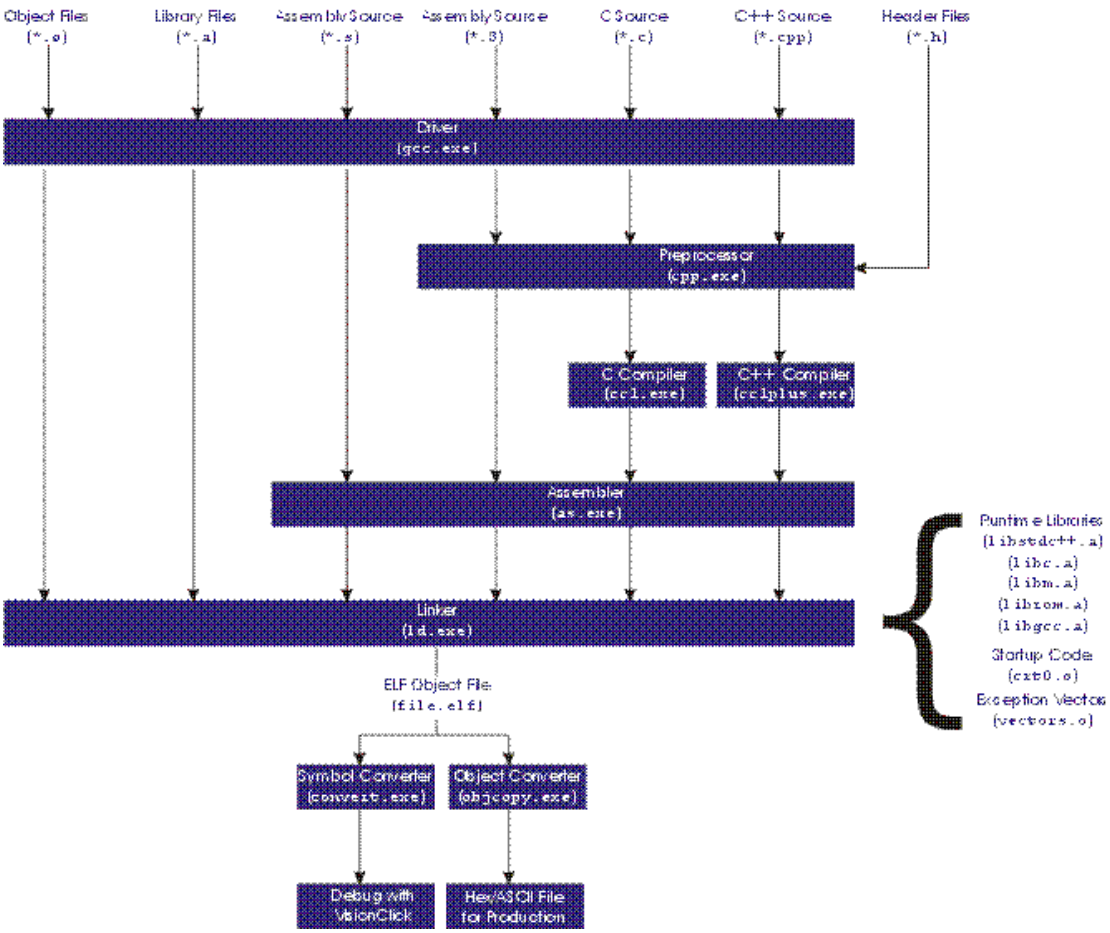

Figure 3.1: How files are processed by gcc

3.3 gcc handles C, C++, assembly language, object files, and libraries

gcc can accept any mix of C, C++, assembly language, object files and libraries on the command line; it handles each one according to the filename extension. Table 3.1 defines the key filename extensions that are recognized by gcc, and what it does with them:

Extension	File type	Operation
.c	C source file	`gcc` runs the file through the preprocessor and then the C compiler
.cpp, .cc, .cxx	C++ source file	`gcc` runs the file through the preprocessor and then the C++ compiler.
.s	Assembly source file	`gcc` passes the file to the assembler unchanged
.S	Assembly source file	Gcc runs the file through the preprocessor and then the assembler
.o	Object file	Gcc passes the file to the linker unchanged
.a	Library file	Gcc passes the file to the linker unchanged

Table 3.1: Filename extensions recognized by gcc

Looking at this table, it is interesting to note that gcc can run an assembly-language file through the C preprocessor before passing the resulting output to the assembler proper. This means that you can use C-style macros in your assembly source, which can be a major convenience for things like defining repetitive structures in memory, or accessing constants that are used in both C and assembly files. It also can simplify the interface between C and assembly language functions, as discussed in section 4.1.

3.4 Selecting the target system (-b <name>)

If Gnu CC can compile code for many different target systems, then how do we know which one we are targeting if there is only one ‘gcc’ command? That’s what the –b option does. For example, the `–b` option, followed by the symbolic identifier `m68k-elf`, tells gcc to run the `m68k-elf` version of the compiler, assembler, and linker tools[1].

Each different target architecture is assigned an identifier that uniquely identifies the *processor*, *system manufacturer*, and *operating system* or *object format*. In traditional Unix-

[1] They are in \XGCC32\m68k-elf\bin (68k) and \XGCC32\powerpc-eabi\bin (PowerPC).

based Gnu compilers, these identifiers are made up of three words separated by hyphens, in the order shown above (example: *m68k-sun-sunos4*). In the EST release of these compilers, we picked the configurations that were most applicable to embedded development and used abbreviated names where possible to keep program names short.

Table 2-1 lists the EST identifiers with the 'official' Gnu CC identifier, and the characteristics of each compiler port.

EST ID	Full ID	Comments
m68k-elf	m68k-unknown-elf	Embedded Motorola 68k family; ELF object file format; no operating system
powerpc-eabi	powerpc-unknown-eabi	Motorola/IBM Power PC architecture; EABI (Embedded Application Binary Interface) calling convention; ELF file format; no operating system

Table 3.2: Gnu CC identifiers for each microprocessor architecture

When you specify a target identifier with the `-b` option, gcc turns that identifier into a directory name of the form `(base directory)\(identifier)\bin` and looks for the tools in that subdirectory. If the directory does not exist, an error message is reported, like this:

```
GCC.EXE: installation problem, cannot execute
'cc1': No such file or directory
```

If you see this message during a compilation, then one of the first things you should check is the target name used with the `-b` option, to make sure it's correct.

3.5 Target-specific options (-m<xxx>)

Within each target microprocessor family, there can be several different versions of the instruction set implemented on different family members. In addition, there are often other features that are specific to a particular family of microprocessors, which require special command-line options to control. These options are set via the `-m<xxx>` family of command line options.

Table 3.3 contains a list of the most important target-specific options that are available in the 68k port of Gnu CC. These options (except for `-mshort` and `–mrtd`) also apply to the assembler.

`-m68000`	Generates code for the original MC68000 implementation; also valid for MC68008, MC68302 through MC68328, and MC68356
`-mcpu32`	Generates code for the any device based upon the CPU32 processor core; this includes MC68330 through MC68396 (except the 68356; see `–m68000`)
`-m68020`	Generates code for the MC68020. This is the default if no –m option is specified.
`-m68030`	Generates code for the MC68030.
`-m68040`	Generates code for the MC68040.
`-m68020-040`	Generates code that will run on any device from MC68020 through MC68040, ie: code is optimized for the MC68040, but none of the 68040-only opcodes are used.
`-m68060`	Generates code for the MC68060.
`-m5200`	Generates code for any device based upon the Coldfire V2 (and above) core. This includes all devices with part numbers in the MCF52xx and MCF53xx ranges.
`-msoft-float`	Prevents generation of hardware floating-point instructions, for those processors which can support it.
`-mhard-float`	Forces generation of hardware floating-point instructions, even if the target processor cannot support a floating-point coprocessor.
`-mshort`	Forces `int` variables and function parameters to be 16 bits wide rather than the default of 32 bits. Note: although the compiler will accept this option for all targets, the XGCC CD-ROM does not contain libraries built with this option for MC68060 or Coldfire targets.
`-mrtd`	Specifies use of the `RTD` instruction when returning from functions, rather than the default `RTS`. `RTD` will result in slightly

	smaller and faster code, since it automatically reclaims the stack space allocated for function parameters; however, the programmer must be careful to only use this option if all functions are declared before they are called, since the called function must remove the exact same amount of stack space that the caller allocated. Although this option is accepted by the compiler for all targets except MC68000, the XGCC CD-ROM only contains libraries built with this option for cpu32 and MC68020 through MC68040.

Table 3.3: Processor-specific options for the 68k compiler

Table 3.4 documents the key processor-specific options available in the PowerPC port of Gnu CC.

`-mcpu=xxx`	Selects the processor variant in use. `xxx` may be one of '403', '505', '601', '602', '603', '604', '620', '821', or '860'. If xxx is one of '403', '821', or '860', then software floating point is also selected (see –msoft-float below), otherwise hardware floating point is selected.
`-mtune=xxx`	'Tunes' the instruction scheduling for the processor variant `xxx`. `xxx` is specified exactly the same as in `–mcpu` above.
`-mlittle` `–mlittle-endian`	Generate code that executes in little-endian mode.
`-mbig` `–mbig-endian`	Generate code that executes in big-endian mode (the default).
`-msoft-float`	Prevents generation of hardware floating-point instructions, for those processors which can support it. This is the default for `–mcpu=403`, `-mcpu=821`, and `–mcpu=860`.
`-mhard-float`	Forces generation of hardware floating-point instructions, even if the target processor does not implement hardware floating-point support. This is the default for all CPU types except when the options `–mcpu=403`, `-mcpu=821`, and `–mcpu=860` are specified.

`-msdata` `-msdata=eabi`	Causes the compiler to place all variables smaller than a certain size threshold into one of the small data sections `.data` (initialized variables), `.sbss` (uninitialized variables), or `.sdata2` (`const` variables). In addition, variables in these sections will be accessed using the 'load indexed' and 'store indexed' instructions, using r13 (or r2 in the case of `.sdata2`) as a base address register. This typically results in smaller and faster code, however the size of `.sdata` and `.sbss` combined cannot exceed 64K bytes. Variables which are larger than the size limit are placed into the `.data`, `.bss`, or `.rodata` sections as appropriate. The size threshold defaults to 8 bytes, and may be changed with the `–Gn` option described below.
`-Gn`	Sets the maximum size of variables that will be placed in the small data sections, as described above for the `–msdata` option. The default value is 8. If this option is passed to the compiler, the same value must also be passed to the linker.
`-mcall-aix`	Instructs the compiler to use the AIX calling convention rather than the default System V.4/EABI calling convention. The two calling conventions are incompatible; either all files to be linked together in a program must use `–mcall-aix`, or none of them may use it.

Table 3.4: Processor-specific options for the PowerPC compiler

3.6 Specifying the optimization level (-O<n>)

The Gnu CC compiler supports several levels of optimization. Optimization is selected by specifying `–O<n>` on the command line, where <n> is a number from 0 through 4 (can be up to 6 on some targets). As is probably obvious, increasing numbers mean higher and more sophisticated levels of optimization. A good level to use for debugging is `–O1`; after

debugging is complete, re-building your code with higher optimization will give you slightly better performance in the finished program.

High levels of optimization can cause funny things to happen when you run your program under control of the debugger. Sometimes variables in a function will only contain valid data when they're being used, and junk before and after; sometimes they may even be completely eliminated by the compiler, causing the debugger to report that it doesn't exist. In addition, the code may not execute in the way that you envisioned; the compiler may sometimes re-arrange loops, move certain statements out of loops into the main body of a function, or may not generate any code for some lines of source. Typically these effects will be minimal or non-existent with optimization set to `–O1`.

3.7 Enabling generation of debug info (-g)

If you plan to test your code under the VisionClick debugger, you will want to include the option `–g` on the compiler's command line. This instructs the compiler to add symbolic debug information to the compiled file, including symbol names and locations, source filenames and line numbers, and the definition of `struct`s and user data types in the program.

This option does not change the actual processor code that is generated by the compiler, it only adds extra debugging information to the object file. For this reason, it's often used even when not planning to debug the code; the extra debug information is sometimes useful because can be accessed by some of the other Gnu CC tools. For example, if debug information is present in the file, the Gnu assembler is able to generate an assembly listing file which shows the original C or C++ source code intermixed with the compiler-generated assembly language code.

Assembly-language source files may also have debug info generated for them, but the option is slightly different: `-Wa,-gstabs`. Refer to section 3.18 for more information on this option.

Please refer to section 3.6 [Specifying the optimization level (-O<n>)] for a discussion on debugging optimized code.

3.8 Position-Independent Code (-fpic, -fPIC)

In some applications it may be desirable for the compiler to generate code that uses relative addressing, rather than absolute addresses, to access functions and variables in the program. This is referred to as Position-Independent Code (PIC). Two forms of PIC are supported by the Gnu tools; typically, `-fpic` generates code which is smaller and executes faster, but limits the total size and/or number of functions in a program, while `-fPIC` generates code which removes these limitations but typically uses larger function prologues and executes a little more slowly.

In contrast to the standard FSF release of the Gnu tools, EST have made several enhancements to the tools which improve support for Position-Independent Code; including:

- Code and data are independently relocatable at run-time
- Program and data relocation are fully supported in EST's startup code (`crt0.o`)

For more details on the use of PIC in embedded applications, please refer to section 4.6.

3.9 Outputting an object file (-c) or assembly language file (-S)

The default action of `gcc` is to try to compile, assemble, and link a program into a finished executable. While this is fine for small programs, for larger projects it would be cumbersome to try to specify a large number of source files all on one command line. In addition, recompiling all source files when only one or two have changed is wasteful and can take a lot of time, even on modern computers. In this case, it makes sense to compile each file separately and then link all the resulting object files. Specifying `-c` on the command line will cause `gcc` to stop after compiling and assembling the specified source files, leaving the object files ready for a subsequent link operation.

Sometimes you will want to see the assembly language code which is generated by the compiler. Specifying the `-S` (that's an upper-case 'S') will cause `gcc` to stop after compiling the C/C++ source, producing an assembly language file with the same name as the C/C++ source file, but with the extension '.s'. See section 3.16 to find out how to create an assembly listing file without stopping after compilation.

3.10 Specifying directories for include files (-I, -I-)

`gcc` maintains a list of directories which contain 'system' include files; each time the `#include <filename>` directive is used in a source file, each directory in this list is searched to find the specified file. You can add a directory to this list by using the `-I<dirname>` option on the command line. This option adds the specified directory to the *head* of the list, i.e. it is searched first before any predefined directories. Note that the directory name must immediately follow the `–I` without any spaces separating them. If the directory name itself has spaces in it, then the entire option including the `–I` should be enclosed in double quotes.

If you follow one or more `–I` directives with `–I-`, then all of the preceding directories specified with `–I` are added to the list of *user* directories, rather than system directories. User directories are searched when the `#include "filename"` directive is encountered in a source file.

3.11 Creating dependency files (-MMD)

When compilation is performed under control of the `make` utility, `make` needs to know all the source files upon which an object file depends. When compiling C and C++ source, any file brought in by the `#include "filename"` directive also becomes a dependency of the object file (meaning that the object file should be re-built if any of the included files have changed). However, in a medium to large-scale project it can be very tedious and error-prone to manually update the makefile. Nested include files make this task even more difficult.

Gnu CC's `–MMD` option dramatically simplifies this task. When `–MMD` is specified on the compiler command line, a dependency file is created with the same name as the object file but with a filename extension of `.d`. This file contains a snippet of text that lists the source files (main and include files) upon which the object file is dependent. It may be included into the body of the main makefile, either manually with a text editor or (the preferred way) at compile time through the use of Gnu make's `include` directive. Thus each time the file is re-compiled, `gcc` re-generates the list of dependencies automatically without manual intervention.

See section 4.8.2 for more detail on compiling with the `make` utility.

3.12 Define a macro (-D<name>)

You can define macro's on the command line using the `–D<name>` option. In this form, the macro is defined to the value 1. The other form of this option, `-D<option>=<value>`, defines the macro to the specified value. For example,

```
-DDEBUGGING
```

defines the symbol `DEBUGGING` to have the value 1, while

```
-DDEBUGGING=0
```

defines the symbol `DEBUGGING` to have the value 0.

It can be helpful to surround this option in double quotes in order to avoid text strings being accidently interpreted as another part of the command line. For example if you want to define the symbol `INT` to have the value `-1`, this can be done in the following option on the command line:

```
"-DINT=-1"
```

One symbol that is often defined on the command line is `NDEBUG`. The ANSI C standard says that defining `NDEBUG` disables the `assert` macro, so since it's already part of the spec, you might as well use it for your own debugging code as well

When you're compiling your program in preparation for debugging, leave `NDEBUG` undefined; this symbol can be tested by a C preprocessor sequence, and debug code can be conditionally compiled into the program, as in the example below:

```
#ifndef    NDEBUG

/* this code is used for debugging only */
printf ("Counter is %d\n", Counter);
#endif
```

When debugging is complete, and it's time to re-build the program to burn EPROM's, add the option `–DNDEBUG` to the compiler command line and the call to `printf()` will be left out of the build.

3.13 Verbose mode (-v)

Normally `gcc` does its work quietly, without displaying any messages except to report warnings or errors. The `-v` command line option will cause `gcc` to display the exact commands and options that it uses to do its job. This can be useful when trying to diagnose compilation or linking problems, or if you just want to see how the whole system works.

3.14 Enabling warning messages (-Wall)

Gnu CC does a good job of checking your source for potential or real problems, and letting you know about them – if you tell it to. Gnu CC has around a billion[2] individual warning messages, and almost every one of them can be enabled or disabled with a variation on the `-W` command line option. If you need to enable or disable a specific warning, we suggest you refer to section 8 of the Gnu CC manual for the specific options available. However, Gnu CC also groups several of the most useful ones together under a 'blanket' option: `-Wall` The Gnu CC manual puts it this way:

> This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.

Some of the more useful checks performed are:

- `printf()` format strings match the arguments passed
- Nested comments
- Implicit type declarations of variables or functions
- Local variables possibly used before being initialized
- Possibly incorrect comparision of signed and unsigned values

Unless you're compiling legacy code which you know works and would be too much effort to edit, you might want to consider using the `-Wall` option every time you compile with GCC; it can help catch problems which are sometimes hard to track down during debugging.

[2] OK, that's a bit of an exaggeration, but there are quite a few. A single –Wall enables all the useful ones.

3.15 Specifying a linker script (-T <filename>)

In most embedded systems, the memory map is complicated enough that it would be impractical to try to specify all the required information on the linker command line. In this case, the linker may be handed a script file that defines some or all of the required parameters. The `-T` option is used to specify the name of this linker script file when linking via the `gcc` driver.

Section 3.21 discusses the layout of linker script files in detail.

3.16 Specifying library files (-l<libname>)

When you link a program, it's almost certain that library files will be used to supply code modules needed at runtime. You can create your own library files and specify them on the link command line just like any other file. You can also specify library files with the `-l<libname>` option, in which case two additional things happen:

- The library name specified by the `<lib>` part of the option is expanded to the form `lib<libname>.a`; for example, if you specified `-lc` on the command name, the actual file name that `gcc` looks for is `libc.a`.
- `gcc` searches a list of library directories for the library file.

This is handy for libraries that are used in a large number of different projects. You can put the library in one global directory which is on the library search path, and then each project that needs that library can reference it with the shortened `-l<libname>` syntax. The linker script files `ram.ld` and `rom.ld` supplied on the CD-ROM use this syntax for the standard C runtime library files.

See section 3.17 for details on how directories are added to this search list.

3.17 Specifying directories for library files (-L<dirname>)

When you are using the `-l<libname>` option to specify library files, you may want to add your own library directories to the list of directories searched by `gcc`. The `-L<dirname>`

does just that. The directory name <dirname> must follow the –L option with no space in between. For example, if you have a set of libraries stored in C:\PROJECTS\LIBS, this directory will be searched by adding -LC:\PROJECTS\LIBS to the linker command line.

3.18 Passing options to the assembler and linker (-Wa, -Wl)

If you need to pass an option directly to the assembler or linker, gcc provides an escape mechanism to support this. passing –Wa,<options> to gcc will cause it to pass <options> directly to the assembler without any changes. For example, if you wanted to pass the option –a to the assembler, to cause it to output a listing file, the gcc option would be –Wa,-a.

Did we say 'without *any* changes'? OK, there is one change made. When you want to pass multiple options to the assembler, or options that require a parameter, they can all be specified in order with a single –Wa, option to gcc. Each option must be separated by a comma, and gcc will replace each comma with a space before passing the options to the assembler. Without this feature, you would need to specify the –Wa, prefix for each assembler option, which would get pretty tedious.

3.18.1 Common Assembler Options

Here we present the assembler options that are most commonly used, and the form for each option as presented to gcc on the command line.

Assembler Option	Description	Example (using gcc)
-a	Create listing file	-Wa,-a
-gstabs	Output debug info	-Wa,-gstabs
-I <dirname>	Add directory to search list for .include directive	-Wa,-I,C:\ASM

Table 3.5: Frequently-used assembler options

3.18.2 Common Linker Options

Similar to the assembler, `gcc` may also be used to pass options directly to the linker. The `gcc` option to do this is `–Wl,<options>`.

A table of the most commonly-used linker options appears below.

Linker Option	Description	Example (using gcc)
`-Map <filename>`	Create map listing in file	`-Wl,-Map,project.map`
`--defsym <name>=<value>`	Define the value of a symbol	`-Wl,--defsym,SYPCR=0xfffa21`
`-u <symbol>`	Declares the symbol to be undefined; this may be used to force the linker to bring in modules from a library file to define to symbol	`-Wl,-u,fir`
`-t`	Lists the name of each input file as it is processed by the linker	`-Wl,-t`

Table 3.6: Frequently-used linker options

3.19 Assembling & Linking via gcc vs. invoking the tools directly

As mentioned earlier, you can pass any mix of C, C++, assembly and object files to gcc and it will call the correct tools to process them and end up with a compiled and linked executable. But why would you want to do it this way? For example, if you want to run an assembly language file through the assembler, why not just invoke the assembler directly rather than having gcc do the extra work?

There’s nothing stopping you from doing it this way, but (as always) there are tradeoffs. If you invoke the tools directly, you have complete control over every option passed to the tool; however, you are also responsible for making sure that every single command line option required by the target system is specified properly. If you use gcc to do the work, it can add these options automatically. Some examples of the parameters automatically set by gcc:

- Include and library paths
- Adding required files to the command line (for example, crt0.o for linking)
- Selecting the appropriate library files based upon compilation options (‘multilibbing’)

Our recommendation is to *always use gcc* to do the work for you, unless you have a special requirement that demands the extra control provided by invoking the tools directly.

3.20 The Gnu assembler

The Gnu assembler has its roots in the world of Unix operating systems, and uses syntax that may be somewhat unfamiliar to embedded developers. This is unfortunate, since it means that some existing assembly language files cannot be assembled successfully without some editing. However, the differences in the Gnu syntax are easily learned; and hopefully we’ll have to use less and less assembly code as compilers and embedded processors get smarter and more powerful.

The following sections describe some of the syntax elements that differ significantly from the microprocessor manufacturer’s specified syntax, or differ from common practice in embedded systems. It is not meant to be a complete description of the assembler’s features; the Gnu AS manual documents the complete features of the program.

3.20.1 Comments

The Gnu assembler supports both block comments and line comments. Block comments are C-style, beginning with `/*` and closing with `*/`. Line comments start with a designated comment character, which can vary for different target microprocessors, and continue to the end of the line. For the M68K family, the line comment character is the vertical bar, `|`; for the Power PC, it’s the pound sign, #.

3.20.2 Statements

The Gnu assembler can accommodate multiple statements per source line; each statement is separated with a semicolon (`;`) or ‘at’ sign (`@`).

3.20.3 Escapes in character strings

The Gnu assembler accepts C-style character escapes, such as `\n` (newline), `\b` (backspace), `\r` (carriage return), etc. In addition, statements may be continued over multiple lines be placing a backslash (\) immediately before the end-of-line character.

3.20.4 Local symbols

A symbol starting with `.L` (a period, followed by an upper-case 'L') is a local symbol, which will not be visible to other modules when they are linked together.

3.20.5 Assembler Directives

Directives always start with a dot, for example `.extern`.

3.20.6 Register names

Register names must be prefixed with a percent sign, to avoid confusing them with symbol names. For example, to load an immediate value into address register `A0` on the M68K family, the assembly code would look like `move.l #SCI,%a0`.

For PowerPC, the numeric register number may be used by itself rather than prepending an 'r'; for example,

```
addi 3,3,1
```

is equivalent to the more verbose

```
addi %r3,%r3,1
```

3.21 Linker scripts

While it is possible to specify on the command line everything the linker needs to know, for most embedded applications this would be very cumbersome and error-prone. The more common solution is to create a linker script file that specifies all the parameters which don't change very often, and then put the project-specific parameters on the linker command line.

3.21.1 Common Linker Directives

The standard linker scripts provided on the XGCC CD-ROM will cover most typical embedded requirements. This section documents some of the linker directives used in those script files, to aid in understanding their operation. If you need to write your own script to meet a special requirement, we recommend you refer to the Gnu linker manual for a detailed reference on each directive.

The tasks handled in a linker script typically boil down to the following:

- Specifying the names of run-time libraries, and the search paths for those libraries
- Defining default values for symbols
- Specifying the memory layout of the program
- Defining options on sections in the output file
- Adding startup and shutdown support to the program

We will cover the first four topics in the following subsections. Startup and shutdown is discussed in section 4.4.1, ‘Initializing peripherals upon startup’.

3.21.1.1 Specifying the names of run-time libraries, and the search paths for those libraries

In a typical embedded application, there are library routines that are used in every single program that is compiled; for example, the standard C runtime library. Since these are used so often, it makes sense to reference these libraries in the script file, so they don’t have to be specified on the command line every time a new program is linked.

As an example, here are the first few lines of EST’s linker script `rom.ld` for the 68K:

```
OUTPUT_ARCH(m68k)
SEARCH_DIR(.)
INPUT(vectors.o)
GROUP(-ltrgt -lrom -lc -lgcc)
```

The `OUTPUT_ARCH()` directive specifies that we are generating the M68K variant of the object file format. `SEARCH_DIR(.)` adds the current working directory to the linker search path; `INPUT(vectors.o)` tells the linker to include the object file `vectors.o` in the final program, to define the exception vectors for the program.

The `GROUP()` directive tells the linker to search the group of library files listed repeatedly when undefined symbols exist. In the example above, several system libraries are referenced; The libraries are searched until no more undefined symbols remain, or until the linker detects that no more symbols were defined in the last pass through the group. This is extremely useful where inter-library dependencies exist; for example, a module in library `libtrgt.a` might reference a symbol which is defined in `librom.a`, which in turn might create a new reference to a symbol defined in `libtrgt.a` which was not included in the previous pass through the files. If these library files were simply referenced on the command line, or in the script file through use of the `INPUT()` directive, the linker would make only a single pass through the files and then find that the newly-referenced symbol was still undefined, causing an error and an unsuccessful link.

3.21.1.2 Defining the default values for symbols

In some cases it can be very useful to allow user code to define a symbol, but to define a default value to it if user code does not do so. This capability is implemented by the linker's `PROVIDE()` directive.

An example of this is the symbol `crt0_flags`, an integer variable accessed by the startup code in `crt0.S`. User code may define a variable with this name in order to control the operation of the startup code which runs before main () is called. However, if no such symbol exists in user code, the linker script file statement

```
PROVIDE(crt0_flags = 0);
```

defines the symbol with its default value of zero, facilitating a successful link. The startup code is written to test `crt0_flags` for this default value and take the appropriate action.

3.21.1.3 Specifying the memory layout of the program

A program is broken into different sections, each one containing a different type of information. Three of the most common section names are `.text`, `.data`, and `.bss`. These names originated in the Unix operating system and today are used in many systems, including embedded applications. The `.text` section is used to hold program code; the `.data` section is used to hold initialized data; and the `.bss` section contains uninitialized data, and quite often the stack space as well.

This is probably a gross oversimplification of the process, but essentially from the point of view of memory sections, the goal of the linking process in is to merge the text, data, and bss sections from each input file into a single text, data, and bss section in a single output file, while resolving references to undefined symbols along the way. The rules for merging and assigning memory addresses to each section are provided by the linker script's `SECTIONS` directive.

3.21.1.3.1 The SECTIONS directive

The basic syntax of the SECTIONS directive is as follows:

```
SECTIONS {
section1 [options] : {
        contents
        }
section2 [options] : {
        contents
        }
section3 [options] : {
        contents
        }
    .
    .
    .
}
```

Figure 3.2: Basic syntax of the linker's SECTIONS directive

The section names `section1`, `section2`, `section3` etc define the name of a section that will appear in the output file. Inside the brackets (where the word `contents` appears in the example above) are listed all the elements from the input files that will be placed in that section.

Section contents are specified as one or more lines in the format `filename(section)`. `filename` specifies the file name of an input object or library file, while `section` specifies a section name within that object or library file. For example, the text section of the file `vectors.o` is specified as `vectors.o(text)`.

In addition, either or both the filename and section name may be a wildcard character `*`. When replacing the filename, the wildcard indicates that the specified section of all input files should go into this output section; similarly, when used in place of the section name, the wildcard causes all sections of the file name to be placed in the output section.

For example, the `SECTIONS` directive of a very simple linker script might look like this:

```
SECTIONS {
.text : {
        *(.text)
        }
.data : {
        *(.data)
        }
.bss : {
        *(.bss)
        }
    .
    .
    .
}
```

This script will create a single output file which contains three sections, named `.text`, `.data`, and `.bss`. The `.text` section will appear first in memory, starting at location 0, followed immediately by the `.data` section. The `.bss` section will not allocate any space in the output file, but addresses will be assigned to any symbols in this section starting at the first free location after the end of the `.data` section.

Each output section will contain the sum of the contents of the same section in the input files, e.g. the output file's `.text` section will contain all data in all `.text` sections of the input files, and the same for `.data` and `.bss`. Any sections with any other names in the input files will not appear in the output file.

3.21.1.4 Defining options on output sections

The linker script shown in the previous section might work on some system, but typical embedded applications need more flexibility; for example, each section may have to be assigned to a particular starting address to match the locations of RAM and ROM memory in

the target system. These types of features are specified by a set of options that can be added to the declaration of each output section.

3.21.1.4.1 Setting the section's start address

The section's start address is defined by placing it immediately after the output section's name. Here is the example from section 3.21.1.3.1, this time with the `.data`. section assigned to address 0x8000:

```
SECTIONS {
.text : {
        *(.text)
        }
.data 0x8000 : {
        *(.data)
        }
.bss : {
        *(.bss)
        }
    .
    .
    .
}
```

3.21.1.4.2 Section alignment

Very often a section will need to be aligned to a certain modulo boundary; for example, in most members of the Motorola M68K family, opcodes must be aligned on a word (16-bit) boundary. This is achieved with the `BLOCK()` option. Here again is our sample script, modified to align opcodes as required:

```
SECTIONS {
.text BLOCK (2): {
        *(.text)
        }
.data 0x8000 : {
        *(.data)
        }
```

```
.bss : {
        *(.bss)
        }
    .
    .
    .
}
```

3.21.1.4.3 Defining symbols

A symbol is defined very easily, with the syntax

```
<name> = <value>;
```

The expression must be ended by a semicolon, to mark the end of the assignment.

The special symbol `.` is used to represent the current memory address. It may be changed by assigning a new value to it, or it may be used in an expression to assign values to other symbols.

In the following example, we to assign the symbol `_etext` to the address immediately following the end of the `.text` section.

```
SECTIONS {
.text BLOCK (2): {
        *(.text)
        _etext = .;
        }
.data 0x8000 : {
        *(.data)
        }
.bss : {
        *(.bss)
        }
    .
    .
    .
}
```

3.21.1.4.4 Placing arbitrary data in a section

The linker allows us to place an arbitrary byte, word, or long-word value anywhere in any output section that we define. This is done with the expressions `BYTE()`, `SHORT()`, and `LONG()` respectively.

For example, let's say that we wanted to put a `JMP START` instruction at the end of our 68K's code section; we could do it this way:

```
SECTIONS {
.text BLOCK (2): {
        (.text)
        SHORT(0x4ef9) /* jmp */
        LONG(start)
        _etext = .;
        }
.data 0x8000 : {
        *(.data)
        }
.bss : {
        *(.bss)
        }
    .
    .
    .
}
```

3.21.2 The standard linker scripts rom.ld and ram.ld

We have provided two enhanced scripts, along with object files to define an exception vector table and an enhanced crt0 startup module, to address the needs of typical embedded applications.

The script `rom.ld` supports targets where code is stored in a block of read-only memory (flash EPROM, UV EPROM, mask ROM, etc) and `.data` and `.bss` sections go into a single block of RAM. The other script file, `ram.ld`, supports systems where all sections (`.text`, `.data`, `.bss`, etc) go into a single contiguous block of RAM. This script is most useful when debugging a ROM-based system.

These scripts have the following common features:

- The symbols `__ram_start` and `__ram_size` define the start and size of the available RAM in the target system. These symbols may be defined on the linker command line to avoid the need to customize the script file for each new project.
- The symbol `__stack_size` defines the amount of RAM that is allocated to the processor's stack; stack space is allocated at the very top of free RAM (after the `.bss` section). Any remaining RAM after the end of the `.bss` section becomes part of the heap space which is available for allocation via calls to `malloc()` and `free()` (or the `new` and `delete` operators in C++).
- Support is provided for user-defined interrupt and exception vectors. Any vectors that are not defined by user code will automatically point to a default handler located in the crt0 startup module.

In addition, the `rom.ld` uses the symbols `__rom_start` and `__rom_size` to define the starting address and size of the system's ROM space, and supports initialization (in crt0) of data RAM from a ROM image.

These scripts work in conjunction with EST's `librom` runtime libraries and enhanced `crt0` startup module. For details on how to link applications using these scripts, please refer to section 4.11.7, 'Building and linking application programs with lib'.

3.22 Building projects with Gnu Make

3.22.1 Make basics

Make is a utility that controls the rebuilding of your project. It compares the date stamps of the source and object files, and those of the object files and the final executables, and re-compiles only the files that have changed. This can not only save time, it also simplifies the task of making sure that your executable is always up to date. For small projects, it's a convenience; for large projects, it's almost mandatory.

How does the make utility know what files make up your project, and what command lines are needed to rebuild them? You provide this information in a text file, which by default is named Makefile. The make utility reads the Makefile, and then checks the date stamps of

each file against the files from which it is made (these are called the file's *dependancies*). If any of the file's dependancies are newer than the file itself (indicating for example that a source file has been updated since the last compile), then the associated command line is executed to bring the file up to date. If a dependancy does not exist, then it is considered to have an extremely old date stamp for the purposes of this comparision.

As an example, say our project is made up of two source files, `hello.c` and `greeting.c`. The build process can be broken down into three steps:

1. `hello.c` is compiled to produce `hello.o`.
2. `greeting.c` is compiled to produce `greeting.o`.
3. `hello.o` and `greeting.o` are linked with the runtime libraries to produce the executable file `hello.exe`.

A makefile representing this project might look like this:

```
hello.exe: hello.o greeting.o
      gcc –o hello.exe hello.o greeting.o

hello.o: hello.c
      gcc –c hello.c

greeting.o: greeting.c
      gcc –c greeting.c
```

In this simple example, make will try to build `hello.exe` since it is the first build target listed in the makefile. `hello.exe` has two dependancies, `hello.o` and `greeting.o`. `hello.o` and `greeting.o` each have one dependancy, which are `hello.c` and `greeting.c` respectively.

If we edit `hello.c`, then it will be saved with a timestamp that is newer than `hello.o`. The next time make is run, it will detect this condition and initiate the command line to re-make all dependants of `hello.c`; so `hello.o` will be compiled, and then `hello.exe` will be linked.

Makefiles may define make variables to replace strings that are used in multiple places. These make variables can simplify the makefile, reduce the chance of errors, and help document the makefile so others may better understand and maintain it. We can simplify the above example by defining the variable `OBJFILES`, producing the modified version below.

```
OBJFILES = hello.o greeting.o

hello.exe: $(OBJFILES)
      gcc –o hello.exe $(OBJFILES)

hello.o: hello.c
      gcc –c hello.c

greeting.o: greeting.c
      gcc –c greeting.c
```

3.22.2 Make command line

Listed in this section are some of Gnu make's more useful and commonly-used options. The full documentation for Gnu make is included on the CD-ROM, and it contains an excellent tutorial on how make works; so we will refer you to that source for the full details.

3.22.2.1 Specifying the target to make

Normally when make runs, it tries to make the first build target listed in the makefile. If you specify one or more targets on the command line, make will instead try to build those targets, in the order listed.

3.22.2.2 Defining variables on the command line

You may define a variable on the command line by using the syntax `name=value`, similar to that used within the makefile. If you need to define a variable which contains embedded spaces, then the definition should be surrounded by double quotes, for example `"OBJFILES=hello.o greeting.o"`.

3.22.2.3 Specifying an alternate name for the makefile (-f <makefile>)

Normally the make utility expects to read its build rules from a file called `Makefile` in the current directory. You can specify an alternate filename with the `–f` option, for `example –f file.mak`.

3.22.2.4 –n: performing a dry run

If you use the `–n` option on the make command line, make will not actually execute the commands to bring the build target up to date; instead, it will only print them on the standard output, so that you can see what actions would be taken in a real build.

3.22.2.5 Forcing a rebuild with –W

You can force make to re-build a file's dependants by specifying it on the command line with the `–W` option. This will cause make to act as if the specified file is very new, and therefore any files which depend upon it will be re-built.

Using our earlier example, if you wanted to re-link the application `hello.exe`, you could invoke make with the command line `make –W hello.o`. Similiarly, if you wanted to force `hello.c` to be recompiled and then linked, the command `make –W hello.c` would achieve this.

3.22.3 Dependancy Files

When an object file depends upon a C or C++ source file, it also depends upon any header files that are included during the compilation. Therefore, when you create a makefile for your project, you must list these header files as dependancies in addition to the actual C source file. This is a major pain, because you have to scan the source files for any `#include` directives and list them all in the makefile; in addition, each time you add or delete an `#include` directive in a source file, you must remember to also update the makefile. Nested include files make it even more difficult to keep the makefile up to date.

Fortunately, Gnu CC implements a simple and effective solution to the problem. When you compile a source file with the Gnu compiler, specifying the option `–MMD` on the compiler command line will cause it to create a text file that contains the dependancies for the source file that was compiled. This file, which has the name of the source file with an extension of `.d`, is in the format required by the make utility. Your makefile can use Gnu make's

`include` directive to include the dependancy file into the main makefile. Since the compiler automatically updates the `.d` file each time the source is compiled, your main makefile effectively is updated at the same time. This can dramatically simplify your job of creating and maintaining a makefile for your project.

When including a makefile fragment, be sure to place a dash in front of the include directive. This tells make not to terminate the build session if the include directive failed, as could happen when the program is being built for the first time and the dependency file does not yet exist. An example might look like this:

```
-include main.d
```

3.22.4 The Makefile template

In the `EXAMPLE` subdirectory, we have provided a makefile template to get you going quickly whenever you start a new project using the XGCC tools. The template implements most of the commonly features of make and gcc while minimizing the amount of typing required to set up a new makefile.

3.22.4.1 Basic setup

When you start a new project, we suggest that you create a new working directory for the project and make a copy of the makefile template there. Then you need to edit the following variables in the makefile:

- `SOURCEFILES` lists the name of all the source files, separated by spaces, which make up the project.
- `PROJECTNAME` defines the name that will be given to the main executable or library file that is built by make. It should not have any filename extension, just the base name of the file; the extension is specified separately.
- `BUILD_FILE_TYPE` defines the filename extension of the main build target. Typically this will be either `.s19` if you want to build an executable program, or `.a` if you are building a library. If the main build target is a hex file with the `.s19` extension, then an executable file (COFF or ELF format, as appropriate) is also built automatically.
- `TARGET_NAME` defines which compiler to use when running gcc. This should beone of the identifiers (either m68k-elf or powerpc-eabi) described in section 3.4.

- TARGET_MACH defines the specific family member to be used, in order to select the correct subset of the family's instruction set.
- TARGET_OPTS specify any other code-generation options needed.
- LDSCRIPT specifies the name of the linker script file to be used when linking the executable file. This can be one of the standard linker script files rom.ld or ram.ld, or you can specify another file name if you have created your own script file for the project.
- If you are using the standard linker script file ram.ld, then the symbols RAM_START and RAM_SIZE should be defined to the starting address and size, respectively, of the RAM memory in your target. If you are using the script file rom.ld, then you will also need to define the symbols ROM_START and ROM_SIZE to specify the start address and size of the ROM memory.

Once these symbols are tailored to the needs of your project, the makefile is ready to go.

3.22.4.2 Additional options

Some additional options are available which implement more features or provide more precise control over how your project is built. These options are set to the most commonly-used settings in the template, and if necessary you can change their settings to get the effect you want.

- LIBDIRS (default value: blank) may be used to define a list of directories which will be searched for library files during the link process.
- SOURCEDIR (default value: blank) allows you to store the source files in one directory while compiling the project in a different location. This can be very handy if you need to build multiple versions of the same program.
- INCLUDEDIRS (default value: blank) may be used to define a list of directories which will be searched for header files during compilation.
- DEBUGGING (default value: -g) sets the debug flag for compiling C and C++ programs. Normally the default value is fine, but you may need to edit it or remove it altogether for special applications.
- OPTIMIZATION (default value: -O2) sets the level of optimization used when compiling C and C++ programs.

- EXTRA_CFLAGS (default value: blank) may be used to specify any additional options when compiling C and C++ programs.
- CREATE_ASM_LISTING: if this variable is set to Y, then a listing file will be created when compiling C/C++ programs or assembly source files. The listing will contain the original high-level language source code interspersed with the generated assembly language code if debugging is enabled.
- EXTRA_ASFLAGS (default value: blank) may be used to specify any additional options when building assembly language files.
- LOADLIBES specifies the name of any library files that should be included when linking the final executable program. If LIBDIRS specifies a list of directories, then these directories will be searched for the named library files.
- LDLIBS: similar to LOADLIBES, but any library files specified here will be placed at the very end of the linker command line, making LDLIBS suitable for library files which are used in multiple projects (whereas LOADLIBES is more useful for libraries specific to this project, since they may also need to access modules contained in the LDLIBS list of files).
- CREATE_MAPFILE: if this variable is set to Y (upper-case), then a map file will be created by the linker showing the names and absolute locations in memory of every module in the executable file.
- EXTRA_LDFLAGS (default blank): use this variable to specify any additional options that may be required when running the linker.

4 Embedded Essentials

4.1 Preprocessor symbols

In addition to those specified by the ANSI language standards, Gnu CC defines several preprocessor symbols during compilation of C, C++, and assembly language files that provide information about the type of compilation being performed and the compiler options in effect during compilation. These symbols may be tested with preprocessor statements such as `#if`, `#ifdef`, `#ifndef` etc in order to control the code that is generated. In this section we present some of the preprocessor definitions that are useful in embedded development.

4.1.1 All targets

Symbol	Comment
_ _embedded_ _	Always defined.
_ _GNUC_ _	When compiling C or C++ code, defined to the compiler's major version number. Not defined when preprocessing assembly-language files.
_ _GNUC_MINOR_ _	When compiling C or C++ code, defined to the compiler's major version number. Not defined when preprocessing assembly-language files.
_ _ASSEMBLER_ _	Defined when preprocessing assembly-language files, not

	defined otherwise.
_ _STRICT_ANSI_ _	Defined when compiling without Gnu extensions.
_ _OPTIMIZE_SIZE_ _	Defined when optimizing for size (-Os).
_ _OPTIMIZE_ _	Defined when any level of optimization, other than –O0, is enabled.
_ _FAST_MATH_ _	Defined when the –ffast-math command line option is passed.

4.1.2 68k

Symbol	Comment
mc68000, _ _ mc68000, _ _ mc68000_ _	Always defined.
mc68302, _ _ mc68302, _ _ mc68302_ _	Defined when `–m68302` command line option passed.
mc68010, _ _mc68010, _ _mc68010_ _	Defined when `–m68010` command line option passed.
mcpu32, _ _mcpu32, _ _mcpu32_ _	Defined when `–mcpu32` or `–m68332` command line option passed.
mc68332, _ _mc68332, _ _mc68332_ _	Defined when `–m68332` command line option passed.
mc68020, _ _mc68020, _ _mc68020_ _	Defined when `–m68020` or `–mc68020` command line option passed (or no `–m` option passed, since `–m68020` is the default.)
mc68030, _ _mc68030,	Defined when `–m68030` or `–mc68030` command line

_ _mc68030_ _	option passed.
mc68040, _ _mc68040, _ _mc68040_ _	Defined when `–m68040` or `–mc68040` command line option passed.
mc68060, _ _mc68060, _ _mc68060_ _	Defined when `–m68060` or `–mc68060` command line option passed.
mcf5200	Defined when `–m5200` command line option passed.
_ _MAX_INT_ _	This symbol will have the value 32767 when the compiler option `-mshort` is in effect, indicating that the `int` data type is 16 bits wide. Any other value indicates that the `int` data type is 32 bits wide.
_ _MRTD_ _	This symbol is defined when the compiler option `–mrtd` is in effect, indicating that the `rtd` instruction is used to return from subroutines.
_ _HAVE_68881_ _	Defined when `–m68881` command line option passed.

4.1.3 PowerPC

Symbol	Comment
PPC	Always defined.
_ _PIC_ _	Defined to the value 1 when the `–fpic` command line option is passed. Defined to the value 2 when the `–fPIC` command line option is passed. Otherwise not defined.
_ _NO_RTTI_ _, _ _no_rtti_ _	Defined to the value 1 when the `–fno-rtti` command line option is passed; otherwise not defined.
_ _NO_EXCEPTIONS_ _, _ _no_exceptions_ _	Defined to the value 1 when the `–fno-exceptions` command line option is passed; otherwise not defined.

_ _EABI_SMALL_DATA_ _	Defined to the value 1 when the `–msdata` or `–msdata=eabi` command line option is passed; otherwise not defined.
_SOFT_FLOAT	Defined when the `–msoft-float` command line option is passed.
_BIG_ENDIAN, _ _BIG_ENDIAN_ _	Defined when compiling in big-endian mode.
_LITTLE_ENDIAN, _ _LITTLE_ENDIAN_ _	Defined when compiling in little-endian mode (`-mlittle`).

4.2 Interfacing C and assembly language functions

4.2.1 68k

This section describes the function call interface used by the gnu compiler on the Motorola M68k family of processors.

4.2.1.1 Calling convention

Functions are called with the JSR (jump to subroutine) instruction. If the function returns a value, it will be in register D0 when the function returns. If the function requires any parameters, they are pushed onto the stack with the rightmost parameter first. All 8- and 16-bit parameters are promoted to integers before being pushed onto the stack; the default size for integers is 32 bits, or 16 bits if the `–mshort` option is passed on the compiler command line.

The compiler maintains a stack frame pointer in register `a6`. The frame pointer is used as a base register to allow access to both function parameters and local variables on the stack using an indexed addressing mode. C functions save the frame pointer on the stack when they are called, and restore it before they return. In addition, they save any processor registers that they modify except for `d0`, `d1`, `a0`, and `a1`; these are considered ‘scratch’ registers which may be used by functions without preserving their contents.

Here's an example of calling a C function from assembly language. Function `abc` has the following prototype:

```
int abc (int a, char *b);
```

To call this function, an assembly language routine would push `b` onto the stack, then `a`. It would then call the function by executing `jsr abc`. Upon returning, register `d0` would contain the return value of the function.

4.2.1.2 Register Usage

Registers D0, D1, A0, and A1 are scratch registers and are not saved and restored when calling other functions. All other registers that are used by a function must be saved on the stack before being modified, and restored from the stack before the function returns.

If the target implements hardware floating point, either internally (such as the 68040) or through a floating-point coprocessor, then FP0 and FP1 are also scratch registers. All other floating-point registers must be saved and restored by the function if used.

4.2.1.3 Stack cleanup

When a function returns control to the function that called it, the stack space consumed by the function's parameters needs to be deallocated (i.e. the stack pointer must be incremented back to its value before the parameters were pushed). There are two ways of doing this, depending upon whether or not the compiler option –mrtd was used to compile the code.

The default strategy (that is, when `-mrtd` is not used) requires the calling function to deallocate the stack space used by function parameters. In our abc example earlier, this can be accomplished with the following instruction (after the JSR ABC):

```
addq.l #8,%sp
```

The alternative approach, when –mrtd is used, requires the called function to clean up the stack; in this case the assembly language function need not do anything, since the C function that it called would have ended with the RTD #n instruction, which removes the parameter space as part of its execution.

The problem for assembly language programmers is, how do you know when you write the code which calling convention is being used? It would certainly be very inconvenient to have to edit every function call if the calling convention were changed. The answer lies in `gcc`'s ability to preprocess assembly language programs before passing them to the assembler. The

68k version of `gcc` defines the preprocessor symbol `__MRTD__` if code is compiled or assembled using the `-mrtd` command-line option. By testing for the presence of this symbol using the `#ifdef` preprocessor construct, you can put code in your program to handle both cases and conditionally assemble the correct version.

In order to preprocess your assembly code, you must name the source file with the '`.S`' extension (i.e. '`program.S`' rather than '`program.s`'). In addition, you must use `gcc` to assemble the program rather than invoking the assembler directly.

Several examples of this type of conditional assembly may be found in `crt0.S`, the source file of the C startup module.

4.2.1.4 16-bit ints

In addition to the RTD calling convention described in the previous section, the other issue of which assembly language programmers need to be aware concerns the size of `int` function parameters. Normally, Gnu CC defaults to 32-bit `int`s, and all function parameters are promoted to `int` size before being pushed on the stack. However, the 68k compiler has a command-line option, `-mshort`, to set the size of `int` variables and function parameters to be 16 bits wide. If this compiler option is in effect, then the amount of stack space allocated by function parameters will be different from the default case. In addition, function parameters on the stack will be located at a different offset from the stack pointer depending on whether or not `int`s are 16 or 32 bits wide.

So how does the assembly language programmer know which case is in effect? This problem is solved in a fashion similar to the RTD calling sequence described earlier. By running assembly language source files through the C preprocessor, you will be able to test the value of the macro `__INT_MAX__`. The `-mshort` compiler option will cause `__INT_MAX__` to have the value 32767. Any other value indicates that `int`s are 32 bits wide. By testing this value using `#if`/`#else`, you can write assembly language programs that have code for both cases and conditionally assemble the correct version.

4.2.2 PowerPC

The PowerPC compiler adheres to the PowerPC EABI (Embedded Application Binary Interface) specification, which among other things defines the calling convention for C functions. We will provide some of the basic information here, and refer you to Motorola's

web site (http://www.mot.com/SPS/ADC/pps/download/8XX/ppceabi.pdf) for a more detailed description.

4.2.2.1 Register usage and Calling convention

Registers r3 through r10 and f2 through f8 are used to pass parameters into a function, in left to right order; these registers are not preserved across function calls. If a function requires more than 32 bytes of integer parameters, or more than 7 floating-point parameters, the remaining parameters are pushed on the stack. Integer function return values are placed in r3 and r4, and floating-point values are returned in f1.

Registers r14 through r30 and f9 through f13 may be used for local variables, and if modified they must be saved on the stack by the function and restored before the function returns.

Register r13 contains a pointer to the symbol `_SDA_BASE_`, which is the base address of the small data sections `.sdata` and `.sbss`. Any variable in either of those two sections may be accessed with a single PowerPC 'load indexed' or 'store indexed' instruction.

The function of register r2 changes depending upon the calling convention used. With the default calling convention, r2 is unused. When small data sections are being used (`-msdata` was passed on the command line), r2 will contain the address of the symbol `_SDA2_BASE_`, which is the base address of the `.sdata2` and `.sbss2` sections. This allows fast access to constant data in a similar fashion to `.sdata`/`.sbss` described earlier.

With position-independent code (PIC), r2 is used in the function prologue to find the global offset table, a table of pointers to all variables and functions used in the program; however the usage differs depending upon which variant of PIC is in use. When 'small' position-independent code is being generated (ie the `-fpic` option was specified), r2 contains the address of the symbol `_GLOBAL_OFFSET_TABLE_`. And when 'large' position-independent code is generated (using `-fPIC`), r2 contains the difference between the data fixup and code fixup offsets.

4.2.2.2 Stack management

Register r1 is used as the stack frame pointer and must always be aligned on an eight-byte boundary. The stack starts in high memory and grows downward. Stack frames are allocated by the called function, rather than the caller. The stack frames form a linked list pointing back toward the first dummy frame created by the startup code; new frames are allocated using the STWU instruction, as in this example:

```
STWU 1,-8(1)
```

4.3 Inline assembly language in C source files

Sometimes it's more convenient to insert a small amount of assembly language code into a C function rather than writing a complete assembly language function from scratch. The Gnu C compiler supports this through the `asm()` operator. Here we provide an introduction to the `asm()` operator; this operator is fully documented in sections 4.31 and 4.32 of the Gnu CC manual.

Gnu CC's `asm()` operator is a very powerful implementation of inline assembly language. In particular, it allows easy access to C expressions (not just variables) from assembly language. In addition, `asm()` statements carry enough information that they can be optimized by the Gnu CC optimizer similar to normal C code.

The parameters for the `asm()` operator are listed in four groups; each group is separated by a colon. The first parameter is the assembly code itself, inside double quote characters. The second group specifies any output values generated by the assembly language code, while the third group defines any input parameters required by the code. Finally, the fourth set of parameters specifies any processor registers altered by the assembly code which were not listed in the output group.

Here's a simple example of a snippet of code that retrieves the status register in a MC68000 processor, and stores it in a C variable:

```
int statusreg;
asm ("move %%sr, %0" : "=r" (statusreg));
```

This `asm()` directive specifies the variable `statusreg` as an output parameter. The parameter consists of a constraint (the text `"=r"`) associated with the name of the variable in parentheses. This particular constraint tells Gnu CC that the output value must be placed into the variable `statusreg` via a general-purpose processor register. If a register is available, the compiler will allocate `statusreg` in a register, substituting its name for our %0 placeholder in the assembly code. Otherwise it will generate extra code to make a register available before inserting our assembly code, and to move the value from the register to `statusreg` and finally restore the register's value after inserting our code.

Note that since `sr` is the name of a CPU register, the gnu assembler requires us to prefix its name with a percent sign. In order to place this percent character into the assembly language output, we must put two percent characters into the `asm()` directive.

As an example of an input parameter, we'll now use `asm()` to write a value to the status register:

```
asm volatile ("move %0,%%sr" : : "r" (statusreg));
```

Since this assembly language code produces no output value visible to the compiler, we leave the output parameter blank. The input parameter specifies the variable `statusreg`, again accessed through a general-purpose register. The compiler will allocate a register and insert the actual register name in place of the `%0` placeholder in the assembly code.

4.3.1 Optimizing assembly language code

What is the `volatile` keyword for in the above example? Because the `asm()` code uses no output parameters, this code is considered by the compiler to have no side effects, and therefore would actually be removed by the optimizer unless specially declared to prevent this. However it very definitely does have side effects (changing the interrupt mask in the status register); they just aren't visible to the compiler. The keyword `volatile` is used to inform the compiler that the assembly code should be not be optimized out of the final program.

Note that the earlier example where we read the value of the status register could also be optimized out of the program if the value that was read was not actually used in a subsequent operation. This is normally desirable if we are only interested in the register's value, but sometimes when controlling peripheral registers the read may be necessary in order to cause other things to happen, for example to clear an interrupt flag. If this is the case, then the `asm()` code that implements the read should also be tagged with the `volatile` keyword to ensure that it is not optimized out of the program.

One other note concerning optimization of assembly code: it's possible that the compiler might re-arrang the the order of independent `asm()` blocks during the optimization process. If your program depends upon the assembly language code being executed in the same order that it is shown in the C source code, then the code should all go into the same `asm()` block.

We have only scratched the surface of this powerful and useful feature; we strongly suggest you review the relevant sections of the Gnu CC manual (one of the HTML documents installed with the XGCC tools) if you plan to make use of it.

4.4 crt0.S/crt0.o

Most C and C++ compiler systems use a small module of assembly language code, called `crt0`, to set up the system before execution starts at `main()`, and Gnu CC is no different. The object module `crt0.o` is automatically included on the linker's command line when `gcc` calls the linker. This module takes on additional responsibilities in systems where there is no operating system underneath the application code. In the EST port of Gnu CC, crt0 is responsible for the following:

- initializing critical peripheral systems after reset
- copying a ROM image of initialized data to RAM
- clearing bss (uninitialized data RAM)
- providing a default exception handler routine for exceptions not handled by user code

4.4.1 Initializing peripherals upon startup

In many embedded systems, there are often hardware peripheral registers that must be set up before the system can start to execute any application code. For example, many embedded microprocessors have chip select hardware to control the RAM and ROM devices on the board, and when the microprocessor comes out of reset these chip selects will often need to be set up before there is any RAM visible to the processor. Since the C compiler generates code that requires the use of the processor stack, a mechanism is required to set up these peripherals before the `main()` function is executed. It would be most convenient if `crt0` allowed the user to somehow specify how this setup is to be done without having to writing a custom `crt0` module for every different hardware platform.

In the EST port of Gnu CC, `crt0` offers two mechanisms to perform peripheral initialization. The first is a system whereby `crt0` can be directed to write user-supplied data to peripheral registers, through the use of initialization records. The second mechanism is through a user-defined function called `hardware_init_hook()`.

4.4.1.1 Initialization records

Initialization records define the address of a peripheral register that needs to have data written to it by crt0 upon power-up. Each record contains a header with the following fields:

- The memory address of the peripheral or variable
- A 32-bit field which encodes the data size (byte, word, or long word) of the data to be written, as well as the number of items to be written

Each record is followed by one or more data items; the exact count is contained in the header's count field. A list of such records may be assembled, marking the end of the list with a null pointer. The symbol `crt0_initialization_list` should be set to point to the first header in the list (this is done automatically by the linker scripts `ram.ld` and `rom.ld`).

As soon as `crt0` gains control, it checks the value of the symbol `crt0_initialization_list`. If it is non-zero, `crt0` starts reading each record and writing the data to the addresses indicated in the record headers, stopping when it reaches the null pointer. It performs these writes without accessing any other memory in the target system (except obviously for the locations specified in the initialization records), so it will run without any RAM being accessible to the processor. This feature makes it ideal for setting up chip selects and other critical peripheral registers in the target system.

It's very easy to define these records in a C program. We have provided a header file, `sys/crt0.h`, which defines some macros to simplify declaration of these initialization records. This mechanism is used in the example I/O drivers in the `EXAMPLE` directory; here's a fragment of the MPC860 driver `860Basic.c`:

```
#include <sys/crt0.h>

#define BaseAddress 0xff000000

/* Set up SIU. */
CRT0_INITLONGS (SIU1, BaseAddress, 3, 0x610000,
0xffffff88, 0xffff0000);
CRT0_INITSINGLEWORD (SIU2, BaseAddress+0xe, 0);
CRT0_INITLONGS (SIU3, BaseAddress+0x10, 4, 0,
0x400000, 0, 0x3c000000);
CRT0_INITLONGS (SIU4, BaseAddress+0x20, 1, 0);
CRT0_INITLONGS (SIU5, BaseAddress+0x30, 1, 0x4001);
```

```
/* Memory controller. */
CRT0_INITLONGS (MEMC1, BaseAddress+0x100, 16,
0xffc00801, 0xfffc0760, 0, 0, 0xc1, 0xffc00800, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0);
CRT0_INITLONGS (MEMC2, BaseAddress+0x164, 2, 0,
0x3f);
CRT0_INITLONGS (MEMC3, BaseAddress+0x170, 4,
0x30001000, 0x2fa20111, 0x800, 0xff0c0027);
```

The following sections document the macros defined in `sys/crt0.h` and how to use them to generate initialization records.

4.4.1.1.1 Byte, Word, and Long initialization records

```
CRT0_INITBYTES(name, where, howmany, stuff...)
CRT0_INITWORDS(name, where, howmany, stuff...)
CRT0_INITLONGS(name, where, howmany, stuff...)
```

These macros create an initialization record to write a string of byte, word, or long-word data to a peripheral. They all have the same form of invocation; only the data size differs between them.

`name` is the name given to the record; it must be unique in the source file containing the macro invocation. `where` is the starting address of the peripheral register(s) to be written. `howmany` defines how many units of data (each consisting of a single byte, word, or long word of data); and `stuff` is the actual data to be written to the peripheral register(s).

4.4.1.1.2 Single-byte and single-word records

```
CRT0_INITSINGLEBYTE(name, where, stuff)
CRT0_INITSINGLEWORD(name, where, stuff)
```

Single-byte and single-word initialization takes advantage of the underlying structure of the initialization record to reduce the overhead in cases where a single, isolated byte- or word-sized unit of data must be written.

`name` is the name given to the record; it must be unique in the source file containing the macro invocation. `where` is the starting address of the peripheral register to be written. `stuff` is the actual data to be written to the peripheral register.

4.4.1.1.3 SPR initialization (PowerPC only)

The PowerPC architecture has a separate address space called the special-purpose registers (SPRs) which are used to control many system-level functions in the processor. These registers are not visible in the processor's memory map; they are only accessible using the `mtspr` and `mfspr` instructions. Since many SPRs are critical to system initialization, the startup code for PowerPC systems provides an extra type of initialization record to facilitate access to the SPR space.

```
CRT0_INITSPR(name, which, value)
```

`name` is the name given to the record; it must be unique in the source file containing the macro invocation. `which` is the number of the SPR register to be written. `value` is the actual data to be written to the SPR register.

When an SPR init record is encountered by the standard crt0 startup code provided by EST, it tests the SPR number for several known values and writes the register using inline code if the SPR number matches. This allows writing SPRs without having any RAM accessible to the processor. If the SPR number is one which is not implemented as an inline write by crt0, then crt0 will construct an `mtspr` opcode in RAM and execute it to perform the write; obviously in this case RAM must be accessible to the processor in order for these SPRs to be writable.

The list of SPR numbers implemented with inline writes is given in table Table 4.1.

Number	Name	Description
638	IMMR	Memory mapping register (Motorola MPC5xx, MPC8xx, MPC82xx)
560, 561, 562	IC_CST, IC_ADR, IC_DAT	Instruction cache, MPC8xx
568, 569, 570	DC_CST, DC_ADR, DC_DAT	Data cache, MPC8xx
784, 786, 787, 789,	MI_CTR, MI_AP, MI_EPN, MI_TWC,	Instruction MMU (MPC8xx)

790, 816, 817, 818	MI_RPN, MI_CAM, MI_RAM0, MI_RAM1	
794, 795, 796, 797, 798, 799, 824, 825, 826	MD_CTR, CAS_ID, MD_AP, MD_EPN, M_TWB, M_TWC, MD_RPN, M_TW, MD_CAM, MD_RAM0, MD_RAM1	Data MMU (MPC8xx).
149	DER	Debug Enable Register (MPC5xx, MPC8xx)

Table 4.1: PowerPC SPRs implemented with inline writes in crt0

4.4.1.1.4 UPM initialization (PowerPC only)

Several of Motorola's MPC8xx devices contain a memory controller called the User Programmable Machine (UPM). These devices require a relatively large, interleaved command/data initialization sequence in order to start operation. In order to simplify the setup of the UPM's, a special initialization record is defined for PowerPC targets.

```
CRT0_INITUPM(name, cmdaddress, cmdvalue,
dataddress, howmany, stuff...)
```

`name` is the name given to the record; it must be unique in the source file containing the macro invocation. `cmdaddress` is the address of the UPM command register. `cmdvalue` is the initial value to be written to the command register; this value will be incremented after each write. `dataaddress` is the address of the data register; each data value is written to this location. `howmany` defines how many long words of data follow the command header; and `stuff` is the actual data to be written to the data register.

Here is an example taken again from `860Basic.c` in the `EXAMPLE` directory:

```
/* UPMB. */
CRT0_INITUPM (UPMB1, BaseAddress+0x168, 0x00800000,
BaseAddress+0x17c, 64,

0x0fffec04, 0x08ffec04, 0x00ffec00, 0x3fffec47,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0x0fffcc24, 0x0fffcc04, 0x08ffcc00, 0x03ffcc4c,
```

```
0x08ffcc00, 0x03ffcc4c, 0x08ffcc00, 0x03ffcc4c,

0x08ffcc00, 0x33ffcc47, 0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0x0fafcc04, 0x08afcc00, 0x3fbfcc47, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,

0x0fafcc04, 0x0cafcc00, 0x01afcc4c, 0x0cafcc00,
0x01afcc4c, 0x0cafcc00, 0x01afcc4c, 0x0cafcc00,
0x31bfcc43, 0xffffffff, 0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,

0xc0ffcc84, 0x01ffcc04, 0x7fffcc86, 0xffffcc05,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff,
0x33ffcc07, 0xffffffff, 0xffffffff, 0xffffffff);
```

4.4.1.2 hardware_init_hook ()

While the initialization record mechanism is extremely useful, sometimes it is necessary to execute code in order to complete system initialization. This is facilitated through a user-defined function called `hardware_init_hook()`.

Immediately after processing the initialization records, `crt0` will check the value of the symbol `hardware_init_hook`. If this symbol is defined (ie non-zero), `crt0` will perform a function call to this location. This function may be defined by the user to perform any hardware-related initialization that may be required.

It is generally not safe to call any C runtime library functions from with `hardware_init_hook()`, since neither the .data nor the .bss sections will have been set up at this point. `hardware_init_hook()` should be restricted to only performing any hardware initialization functions that are required, and then exit. If application-level initialization is required before `main()` is called, there is another mechanism described in section 4.4.2 which is well-suited to this task.

4.4.2 Software initialization before entering main ()

There is one final initialization mechanism in `crt0`, and this one may be useful for your application code as opposed to hardware-related initialization. Just before `main()` is entered, `crt0` will call the function `software_init_hook()`, if it exists. When this function is called, all initialization is complete (ie bss will have been cleared to zero, and initialized variables will have been set with their starting values) and you are free to call any function defined in your application code or in the C runtime library. One warning: C++ constructors will not yet have been called when `software_init_hook()` executes, so be careful not to perform any operations in this function which would use C++ objects.

4.4.3 Default exception handling procedure

We spend a lot of time thinking about the interrupt handlers that we write to support our custom hardware. But what about the vast majority of exception vectors for which there is no handler? It's prudent to define handlers for all exception vectors, to avoid a situation where a spurious noise-triggered interrupt sends the processor off into space.

The EST implementation of `crt0` provides a 'default' exception handler, with the name `__vector_default`, to deal with this situation. In addition, the linker scripts rom.ld and ram.ld use the `PROVIDE` command to force any vector table entries that are not defined in user code to point to this default handler.

The default handler deals with unhandled interrupts and exceptions by re-starting the program from scratch, just as if a hardware reset occurred. However, there is a mechanism by which the application code can detect that the re-start was caused by an exception rather than a reset. `crt0` defines an integer variable called `__unhandled_exception`. If crt0 was entered from a hardware reset, this variable will be zero; but if crt0 was entered due to an unhandled exception, this variable contains the number of the exception vector that caused the restart. Another variable, called `__unhandled_exception_pc`, is set to the value contained in the program counter at the point where the exception occurred.

The header file `sys/crt0.h` can be included into a C program to declare `__unhandled_exception` for access by C programs.

4.4.4 crt0 entry points

The crt0 startup module provides three entry points: `_start`, `_start2`, and `_restart`. Each of these entry points is described in the following sections. These functions are prototyped in the header file `sys/crt0.h`.

4.4.4.1 _start

`_start` is the 'cold-start' entry point, used to bring the system up from a system reset. When your program is placed in ROM, the reset vector will be set (in vectors.o) to transfer control to this location. `_start` performs all initialization steps, in the following order:

- Interrupts are disabled and some critical processor registers (for example the Machine State Register in the PowerPC, or the Status Register in the M68k/ColdFire architecture) are initialized.
- The system's critical peripherals are initialized, as dictated by the user-defined initialization records
- The stack pointer is set to the value of the symbol `__stack` (if defined)
- The function `hardware_init_hook()` is called, if defined
- The `.bss` section is cleared to all zeros, unless the symbol `crt0_flags` is defined and bit 1 of `crt0_flags` is non-zero
- If the program was linked as a ROM-resident executable (using the linker script `rom.ld`), a ROM-resident image of the `.data` section is copied to RAM to set up all initialized variables.
- The variables `__unhandled_exception` and `__unhandled_exception_pc` are initialized to zero, to indicate to the application program that it was entered as a result of a system reset.
- If the program was compiled and linked using position-independent code (PIC), then the processor's base pointer to the global offset table is initialized. The code fixup is calculated based upon the program's offset from its linked address, and a data fixup value of zero is assumed.
- An initial dummy stack frame is set up, and any final processor-specific initialization is done.

- If defined, the function `software_init_hook()` is called.
- The application's `main()` function is called, and if it returns the return value is passed to the `exit()` function.

4.4.4.2 _start2

_start2 is also a 'cold-start' entry point, however it is intended for use when the program is being hosted within another system environment, for example a ROM monitor program. The `_start2` entry point has several function call parameters which provide the host environment with the opportunity to control some of the startup parameters. It is prototyped as follows:

```
void _start2 (int argc, char *argv [], void
*RamStart, LONG RamSize) __attribute__((noreturn));
```

As shown in the prototype above, `_start2` allows the host environment to pass command line parameters to the program in addition, the host environment may specify the address of the program's RAM buffer to be used for data, bss, heap, and stack memory. (The program must be compiled with the `-fpic` or `-fPIC` option in order to use this feature. If the `RamStart` parameter (the address of the RAM block) is set to zero, then crt0 will use the default RAM addresses as specified at link time.

When control transfers to _start2, the same sequence of steps is performed as for _start, except for the following:

- The critical CPU registers (for example, MSR on PowerPC) are not altered upon entry
- For programs compiled as position-independent code, the global offset table will be initialized to use the data fixup offset as calculated based upon the `RamStart` parameter passed in the function call. The code fixup offset is calculated in the same manner as for _start.

4.4.4.3 _restart

The symbol `_restart` is a 'warm-start' entry point. It is prototyped as follows:

```
void _restart (LONG new_unhandled_exception, void *
new_unhandled_exception_pc, int UseDataFixup)
__attribute__((noreturn));
```

`_restart` is used by the default exception handler to restart the system when an unhandled exception occurs. It may also be called by user-defined exception handlers if a restart is desired.

`__restart` performs all the same initialization as `_restart`, with the exception that the variables `__unhandled_exception` and `__unhandled_exception_pc` are set to the vector number and execution address, respectively, that caused re-entry into crt0. These variables may be inspected by the application program to determine why the program was re-started, and take appropriate action if possible.

4.5 Exception handlers

It's easy to write your own interrupt and exception handlers in C. Each entry in the processor's exception vector table is assigned a unique name. To define a handler routine for that exception, simply define a C function with one of the reserved names; the code in `vectors.S` / `vectors.o` ensures that function will be called when the exception occurs. Any exceptions for which you do not provide a handler will be vectored by the linker script to call the default handler function `__vector_default`, located in the crt0 startup module. Section 4.4.3 provides more detail on the exact operation of this function.

There are some specific requirements that each processor family places on exception handlers; these are detailed in the following subsections.

4.5.1 M68K

M68k exception handlers are addressed directly in the processor's exception vector table, which is defined in the file `vectors.o` (this file is supplied as part of the XGCC installation). In order to generate the correct function entry/exit sequence, exception handlers written in C must be declared using a special compiler directive - `__attribute__((interrupt))`. The example fragment below is taken from `332-io.c` in the `EXAMPLE` subdirectory.

```
void __vector_40 (void) __attribute__
((interrupt));

void __vector_40 (void)
{
.
.
}
```

As shown in the example, you must first declare the function with the interrupt attribute. You can then write the function just like any other C function, except that the compiler will save and restore the necessary registers in the function prologue and epilogue, and end the function with the `RTE` (return from exception) instruction rather than `RTS` (return from subroutine). Exception handlers must always be declared as shown above, i.e. with no parameters and returning `void`.

Most of the reserved names for the user-defined m68k exception handlers are in the form `__vector<number>`, where `<number>` represents the vector number as two hexadecimal characters. For example, `__vector_40` is the first user vector, number 64 in decimal. Most of the Motorola-defined exceptions have the Motorola-defined name in place of the number; a list is given in Table 4.2.

Vector number	**Description**	**Function name**
2	Access fault (bus error)	`__vector_access_fault`
3	Address error	`__vector_address_error`
4	Illegal opcode	`__vector_illegal_instruction`
5	Divide by zero	`__vector_divbyzero`
6	CHK, CHK2 instruction	`__vector_chk`
7	TRAPcc, TRAPV, FTRAPcc instructions	`__vector_trapcc`
8	Privilege violation	`__vector_priviledge`
9	Trace	`__vector_trace`

10	Line 1010 emulation	`__vector_lineA`
11	Line 1111 emulation	`__vector_lineF`
13	Coprocessor protocol violation	`__vector_CPprotocol`
14	Format error	`__vector_format`
15	Uninitialized interrupt	`__vector_uninitialized`
24	Spurious interrupt	`__vector_spurious`
25	Autovector level 1	`__vector_auto1`
26	Autovector level 2	`__vector_auto2`
27	Autovector level 3	`__vector_auto3`
28	Autovector level 4	`__vector_auto4`
29	Autovector level 5	`__vector_auto5`
30	Autovector level 6	`__vector_auto6`
31	Autovector level 7	`__vector_auto7`
32	TRAP #0	`__vector_trap0`
33	TRAP #1	`__vector_trap1`
34	TRAP #2	`__vector_trap2`
35	TRAP #3	`__vector_trap3`
36	TRAP #4	`__vector_trap4`
37	TRAP #5	`__vector_trap5`
38	TRAP #6	`__vector_trap6`
39	TRAP #7	`__vector_trap7`
40	TRAP #8	`__vector_trap8`
41	TRAP #9	`__vector_trap9`
42	TRAP #10	`__vector_trapA`

43	TRAP #11	`__vector_trapB`
44	TRAP #12	`__vector_trapC`
45	TRAP #13	`__vector_trapD`
46	TRAP #14	`__vector_trapE`
47	TRAP #15	`__vector_trapF`
48	FP branch	`__vector_Fpbranch`
49	FP inexact result	`__vector_FPinexact`
50	FP divide by zero	`__vector_FPdivbyzero`
51	FP underflow	`__vector_FPunderflow`
52	FP operand error	`__vector_FPoperand`
53	FP overflow	`__vector_FPoverflow`
54	FP signalling NAN	`__vector_FPnan`
55	FP unimplemented data type	`__vector_FPunimplemented`
56	MMU configuration error	`__vector_MMUconfig`
57	MMU illegal operation	`__vector_MMUillegal`
58	MMU access level violation	`__vector_MMUaccess`

Table 4.2: exception vector function names for the 68k

4.5.2 PowerPC

Power PC exception handlers should be defined as shown in the example below:

```
void __vector_externalinterrupt (LONG
new_unhandled_exception, void *
new_unhandled_exception_pc);
```

`new_unhandled_exception` contains the exception vector number which caused entry into the function. `new_unhandled_exception_pc` is the address of the opcode that was executing when the interrupt occurred. There is no special compiler directive needed to

declare exception handlers for PowerPC; the file `vectors.o` contains the prologue code for each vector that restores the machine to a safe state after the exception and then calls the appropriate handler function.

Reserved names for the Power PC exception handler functions are listed in Table 4.3. These functions are all declared in the header file `sys/ppc-exceptn.h`.

Vector offset	**Description**	**Function name**
0x200	Machine check	`__vector_machinecheck`
0x300	Data access	`__vector_dataaccess`
0x400	Instruction access	`__vector_instructionaccess`
0x500	External interrupt	`__vector_externalinterrupt`
0x600	Alignment	`__vector_alignment`
0x700	Program exception	`__vector_program`
0x800	Floating-point unavailable	`__vector_fpunavailable`
0x900	Decrementer	`__vector_decrementer`
0xa00	Reserved	`__vector_reserved1`
0xb00	Reserved	`__vector_reserved2`
0xc00	System call	`__vector_systemcall`
0xd00	Trace	`__vector_trace`
0xe00	Floating-point assist	`__vector_fpassist`
0xf00	Not assigned	`__vector_0f00`
0x1000	Software emulation	`__vector_swemulation`
0x1100	Instruction TLB miss	`__vector_instructiontlbmiss`
0x1200	Data TLB miss	`__vector_datatlbmiss`
0x1300	Instruction TLB error	`__vector_instructiontlberror`

0x1400	Data TLB error	`__vector_datatlberror`
0x1500	Unassigned	`__vector_1500`
0x1600	Unassigned	`__vector_1600`
0x1700	Unassigned	`__vector_1700`
0x1800	Unassigned	`__vector_1800`
0x1900	Unassigned	`__vector_1900`
0x1a00	Unassigned	`__vector_1a00`
0x1b00	Unassigned	`__vector_1b00`
0x1c00	Data breakpoint	`__vector_databreakpoint`
0x1d00	Instruction breakpoint	`__vector_instructionbreakpoint`
0x1e00	Maskable external breakpoint	`__vector_maskablebreakpoint`
0x1f00	Non-maskable external breakpoint	`__vector_nonmaskablebreakpoint`

Table 4.3: exception vector function names for the PowerPC

4.6 Position-Independent Code (PIC)

By default, the code generated by the Gnu compiler is position-dependant; the code uses absolute addresses when making reference to functions and variables, and therefore the program will only operate correctly when loaded at its link address. However, the Gnu tools also include support for generating Position-Independent Code (PIC) from your C and C++ programs.

4.6.1 PIC overview

Programs compiled as Position-Independent Code do not make use of any absolute addresses; rather, when a variable or function address is needed in the program, it is loaded from a table

of pointers, called the Global Offset Table (GOT), which is constructed automatically by the compiler.

The GOT contains the address of each variable and function as calculated at link time by the linker. The addresses in the GOT are modified, or 'fixed up', at runtime by the startup code, which calculates the difference between the program's link address and the address at which is actually executing, and adds that difference (called the 'fixup' value) to each pointer in the GOT before the compiler-generated code runs.

4.6.2 -fpic ('little' PIC) vs. –fPIC ('big' PIC)

As described in section 3.8, Position-Independent Code is enabled with the command line options `-fpic` and `-fPIC`. `-fpic` ('p', 'i'. And 'c' all lower-case) enables 'small' PIC, which typically results in faster and smaller code than the `-fPIC` ('P', 'I'. And 'C' all upper-case) option; it does this by limiting the size of the GOT to 64K bytes, or 16K entries, so any entry may be accessed with a single-word offset. Each function or variable in the entire program has exactly one entry in the GOT.

On PowerPC, register R2 is used as a base address to the GOT, and GOT entries are accessed using the indexed load instructions. On 68k and ColdFire, address register A5 contains the base address of the GOT, and the indexed addressing mode with 16-bit offset is used to access its contents. In both cases, the 'base address' actually refers to the location midway between the start and end of the GOT, to allow both positive and negative offsets for a total span of 64K.

In contrast, the `-fPIC` option ('P', 'I'. And 'C' all upper-case) enables 'big' PIC. In this mode, the size of the GOT is unlimited; however, you pay a penalty in the form of slightly larger and slower code. In the 68k and ColdFire families, the indexed addressing mode is used, with 32-bit offsets, to load values from the GOT; this is slightly larger and slower than the 16-bit offset. In addition, only CPU32 and up (68020, 030, 040 etc) support the 32-bit offset form of the indexed addressing mode, so 'big' PIC is not available on the original MC68000 and its derivatives.

In PowerPC, 'big' GOT causes each function to allocate a 'private' area in the GOT, and the function prologue calculates a base pointer to that area in a CPU register when the function is called. Every function or variable referenced in the function causes an entry to be created in that 'private' area of the GOT, so there will be multiple GOT entries for any variable that is referenced in multiple functions. Since the PowerPC limits index values to 16 bits, this mode allows up to 16K GOT entries *per function*, as opposed to 16K entries for the entire program

in ‘little’ PIC. The function prologue is larger by a few instructions, since the compiler must calculate a pointer to the ‘private’ area in the GOT; however, after the additional overhead of the function prologue, variable accesses incur no additional penalty compared to ‘little’ PIC.

4.6.3 Code and data fixups

As mentioned in section 4.6.1, the reason that code generated with the –fpic or –fPIC options is position-independent is that the addresses in the Global Offset Table are ‘fixed up’ by the program’s startup code so that they contain the correct run-time addresses. In the standard Gnu distribution, the same fixup value is applied to all pointers in the GOT; this implies that code and data must be relocated together as a unit, rather than being able to move code and data separately. Practically speaking, it also implies that the program must be executed from RAM, since it’s very unlikely that the exact same fixup value could be successfully applied to both a text section resident in flash memory and the data and bss addresses in RAM.

The EST distribution has some enhancements to the compiler and startup code that make PIC more suitable for embedded applications. Two fixup values are used, one for code pointers and the other for data pointers. Any pointer containing an address that falls between the start and end of the text section has the code fixup applied, while pointers with other values have the data fixup applied. Null pointers are not fixed up. This allows data and code to be relocated separately at runtime; for example, the code could be executed at its link address in flash memory, but data moved to a new location in RAM.

The fixup values are calculated in the startup module `crt0.o`. It takes the address at which it is executing and subtracts from that the address at which it was linked to run; this difference is the code fixup. Since there is no automatic way to deduce what value to use for the data fixup, it is calculated based upon the `RamStart` parameter passed in the call made into crt0’s `_start2` entry point. If this parameter is zero, then the data fixup value is zero and the linked addresses are used for RAM variables. Otherwise, the linked address of the starting address of RAM is subtracted from the `RamStart` parameter to arrive at the data fixup value. If crt0 is entered through the ‘cold start’ entry point `_start`, `RamStart` (and therefore the data fixup value) is assumed to be zero.

4.6.4 Unhandled exceptions and the data fixup value

If an unhandled exception results in control passing to crt0 to cause a warm start, then crt0 contains code which will re-calculate the data fixup value which it requires in order to re-initialize the system and re-enter `main()`.Recall that there is no automatic way to deduce

what this value should be; it is an arbitrary value imposed by the external operating environment. In a restart after an exception, it is possible to calculate the data fixup value based upon the value of one of the critical CPU registers (R13 for PowerPC, A5 for 68k/ColdFire).

However, depending upon the requirements of your application and the type of exception encountered, it may be possible that one or more CPU registers might have been corrupted and may no longer contain the correct base address. For example, suppose that the processor was subjected to an EMI pulse that caused it to jump out of its normal code section and start executing some data locations as opcodes. Eventually an illegal opcode trap, bus error, or watchdog timer would cause an exception to occur which resulted in a restart through `__vector_default`. Having possibly run through dozens of unknown opcodes before entering the exception handler, the register's contents quite easily could have been corrupted before the exception occurred.

If this type of catastrophic error is possible in your application, the default method used by crt0 to calculate the data fixup value will not be reliable. You must either not use data fixup (ie you must not relocate RAM addresses at runtime), or you must provide an alternate method for obtaining the RAM start address and have your exception handler call _restart directly

4.7 Omitting exception and RTTI support from C++ programs

The Gnu C++ compiler in XGCC includes full support for C++ exceptions and Run-Time Type Information (RTTI). By default, any time you have at least one C++ module in your program, the linker will include the run-time support code required to implement these features. The total overhead of these run-time modules is about 16K bytes of code and 2K bytes of data (on PowerPC).

For larger programs, this overhead does not represent a problem, and many developers will elect to leave the exception and RTTI support in their programs. However, this may represent significant overhead for smaller projects; in addition, some developers may choose not to use these features. For this reason, the XGCC system is configured to allow you to compile and link programs without these features.

To remove exceptions and RTTI, compile all modules in the program with the command-line options `–fno-exceptions –fno-rtti`. In addition, these options should be passed to

the linker so that it will select the run-time libraries that were compiled with these same options. These options should always be used together; omitting one or the other will cause errors at link time.

4.8 Runtime libraries

4.8.1 libgcc.a

libgcc is a library of support routines that are needed by the compiler to perform operations that are too large to be efficiently open-coded; that is, operations that are used frequently and cannot be implemented in a short instruction sequence. In this case, the compiler inserts a call to a support subroutine rather than inserting the instruction sequence over and over again in the compiled code. This results in smaller executables with little or no impact on performance.

Examples of this type of operation might include software floating-point math support, integer multiply and divide (on some targets), memory-to-memory moves, saving and restoring registers on function entry and exit, and functions for calling C++ constructors and destructors.

The CD-ROM installs pre-built copies of libgcc.a for each target configuration, and it's very unlikely that you would ever have to change this library unless you are porting the compiler to a new processor architecture.

4.8.2 The newlib runtime library

As described in section 1.4.3, newlib is a complete implementation of the standard C runtime library suitable for embedded applications. It takes relatively small amounts of memory to support the library functions, and applications built with newlib may be distributed without royalties or disclosure of library source code.

4.8.2.1 Functions defined

Full documentation on these functions is available in the newlib reference manual. Following is a summary of the library functions implemented by newlib:

- From stdlib.h: abort, abs, assert, atexit, atof/atoff, atoi, atol, bsearch, calloc, div, ecvt/scvtf/fcvt/fcvtf, gcvt/gcvtf, ecvtbuf, fcvtbuf, exit, genenv, labs, ldiv, malloc/realloc/free, mbtowc, qsort, rand/srand, strtod/strtodf, strtol, strtoul, system, wctomb
- From math.h: acos/acosf, acosh/acoshf, asin/asinf, asinh/asinhf, atan/atanf, atan2/atan2f, atanh/atanhf, jN/jNf/yN/yNf, chrt/chrtf, copysign/copysignf, cosh/coshf, erf/erff/erfc/erfcf, exp/expf, expm1/expm1f, fabs/fabsf, floor/floorf/ceil/ceilf, fmod/fmodf, frexp/frexpf, gamma/gammaf/lgamma/lgammaf, hypot/hypotf, ilogb/ilogbf, infinity/infinityf, isnan/isnanf/isinf/isinff/finite/finitef, ldexp/ldexpf, log/logf, log10/log10f, log1p/log1pf, matherr, modf/modff, nan/nanf, nextafter/nextafterf, pof/powf, rint/rintf/remainder/remainderf, scalbn/scalbnf, sqrt/sqrtf, sin/sinf/cos/cosf, sinh/sinhf, tan/tanf, tanh/tanhf
- From ctype.h: isalnum, isalpha, isascii, iscntrl, isdigit, islower, isprint, isgraph, ispunct, isspace, isupper, isxdigit, toascii, tolower, toupper
- From stdio.h: clearerr, fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, fiprintf, fopen, fdopen, fputc, fputs, fread, freopen, fseek, fsetpos, ftell, fwrite, getc, getchar, iprintf, mktemp/mkstemp, perror, putc, putchar, puts, remove, rename, rewind, setbuf, setvbuf, siprintf, printf/fprintf/sprintf, scanf/fscanf/sscanf, tmpfile, tmpnam/tempnam, vprintf/vfprintf/vsprintf
- From string.h: bcmp, bcopy, bzero, index, memchr, memcmp, memcpy, memmove, memset, rindex, strcat, strchr, strcmp, strcoll, strcpy, strcspn, strerror, strlen, strlwr, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok, strupr, strxfrm
- From signal.h: raise, signal

- From `time.h: asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, time`
- From `locale.h: setlocale, localeconv`

In addition, the macros `va_start`, `va_arg`, and `va_end` are defined to support functions with variable argument lists. Both the `stdarg.h` and `vararg.h`, header files are included to support both K&R and ANSI-compliant code.

4.8.2.2 Integer-only printf()

In order to help conserve memory space, the `printf()`functions are implemented in both integer-only and floating-point-capable versions. The integer-only versions are about half the size of the standard versions, so you will save space there; in addition, if there is no other floating-point math in your program, using `iprintf()` instead of `printf()` will prevent the floating-point support library routines from being linked into your program. This can also save significant code space, particularly on targets that do not implement floating-point math in hardware.

To use the integer-only versions, simply substitute `iprintf` wherever `printf` would normally be used. This applies to all functions in the `printf()` family, ie: `iprintf()`, `fiprintf()`, `siprintf()`, `vfiprintf()`, etc.

4.8.2.3 float versions of math functions

Each math function defined in `math.h` is defined to operate upon and return double-precision values. newlib also makes available equivalent functions which operate upon and return single-precision values. The advantage to using the single-precision versions is that if you don’t need the extra precision of the double type, you will generally get better performance with the single-precision version, especially on lower-cost processors that implement floating-point math in software rather than hardware.

To use the single-precision functions, simply append the letter ‘f’ to the function name. For example, `tanh()` is implemented using doubles, whereas `tanhf()` is implemented with floats.

4.8.2.4 Using newlib

Newlib is pre-compiled by EST and installed on your hard drive with the compiler tools. Multiple versions are installed, one for each major family member and/or compiler option setting. All you have to do is make sure that `-lc` and `-lm` are referenced at some point on the linker command line. If you use EST's `rom.ld` and `ram.ld` linker scripts, this is done for you in the linker script.

4.8.3 Support functions required by newlib

A total of 18 supporting functions are required in order to implement the newlib library on a new system. These functions are listed in the table below. newlib is fully capable of implementing support for sophisticated OS features such as file I/O and multitasking. If your development project does not include these features however, then many of the support functions may be stubbed out (ie perform no operation and return a default value).

Generic versions of many of these functions have already been implemented for you in the librom library included with the XGCC CDROM. If they will meet your needs (as they will for many systems), then there is very little work needed to get newlib running on your custom hardware. Please refer to section 4.10 for more detail on EST's librom I/O subsystem.

Function/variable	Description
`void _exit (int result)`	Return control to host system; called by `exit()` and `system()`
`int close (int fd)`	Closes the file identified by the file descriptor `fd`
`char **environ`	Pointer to environment strings
`int execve(char *name, char **argv, char **env)`	Transfers control to a new process
`int fork (void)`	Creates a new process
`int fstat(int file, struct stat *st)`	Returns status of an open file.
`int getpid (void)`	Returns process ID of currently executing process
`int isatty (int fd)`	Returns non-zero if the file indexed by `fd` is a terminal device

`int kill (int pid, int sig)`	Signal a process
`int link (char *old, char *new)`	Rename a file
`int lseek (int fd, int ptr, int dir)`	Sets the position of a file.
`int read (int fd, char *ptr, int len)`	Read `len` characters from the file indexed by `fd`
`caddr_t sbrk (int incr)`	Increase program data space
`int stat (char *file, struct stat *st)`	Returns status of file (by name)
`int times (struct tms *tbuff)`	Returns timing information for the current process.
`int unlink (char *filename)`	Delete a file.
`int wait (int *status)`	Wait for a child process.
`int write (int file, char *ptr, int len)`	Write `len` characters to the file indexed by `fd`.

Table 4.4: support functions required by newlib

4.9 Linking the correct libraries ('multilib')

Many embedded microprocessor architectures implement multiple versions of their instruction sets; some high-end family members may (for example) implement hardware floating-point math, which other less sophisticated members rely on software emulation. In addition, many processor architectures offer a choice of multiple calling conventions, each with its own advantages and disadvantages, in order to optimize performance for particular applications. Typically, when you select a particular instruction set or calling convention to use for your application code, you must insure that all runtime libraries that you link with

your application code were also compiled with the same set of compiler options; otherwise it's very likely that your program will not work.

This seemingly obvious and simple issue can become something of a nuisance when you need to support multiple projects that were built with different compiler options. If you had installed only a single copy of the library, you would be forced to repetitively re-build it from source code each time you wanted to link with a project built with a different set of compile options. The XGCC compiler tools resolve this problem by installing multiple sets of the runtime libraries, with each set built with a different combination of command-line options; then, the compiler command line options given at link time are used to select the appropriate libraries with those same options, by passing the appropriate directory name to the linker.

There is one catch to the scheme: it only works if you link your program using the gnu driver program, gcc. If you invoke the linker directly (for example, using `m68k-elf-ld` or `powerpc-eabi-ld` on the command line rather than `gcc`) then you are responsible for specifying the exact location of all the object files and libraries that the linker should include use to build the executable. Unless you have a particular requirement to control every option passed to the linker, it's advisable to always use gcc to link your programs, because it takes care of many details for you in the link process. If you want to see the exact command line that gcc uses to invoke the linker, then add the option `-v` on the gcc command line and it will display the command line on the console when it links.

The libraries are stored in two sets of subdirectories in the compiler directory tree. Libraries relating to the target environment (eg: `libc.a`, the startup code `crt0.o`, etc) reside in `\xgcc32\<targetname>\lib`. Compiler-specific support libraries (for example: `libgcc.a`, `libstdc++.a`, etc) are stored in `\xgcc32\lib\gcc-lib\<targetname>\<compiler version>`. `<targetname>` is the 'configure' name for the target architecture (eg: `m68k-elf` for 68k, `powerpc-eabi` for embedded PowerPC, etc) and `<compiler version>` is the numeric version number of the compiler release (e.g.: 2.95.2).

The name of each subdirectory roughly corresponds to the names of the command line options with which they are compiled. Table 4.5 lists the directory name for each combination of processor core and build option in the 68k family.

	MC68000 `[-m68000]`	CPU32 `[-mcpu32]`	MC68020, MC68030, MC68040	MC68060 `[-m68060]`	Coldfire `[-m5200]`

			[none]		
Default	m68000	mcpu32	.	m68060	m5200
-mshort	mshort\m68000	mshort\mcpu32	mshort		
-mrtd		mrtd\mcpu32	mrtd		
-mshort –mrtd		mshort\mrtd\mcpu32	mshort\mrtd		
-msoft –float			msoft-float		
-msoft–float –mshort			msoft-float\mshort		
-msoft–float –mrtd			msoft-float\mrtd		
-msoft–float –mshort –mrtd			msoft-float\mshort\mrtd		

Table 4.5: multilib options and directory locations for 68k targets

For PowerPC, Table 4.6 lists the build options and directory names for the runtime libraries.

Calling Convention	Big-endian	Little-endian [-mlittle]	Big-endian, software floating-point [-msoft-float]	Little-endian, software floating-point [-mlittle –msoft-float]
Default	.	le	nof	nof/le
-mcall-aix	ca	le\ca	nof\ca	nof\le\ca
-msdata	msdata	msdata\le	nof\msdata	nof\le\msdata
-fpic	pic	le\pic	nof\pic	nof\le\pic
-fPIC	picREL	le\picREL	nof\picREL	nof\le\picREL
-fno-exceptions, -fno-rtti	nortti\noexcp	le\nortti \noexcp	nof\nortti \noexcp	nof\le\nortti\ noexcp
-mcall-aix, -fno-exceptions, -fno-rtti	ca\nortti \noexcp	le\ca\nortti \noexcp	nof\ca\nortti \noexcp	nof\le\ca\nort ti\noexcp

-msdata, -fno-exceptions, -fno-rtti	msdata\nortti\noexcp	le\msdata\nortti\noexcp	nof\msdata\nortti\noexcp	nof\le\msdata\nortti\noexcp
-fpic, -fno-exceptions, -fno-rtti	pic\nortti\noexcp	le\pic\nortti\noexcp	nof\pic\nortti\noexcp	nof\le\pic\nortti\noexcp
-fPIC, -fno-exceptions, -fno-rtti	picREL\nortti\noexcp	le\picREL\nortti\noexcp	nof\picREL\nortti\noexcp	nof\le\picREL\nortti\noexcp

Table 4.6: multilib options and directory locations for PowerPC

4.10 Customizing the link process: XGCC's "Modular Linking"

When an executable program is linked, the following components are included in the link process (listed in order of their appearance on the linker's command line):

1. The 'start file', if any, required by the compiler's support code. For PowerPC this file is named `ecrti.o`.
2. The program's startup module, `crt0.o`
3. The object modules which make up the program (specified by the programmer on the `gcc` command line)
4. (If linking with `g++.exe`) the C++ runtime library `libstdc++.a` and the floating-point math library `libm.a`.
5. The compiler support library `libgcc.a`.
6. The exception vector table module `vectors.o`.
7. The runtime libraries `librom.a` and `libc.a`.
8. The compiler support library `libgcc.a` is linked again to resolve any remaining compiler support routines.
9. The 'end file', if any, required by the compiler's support code. For PowerPC this file is named `ecrtn.o`.

Of the components listed above, provision is made that nos. 2, 6, and 7 may be easily replaced or eliminated by the developer. We refer to this as XGCC's "modular linking" process, since these portions of the link process may be altered without affecting the other

(required) steps. The manner in which this is done differs slightly for each module, depending upon its role in the process.

4.10.1 Replacing the startup module crt0.o (link step 2)

The inclusion of the startup module `crt0.o` is controlled by an `INPUT` directive in the linker script file `startup.ld`, located in the compiler directory tree. Since this directive causes the linker to search for an object file in its list of library directories, in the default case the linker will find EST's standard crt0.o module when it searches the compiler library directories. However, the `SEARCH_DIR` directive in `startup.ld` causes the linker to add the project's working directory to this search list; this also means that you can substitute your own `crt0.o` module in place of the standard one simply by placing it in the project's working directory, since this directory will be searched first.

4.10.2 Replacing or eliminating the exception vector table module (link step 6)

The exception table module, `vectors.o`, is necessary for applications that run on the 'bare metal' and assume control of the processor's exception vector table. In some applications however, it may be necessary to customize this module for special requirements. Still other applications may run under control of a host software environment such as a real-time OS or ROM monitor, which controls the processor's exception vector table itself; and in these cases it may be desirable to remove `vectors.o` completely from the link process.

The inclusion of the exception vector table module is controlled by a linker script named `vectors.ld`, located in the compiler's library directory tree. To substitute a custom exception vector module, use the same trick as described for the startup module (see section 4.10.1): simply provide your own object module named `vectors.o` in the project's working directory, and the linker will find this module first as it searches its list of library directories. To eliminate the vector table module altogether from the link, simply create an empty file named `vectors.ld` in your project's working directory; again, this file will be found first when the linker searches for it, and since it is empty the vector table module will not be linked in. other special requirements may also be accommodated by this technique.

4.10.3 Modifying or eliminating run-time libraries (link step 7)

By default, EST's I/O integration library `librom.a` (described in section 4.11) and the newlib standard C runtime library `libc.a` are included in the link process. While this will

be appropriate for many applications, there will inevitably be occasions when one or both of these libraries will need to be replaced or excluded from the link process.

The libraries are linked under control of the linker script `libs.ld`, located in the compiler's library directory tree. To modify the set of libraries used in this link step, simply create your own file named `libs.ld` in your project's working directory and specify the libraries you need; since your working directory always appears first in the linker's list of search directories, this file will override the standard one in the compiler directory tree.

4.11 EST's librom.a I/O subsystem

It's sometimes necessary to interface to newlib at the level described in section 4.8.3. However, for many embedded products, it's possible to define a much simpler interface that dramatically reduces the amount of work involved to write the hardware-specific driver code. This is the role fulfilled by librom.

librom implement the following features:

- Two versions of I/O driver: buffered (interrupt-driven) and non-buffered (polled I/O)
- Any number of named devices may be defined
- Devices may be opened by calling `fopen()` with user-defined device name
- Input routines support backspace processing for line editing on character entry
- Character echo on input
- Input translation of carriage returns to newlines
- Output drivers support translation of newlines to CR/LF sequence
- All editing and translation features may be enabled/disabled at run time through simple I/O control function call
- User-defined 'idle' function may be called when waiting for input characters or output buffer space

The following additional features are implemented in the buffered (interrupt-driven) implementation of the library:

- Function calls available to get number of characters waiting in input and output buffers
- User-defined function can be called by the interrupt service routine upon receipt of characters, or when the output buffer is emptied

Despite this wealth of features, it is very simple to implement this library on your custom hardware. The MC68332 driver in the `EXAMPLE` subdirectory implements interrupt-driven serial I/O on the SCI serial port with only 26 lines of C code in two functions.

A pre-built linker archive named `librom.a` is installed with the XGCC tools; it contains all the functions that provide the interface between newlib and your hardware driver routines. You must write and link an object file containing the hardware-specific driver functions that are called by the routines in `librom.a` and by newlib.

4.11.1 librom implementation of newlib support functions

Following is a summary description of how librom implements each support routine required by newlib.

Function/variable	`librom` implementation
`void _exit (int result)`	Typically implemented as an endless loop, or a debugger ‘trap’ instruction, since there is no host operating system to which we can exit.
`int close (int fd)`	Implemented.
`char **environ`	Null pointer

`int execve(char *name, char **argv, char **env)`	`errno=ENOMEM;`
	`return -1;`
	`librom` only implements one ‘process’: the application program.

`int fork (void)`	`errno=EAGAIN;`
	`return -1;`
	`librom` only implements one ‘process’: the application program.

`int fstat(int file,`	`errno = EIO;`

`struct stat *st)`	`return 0;` `librom` does not implement a file system, so this function is not supported.
`int getpid (void)`	`return 1;` `librom` only implements one ‘process’: the application program.
`int isatty (int fd)`	`return 1;` As far as `librom` is concerned, everything is a terminal device.

`int kill (int pid, int sig)`	`errno = EINVAL;`
	`return –1;`
	`librom` only implements one ‘process’: the application program.

`int link (char *old, char *new)`	`Errno = EMLINK;`
	`return –1;`
	`librom` does not implement a true file system, so this function is not supported.

`int lseek (int fd, int ptr, int dir)`	`return 0;` `librom` does not implement a file system, so this function is a no-op.
`int read (int fd, char *ptr, int len)`	Implemented.
`caddr_t sbrk (int incr)`	Implemented.

`int stat (char *file, struct stat *st)`	`st->st_mode = 0;`
	`return 0;`
	`librom` does not implement a file system, so this function is not supported.

`int times (struct tms`	`return –1;`

`*tbuff)`	

`int unlink (char *filename)`	`Errno = ENOENT;`
	`return –1;`
	`librom` does not implement a file system, so this function is not supported.

`int wait (int *status)`	`errno = ECHILD;`
	`return –1;`
	`librom` only implements one ‘process’: the application program.

`int write (int file, char *ptr, int len)`	Implemented.

Table 4.7: librom’s implementation of the newlib support functions

4.11.2 Implementing stream I/O with librom

Librom uses a data structure called the I/O device table to keep track of the I/O devices that are available to newlib. Each entry in the table represents an individual I/O device. Any number of devices may be implemented in the table, with a null entry to mark the end of the table. Each device has a name associated with it, which is stored as a character string referenced by the Name field of the table entry. This allows your application code to access each device by name, using the open() and fopen() library functions.

Five functions are used to control each device: `open()`, `close()`, `read()`, `write()`, and `devicecontrol()`.Each entry in the I/O device table contains a pointer to the `open()`, `close()`, `read()`, `write()`, and `devicecontrol()` function for that device.

The I/O device table allows librom to support multiple, named devices in your embedded system, but it doesn’t necessarily make it easier to write the code that controls those devices. To tackle that issue, librom also contains driver code to implement input/output control of two classes of stream I/O device, which we refer to as Non-buffered (polled) devices and Buffered (interrupt-driven) devices. In other words, librom contains a set of functions that you can point to in your I/O device entries to implement either polled or interrupt-driven I/O on a particular device. These functions translate the function call interface required by newlib

into another, much simpler interface that is quite easy to implement for most peripheral systems in common use on current microprocessors. They also implement several handy features such as line editing, character translation, and character buffering, which are not addressed in newlib.

In order to use these functions, a second data structure is required for each device. This data structure points to a small set of functions that implement the hardware-level interface to your target system. In the case of a device using Buffered I/O, it also points to a set of FIFO buffers that are used to transfer characters to and from the device under DMA or interrupt control. Two data types are defined in the librom header files, tNonBufferedDevice and tBufferedDevice, to represent non-buffered and buffered devices respectively. The address of the device's data structure should be stored in the DeviceInfo field of the I/O device table entry.

4.11.3 The DeviceControl() function call

`DeviceControl()` allows access to several features of the librom driver library which do not fit into the standard C library's 'stream I/O' model. It is similar in concept to the Unix system call `ioctl()`, although much simpler and less capable. `DeviceControl()` has the following capabilities:

- Installs user alert functions to be called when data is received or the transmit buffer is emptied (Buffered devices only)
- Allows setting and querying the flags for a device, controlling character translation, line editing, etc
- Allows the application code to determine how many characters are stored in a device's input or output buffer
- A range of user-defined function codes is available for user-written driver routines.

The `devicecontrol()` function is defined in the header file `sys/IODctrl.h` as shown below:

```
long DriverControl (int filedes, int function,
...);
```

The `filedes` parameter refers to a file descriptor returned from the `open()` function call. If a device was opened with `fopen()` instead, then the newlib function `fileno()` will return the file descriptor.

Although the function returns a long, the actual value returned is interpreted differently depending upon the function code passed in the function parameter. The function parameter tells `DeviceControl()` which operation should be performed. Table 4.8 lists the various actions performed by `DeviceControl()`.

Function code	Description
`IODFN_GETFLAGS`	Returns the flags for the device. Device flags are documented in Table 4.9.
`IODFN_SETFLAGS`	This call requires an additional parameter after the function code, an integer containing the new flags to be assigned to the device. Device flags are documented in Table 4.9.
`IODFN_SETALERT`	This call sets an alert function that is called when data received on the device's input channel. This call requires an additional parameter after the function code, a function pointer defined as follows: `void (*DataReceived) (struct sBufferedDevice *Device);` The function will only be called if the device uses Buffered I/O; the non-buffered drivers do not implement any alert functions. The return value of this call is the old value of the alert function that was stored in the device's descriptor table.
`IODFN_SETEMPTY`	This call sets an alert function that is called when the transmit buffer is emptied on the device's output channel. This call requires an additional parameter after the function code, a function pointer defined as follows: `Void (*TXEmpty) (struct sBufferedDevice *Device);` The function will only be called if the device uses Buffered

	I/O; the non-buffered drivers do not implement any alert functions. The return value of this call is the old value of the alert function that was stored in the device's descriptor table.
`IODFN_GETAVAIL`	This call returns the number of characters in the device's receive buffer which are waiting to be read. This buffer is independent of any buffering which may be performed by newlib; see the documentation on `setvbuf()` for more information on newlib's buffering. If this function is not implemented by the device, the value –1 is returned. On a non-buffered device, this function can only return the values zero (no characters waiting) or one (at least one character is available).
`IODFN_GETBUFF`	This call returns the amount of space available in the device's transmit buffer. This buffer is independent of any buffering which may be performed by newlib; see the documentation on `setvbuf()` for more information on newlib's buffering. If this function is not implemented by the device, the value –1 is returned. On a non-buffered device, this function can only return the values zero (no space available) or one (there is room for at least one more character).
`IODFN_SETUSER`	This call sets a user parameter into the device's descriptor, returning the old value of the user parameter. The user parameter is a pointer of type `void *`. This value is not used by the device driver in any way, it is there simply for use by the user's application code. This function is only implemented on buffered I/O devices.
`IODFN_USER`	This is the first value of a range of function codes that are allocated for use by user-written drivers. Any function code with this value or higher may be implemented by user drivers without interfering with driver code from EST

Table 4.8: Actions implemented in the DeviceControl() function

A number of flags are maintained for each device, which control various options on the device. These flags may be set using the `IODFN_SETFLAGS` subfunction of `DeviceControl()`, and retrieved using the `IODFN_GETFLAGS` subfunction. Table 4.9 documents the function of each flag.

Flag name	Function
`IODF_CRIN`	Setting this flag enables translation of carriage returns to linefeed characters on the device's receiver. If the flag is cleared, no such translation is performed.
`IODF_NLOUT`	Setting this flag enables translation of linefeed characters to the string CR/LF (carriage return followed by a line feed) on the device's transmitter.
`IODF_EDIT`	Setting this flag enables line-editing features on the device's receive side. When used with the `IODF_ECHO` flags, this facilitates the use of `fgets()` to provide interactive entry of lines of text with simple editing features such as backspace processing. The specific operations supported are: • If a carriage return character is encountered during a call to `read()`, the function will return immediately with any characters received from the device up to the carriage return. • If a backspace character is encountered and one or more characters have been read from the device, the preceeding character will be removed from the buffer and the erase sequence "`\b \b`" (backspace, space, backspace) will be echoed to the device's transmitter if character echo is enabled. If there are no previous characters in the buffer when the backspace is encountered, then no action will be taken. In either case, the backspace character is not returned from the call to `read()`.
`IODF_ECHO`	This flag enables automatic echo of characters from the

	device's receiver to its transmitter during the call to `read()`.

Table 4.9: Device flags

4.11.4 Writing a non-buffered driver

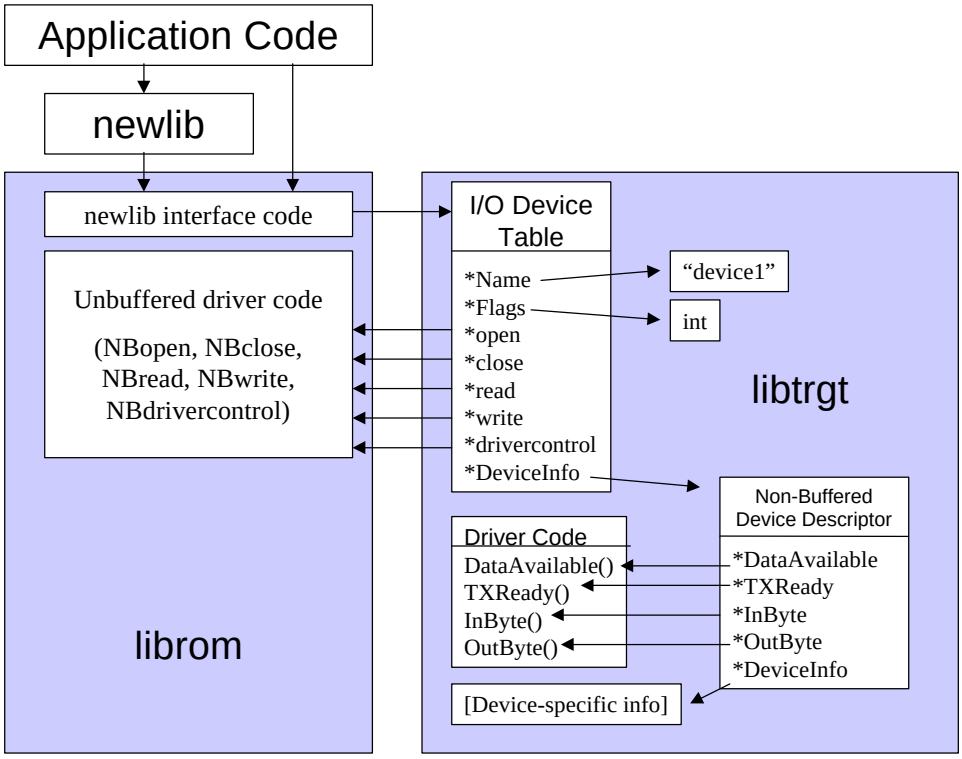

Figure 4.1: Non-buffered I/O device implementation

A non-buffered (polled I/O) driver needs four functions defined: `InByte()`, `OutByte()`, `DataAvailable()`, and `TxReady()`. Each function is described in the following subsections.

The actual name of the function is not important, since each function will be addressed by a pointer in a data structure, rather than by its name. What is important however, is that the

function must match the prototypes listed here, and must operate in the manner described. The data structure that stores the pointers to these functions, of type `tNonBufferedDevice`, is described in section 4.11.4.5.

4.11.4.1 InByte ()

```
int InByte (tIODev *Device);
```

`InByte()` reads the next available character from the device. The function should poll the hardware until a character is available, calling `IODeviceIdle()` until a character is ready and then returning its value.

4.11.4.2 OutByte ()

```
void OutByte (tIODev *Device, int Char);
```

`OutByte()` writes the next character `Char` to the device. The function should poll the hardware, calling `IODeviceIdle()`until the device is ready to accept the character.

4.11.4.3 DataAvailable ()

```
int (*DataAvailable) (tIODev *Device);
```

`DataAvailable()` returns a flag indicating whether a character is ready to be read from the device. The function should return a non-zero value if data is available to be read, otherwise it should return zero.

4.11.4.4 TxReady ()

```
int (*TxReady) (tIODev *Device);
```

`TxReady()` returns a flag indicating whether the device is ready to send another character. The function should return a non-zero value if the device is ready, otherwise it should return zero.

4.11.4.5 The tNonBufferedDevice structure

In order to make your driver functions accessible to librom, you must declare a variable of type `tNonBufferedDevice` and initialize the variable to point to the `InByte()`,

`OutByte()`, `DataAvailable()`, and `TxReady()` functions which control the device. The definition of `tNonBufferedDevice` looks like this:

```
typedef struct sNonBufferedDevice
{
     int (*DataAvailable) (tIODev *Device);
     int (*TXReady) (tIODev *Device);
     int (*InByte) (tIODev *Device);
     void (*OutByte) (tIODev *Device, int Char);
     void *DeviceInfo;
} tNonBufferedDevice;
```

Variables of type `tNonBufferedDevice` may be declared `const`, to conserve RAM space by placing them in read-only memory.

As shown above, `tNonBufferedDevice` is basically made up of pointers to your driver functions. There is one structure member, DeviceInfo, which is available for use by your driver routines; it is not used by the librom functions. This may be used (for example) to store a pointer to the hardware registers for the device.

If your device does not implement a particular function, for example it can receive data but not transmit, then you should place a null pointer in the structure member associated with the unimplemented function so that the librom functions will return an error to the application if it tries to access the unimplemented functionality.

4.11.4.6 Example Non-buffered driver

The file `NBio555.c` in the EXAMPLE subdirectory implements non-buffered I/O for the dual SCI ports on the Motorola MPC555. Since the SCI ports implement an identical set of hardware registers, a single set of driver functions was defined for both ports, and the `DeviceInfo` field of each `tNonBufferedDevice` structure was used to store a pointer to the first hardware register of each SCI port.

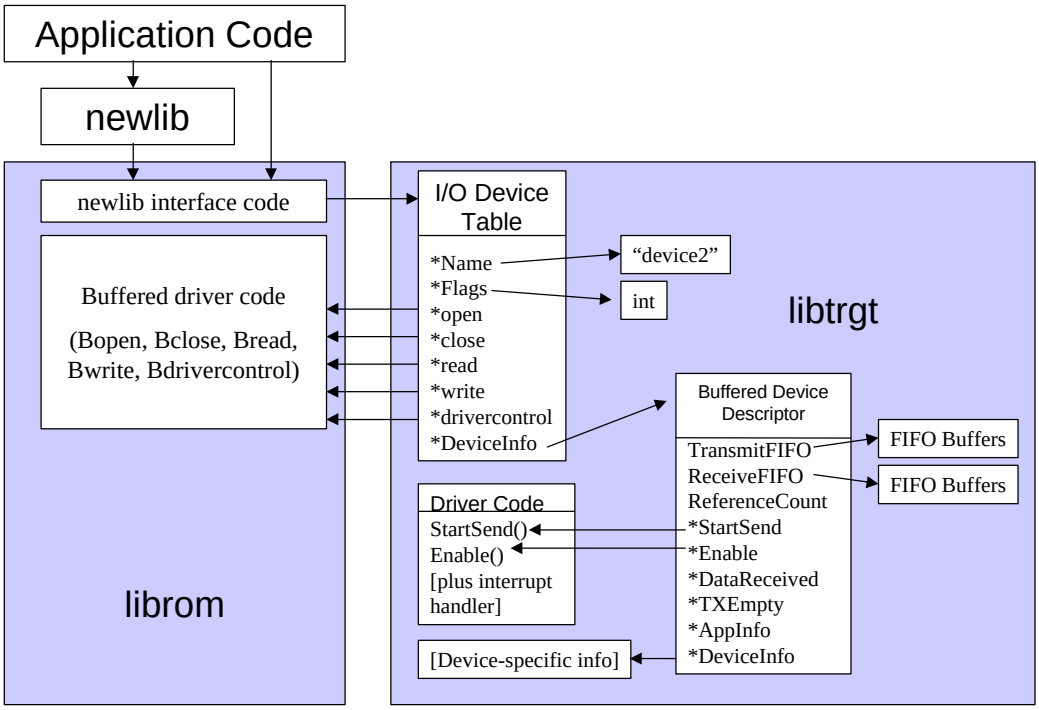

Figure 4.2: Buffered I/O device implementation

4.11.5 Writing a buffered driver

Buffered drivers use a similar structure to that employed by non-buffered drivers – a set of functions which control the device are addressed by a data structure of function pointers. We recommend that you read section 4.11.4 before diving into this one. However, there are some differences in the operation of buffered drivers, due to the inherent requirements and capabilities of interrupt- or DMA-driven transfers.

Buffered devices are structured under the assumption that data will be transmitted and received under control of some kind of background process; in most systems, this is usually an interrupt handler routine or a DMA controller. As such, they do not define functions to send and receive characters to/from the device; rather, two FIFO (first-in first-out) buffers are

defined, one for transmit and one for receive, and all data transfers happen through these FIFO buffers.

To transmit data, the upper-level routines in librom will move data into the transmit FIFO and then call the device's `StartSend()` function, whose purpose is to initiate data transmission within the device (ie enable the transmitter interrupt, set up the transmit DMA, etc).

Whenever data is available, triggering a receive interrupt or DMA transfer, the DMA controller or interrupt handler should write data into the receive FIFO. When a call is made to the librom's `read()` function by the foreground application code, in order to read data from the device, librom will remove the data from the driver's receive FIFO and place it in the location requested by the application, polling the FIFO in a loop if necessary in order to get the requested number of bytes.

4.11.5.1 The tBufferedDevice structure

In order to make your driver functions accessible to librom, you must declare a variable of type `tBufferedDevice` and initialize the variable to point to the functions that control the device. Unlike `tNonBufferedDevice`, variables of type `tBufferedDevice` must be located in RAM since several of the structure members are modified during execution.

The definition of `tBufferedDevice` looks like this:

```
typedef struct sBufferedDevice
{
     tFIFO TransmitFIFO;
     tFIFO ReceiveFIFO;
     void (*StartSend) (struct sBufferedDevice
*Device);
     int (*Enable) (int NewState);
     int ReferenceCount;
     void (*DataReceived) (struct sBufferedDevice
*Device);
     void (*TXEmpty) (struct sBufferedDevice
*Device);
     void *DeviceInfo;
     void *AppInfo;
} tBufferedDevice;
```

If your device does not implement a particular function, for example it can receive data but not transmit, then you should place a null pointer in the structure member associated with the unimplemented function so that the librom functions will return an error to the application if it tries to access the unimplemented functionality.

4.11.5.2 Enable ()

```
int (*Enable) (int NewState);
```

`Enable()` is called both to initialize the device before it is used, and to de-initialize (shut down) the device when it is no longer needed by the application.

The first time the application code makes a call to `open()` referencing a buffered device, the device's `enable()` function will be called with a non-zero value in `NewState`. Likewise, when all file descriptors referencing the device are closed, librom will call the device's `Enable()` function with `NewState` equal to zero. The `Enable()` function should test `NewState`, initializing the device if it is non-zero and disabling the device otherwise.

4.11.5.3 ReferenceCount

`ReferenceCount` is a counter used by librom to keep track of how many open file descriptors are attached to the device. Normally your device driver functions will not need to do anything with `ReferenceCount`. However, if you define a buffered device which is automatically opened and enabled upon power-up, you should set the `ReferenceCount` member to the number of file descriptors which are attached to this device, since there will be no call to `Enable()` to set it for you.

The most common case where this is necessary is when a console device is defined and attached to `stdin`, `stdout`, and `stderr`. In this case, the device is 'live' when the system starts executing main(), and the `stdin`/`stdout`/`stderr` file descriptors are already open and attached to this device. In this case, the reference count should be set to 3.

4.11.5.4 StartSend ()

```
void (*StartSend) (struct sBufferedDevice *Device);
```

StartSend is called by librom when it writes data to the device's transmit FIFO. This function should enable the transmitter interrupt, set up the transmitter's DMA controller, or do whatever else is necessary to start transmitting characters from the transmit FIFO.

4.11.5.5 Application alert functions

There are two function pointers in the tBufferedDevice data type which point to functions in the application code, rather than in the driver code. These function pointers permit the driver's interrupt handler to notify or 'alert' the application code when an event occurs in the driver. Two events may be handled in this way: the reception of a character on the device's receiver, and the emptying of the device's transmit FIFO. The function pointers may be set and retrieved by the application code through the DeviceControl() library function.

4.11.5.5.1 DataRecieved()

```
void (*DataReceived) (tBufferedDevice *Device);
```

This function will be called each time a character is received by the device. The function call is made from within the device's interrupt handler, so it is important that the function execute quickly in order not to impact on the performance of the product. Also, it's important not to call any C runtime library functions from within the DataReceived() handler, since most of them are not reentrant and will likely crash if called from within an interrupt handler.

4.11.5.5.2 TxReady()

```
void (*DataReceived) (tBufferedDevice *Device);
```

This function will be called whenever the last character is removed from the transmit FIFO by the device's interrupt handler. Similar to DataReceived() above, this function must execute quickly and must not call any C runtime library functions.

4.11.5.6 The interrupt handler(s)

The interrupt handler must contain code to recognize both the 'Data received' and 'TX empty' conditions, and call the appropriate function via the DataReceived/TxReady pointers in the tBufferedDevice descriptor. Remember that the programmer may choose not to define alert functions; if this is the case, then the associated function pointer will be null. The interrupt handler will have to check the value of the pointer and only call the function if a non-null pointer is found.

4.11.5.7 tBufferedDevice's transmit and receive FIFOs

The `tBufferedDevice` structure contains two members of type `tFIFO`, named `TransmitFIFO` and `ReceiveFIFO`. These FIFOs are used to store data being transmitted to and received from the I/O device.

The data type `tFIFO` implements a circular buffer that operates in a first-in, first-out manner. The data type is defined in the header file `sys/tFIFO.h`. A `tFIFO` structure contains a pointer to the FIFO's data buffer, an integer recording the size of the buffer, and 'in' and 'out' counters recording the current write and read position, respectively, in the buffer.

Several utility functions are available in librom to read and write data from/to a `tFIFO`. These functions are documented in the following subsections.

4.11.5.7.1 FIFOgetc()

```
static inline int FIFOgetc (tFIFO *FIFO, char
*Dest);
```

This function gets the next character available in the addressed FIFO, and writes it to location `*Dest`, updating the FIFO's read counter.

If the operation was successful, the function returns a non-zero value; if there was no character available, the function returns zero.

4.11.5.7.2 FIFOputc()

```
static inline int FIFOputc (tFIFO *FIFO, char c);
```

`FIFOputc()` writes the character `c` to the addressed FIFO, and returns a non-zero value if the write was successful. If the FIFO was full when the write was attempted, the function will return zero.

4.11.5.7.3 ReadFIFO()

```
int ReadFIFO (tFIFO *FIFO, int Max, char *Dest);
```

`ReadFIFO()` reads up to `Max` bytes from the addressed FIFO, writing them to the buffer addressed by `Dest`. The return value is the actual number of bytes that were read from the FIFO, which will have a value in the range of zero through `Max`.

4.11.5.7.4 WriteFIFO()

```
int WriteFIFO (tFIFO *FIFO, int Max, char *Source);
```

`WriteFIFO()` transfers up to `Max` bytes of data from the buffer addressed by `Source` into the addressed FIFO. The return value is the actual number of bytes transferred, which will be a value in the range of zero through `Max`. If the return value is less than `Max`, then it indicates that the FIFO became full during the call to `WriteFIFO ()`.

4.11.5.8 Example Buffered I/O driver

The file `332-io.c` in the `EXAMPLES` subdirectory implements buffered I/O on the SCI port of the Motorola MC68332.

4.11.6 Implementing the I/O device table

To complete the I/O driver, one more piece of information must be provided to librom: a table that relates device names to the `tBufferedDevice` or `tNonBufferedDevice` structure representing the device, and to the top-level librom driver functions which control them. To do this, define an array of type `tIODev` with the name `IODevices`. The array should have one entry for each device, and one more null entry (all zeros) at the end to mark the end of the table.

The data type `tIODev` is defined in the header file `sys/IODev.h`. The definition looks like this:

```
typedef struct sIODev
{
     char *Name;
     int *Flags;
     int (*open) (struct sIODev *Device, int
filedes, int flags, va_list args);
     int (*close) (struct sIODev *Device, int
filedes);
     int (*read) (struct sIODev *Device, int
filedes, void *_buf, size_t _nbyte);
     int (*write) (struct sIODev *Device, int
filedes, const void *_buf, size_t _nbyte);
     long (*drivercontrol) (struct sIODev *Device,
```

```
int filedes, int function, va_list args);
     void *DeviceInfo;
} tIODev;
```

This data type is also largely made up of pointers to functions.You are required to define only the `Flags` member, which defines the options active on the device, and the `DeviceInfo` member, which should contain the address of the `tNonBufferedDevice` or `tBufferedDevice` structure which defines your device. librom has two sets of functions, one for buffered devices and one for non-buffered devices, which should be referenced in the other members. If your device does not implement a particular function, for example it can receive data but not transmit, then you should place a null pointer in the structure member associated with the unimplemented function so that the librom functions will return an error to the application if it tries to access the unimplemented functionality.

Structure member	Buffered function	Non-buffered function
`open()`	`Bopen()`	`NBopen()`
`close()`	`Bclose()`	`NBclose()`
`read()`	`Bread()`	`NBread()`
`write()`	`Bwrite()`	`NBwrite()`
`drivercontrol()`	`Bdrivercontrol()`	`NBdrivercontrol()`

Table 4.10: Buffered and Non-buffered functions for the I/O device table entries

Here is an example of a definition of `IODevices`, taken from the MC68332 buffered driver:

```
tIODev IODevices [] = {
{"console", &ConsoleFlags, Bopen, Bclose, Bread,
Bwrite, Bdrivercontrol, &Console},
{0}
};
```

In this example we define a table containing a single device called “console”. It is a buffered device, so it references the `Bxxx` functions from librom. The `DeviceInfo` member points to a structure of type `tBufferedDevice`, called `Console`.

Here’s another example, this time the non-buffered driver from `NBio555.c`.

```
tIODev IODevices [] =
{
{
      "SCI1",
      &SCI1flags,
      0,
      0,
      NBread,
      NBwrite,
      NBdrivercontrol,
      &nbSCI1
},
{
      "SCI2",
      &SCI2flags,
      0,
      0,
      NBread,
      NBwrite,
      NBdrivercontrol,
      &nbSCI2
},
{0}
};
```

In this example, the table defines two devices, one called `SCI1` and the other called `SCI2`. Both are non-buffered devices, so they reference the `NBxxx` functions from librom. They each reference a `tNonBufferedDevice` structure in their respective `DeviceInfo` fields. We have defined routines to read and write data from/to these devices, but no functions are available to determine if data is available from the receiver or whether the transmitter is ready, so we put null pointers in the `DataAvailable` and `TxReady` fields.

4.11.7 Building and linking application programs with librom

When you have written your driver functions, they should be compiled and linked with the other source files that make up your program. In the example below, we will compile an

application program `hello.elf`, using the file `332-io.c` to provide the necessary I/O drivers:

```
C:\XGCC\EXAMPLE> gcc –b m68k-elf –O2 –g hello.c
332-io.c –o hello.elf –wl,--defsym,__ram_size=24k,-
--defsym,__stack_size=4k –T ram.ld
```

5 Wrapping Up

We hope that you will find the gnu tools presented here to be as useful and productive as we have. One of the most important factors in using these tools is networking with other users. In this section we would like to present some of the resources that we have found to be useful in achieving this end.

5.1 Additional resources

5.1.1 Web sites

- The EST home page:
  http://www.estc.com
- The CrossGCC Frequently-Asked Questions (FAQ) web page:
  http://www.objsw.com/CrossGCC
  This document discusses how to build cross-compilers using Gnu CC.
- PowerPC SVR4 function calling conventions:
  http://www.esofta.com/pdfs/SVR4abippc.pdf
- PowerPC Embedded Application Binary Interface:
  http://www.esofta.com/pdfs/ppceabi.pdf or
  http://www.mot.com/SPS/ADC/pps/download/8XX/ppceabi.pdf

- Coldfire developers will find the Wildrice web site to be a valuable resource and jumping-off point to other resources on the web:
  http://www.WildRice.com/ColdFire/
- Another excellent ColdFire resource is David Fiddes' site. David has also done a port of the Gnu tools to Win32, hosted on cygwin, and among other things has provided runtime libraries for the Motorola ColdFire evaluation boards.
  http://www.users.surfaid.org/~fiddes/coldfire/
- If you are interested in using the Standard Template Library in your C++ programs, this page has some good links:
  http://www.cyberport.com/~tangent/programming/stl/resources.html
- Tools for manipulating S-record files (in order to program EPROMs, embed S-record information in C source, and other handy stuff) may be found here:
  http://www.tip.net.au/~millerp/srecord.html

5.1.2 Mailing lists

- The crossgcc mailing list discusses issues relating to the use of the Gnu tools as cross-compilers, with special emphasis on targeting embedded systems. Subscribe to this list by sending an empty e-mail to crossgcc-subscribe@sourceware.cygnus.com. Messages are posted to the list by sending them to crossgcc@sourceware.cygnus.com.
- There is an excellent mailing list dedicated to the Motorola Coldfire architecture. You can subscribe to it by sending an e-mail to requests@WildRice.com with the text "subscribe ColdFire" in the body of the message; no subject is required. Leave out the double quotes as well, just type the words in the body of the message. To post messages to the list, send mail to ColdFire@WildRice.com.

5.1.3 Newsgroups

- comp.sys.m68k
- comp.sys.powerpc
- comp.sys.powerpc.tech

6 Index

|

1

6

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W