Embedded MCore Development Guide

MULTI 2000 Release

DISCLAIMER

GREEN HILLS SOFTWARE, INC. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Green Hills Software, Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Green Hills Software, Inc. to notify any person of such revision or changes.

Revision History

Revision	Release Date	Location of Revision(s)

PubID: D13B-I1299-89NG
Timestamp:December 10, 1999 10:34 am
Embedded MCore Development Guide

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

CONTENTS

Preface

About this Manual

This manual describes the Green Hills cross development tools for the MCore family of microprocessors. Cross development means using one computer system, called the host, to write, compile, and debug programs for execution on a different computer system, called the target. The Green Hills cross development products are available for many different hosts and many different operating systems. The examples in this manual apply to a UNIX environment on a Sun workstation; users in other environments should make the appropriate adjustments. The required adjustments are generally obvious; exceptions are explained in the accompanying text.

This manual also presumes that Green Hills products are installed in the directory **/usr/green**. If this is not the case, substitute the correct directory and place it in your path.

Typographical Conventions

Convention	Example	Description
bold text	-noansi	name of program, command, directory, or file
bold characters in quotes	"**A**"	name to enter as shown, without quotes
courier	setenv TMPDIR	samples of code, or instructions to enter
italic text in a command line	**-o** *filename*	place-holder for user-supplied information
square brackets, []	**.macro** *name* [*list*]	encloses optional commands or terms
square brackets [] around boldface **default**	Specifies **char** as signed [**default**].	command or option is the default

For example, in the command description

ccmcore [*-cpu=processor*] *filename*

the command **ccmcore** should be entered as given, the *-cpu=processor* is optional with the appropriate cpu option replacing *processor*, and the appropriate file name replacing the word *filename*.

What This Manual Covers

Chapter	Provides
1. Introduction	Overview of components and toolset operation
2. Building an Executable	Introduction to the Compiler Driver
3. The Toolset	How the compiling, linking, and debugging tools work
4. The MCore Processor	Description of the MCore target environment
5. Embedded Features	Special requirements for embedded developers
6. Debug Formatting	How to use .dbg files
7. ELF Files	Organization of Executable and Linking Format files
8. Compiler Driver Options	Description of all the compiler options
9. Macro Assembler	How to use the MCore macro assembler
10. Macro Assembler Directives	Explanation of all the MCore macro assembler directives
11. MCore Macro Assembler Reference	MCore addressing modes and instruction formats
12. The Librarian	How to use the librarian
13. ELXR	How to use ELXR
14. Utility Programs	Description of all supported and unsupported utilities
15. Runtime Environment and Library Organization	Structure of the Green Hills runtime environment and how to modify and customize it
16. The MCore Simulator	How to use MULTI with the Simulator

Chapter	Provides
App. A, Enhanced asm Facility	Introducing assembly language instructions into C code
App. B, Viewpathing	A hierarchical method for searching multiple directories for input files
App. C, C Runtime Libraries	Listings of the Green Hills C Library

1

Introduction

This chapter contains:

- Components of the Toolset

This manual is the primary documentation for MCore cross development. Listed here are various tools you will need, how they relate to each other, and how those features are unique to cross development.

Components of the Toolset

The complete Green Hills MCore cross development toolset includes the following components.

Compiler Drivers

A compiler driver is a program which invokes the other components of the tool set to process a program. There is a separate compiler driver for each source language. The drivers use command line arguments and source file extensions to determine which compiler or assembler to invoke for each source file, then sequence the resulting output through the subsequent linker and conversion utilities, relieving the user of the burden of invoking each of these tools individually.

CompilersEach Green Hills optimizing compiler is a combination of a language-specific front end, a global optimizer, and a target-specific code generator. Green Hills provides compilers for five languages: Ada, C, C++, FORTRAN, and Pascal, including all major dialects. All languages for a target use the same subroutine linkage conventions. This allows modules written in different languages to call each other. The compilers generate assembly language.

AssemblerThe relocatable macro assembler translates assembly language statements and directives into a relocatable object file containing instructions and data.

LibrarianThe Librarian combines object files created by the Assembler or Linker into a library file. The linker can search library files to resolve internal references.

LinkerThe Linker combines one or more ELF object modules into a single ELF relocatable object file or executable program.

DebuggerThe MULTI Debugger is a windowing source level debugger that debugs programs written in Ada, C, C++, FORTRAN, Pascal, and assembly language. MULTI can debug a program being executed by a simulator or by a target.

SimulatorThe Simulator is a program which executes on the host system and simulates the execution of the MCore instructions.

ROM Monitor

The ROM Monitor is a program which resides on a target system and which interfaces with the MULTI Debugger to enable it to download programs to that target and debug them.

2

Building An Executable Program

This chapter contains:

- How to Build a Program for Use with the MULTI Debugger
- The Compiler Driver
- How to Build a C Executable Program
- How to Build a C++ Executable Program
- How to Build Programs with C and C++ Modules

You can create an executable program by compiling source files written in a high-level language such as Ada, C, C++, or FORTRAN into assembly code, assembling assembly language files into object files, and linking together these object modules with object module libraries, into an executable program.

To simplify this task and coordinate many diverse programs and files, Green Hills provides a program for each language called the compiler driver. This chapter describes the Green Hills compiler driver and provides instructions and examples for building your executable program and debugging it with the MULTI Debugger.

How to Build a Program for Use with the MULTI Debugger

Green Hills MULTI provides a powerful source level debugger. To take advantage of the MULTI Debugger, use the **-G** option when building the executable program. This option causes the Green Hills compilers to place extensive information regarding variables, data types, and source files into auxiliary debug file information that MULTI uses (see Chapter 6, “Debug Formatting”).

The **-G** option may appear anywhere on the driver command line when source files are compiled and also when object files are linked, as shown in this example:

```
% ccmcore -G file1.c file2.c -o program
                    % multi program
```

MULTI also supports debugging of optimized code; you can use the **-G** option with optimization enabled, although for best debugging results, the optimizer should be disabled.

The Compiler Driver

Starting with source files written in you can build an executable program in one command line, using the Green Hills compiler drivers C or C++.

The compiler driver performs three major functions:

- Compiles source language files
- Assembles assembly language files
- Links together object files and libraries

You can invoke the driver from the command line. The driver calls the appropriate compiler with the correct default options. By default, the driver uses the linker to link in object files from the appropriate libraries.

The compiler driver for each language is:

ccmcoreC compiler driver

cxmcoreC++ compiler driver

The driver begins with a list of input files and looks at the extension of each filename to determine the file types and what compilation steps to perform on that file. For example, a file with a **.c** extension is a C source file, and a file with a **.f** or **.for** extension is a FORTRAN source file. Each file needs to be compiled with the appropriate language compiler, assembled, and then linked. Source files of different languages can be included within the same executable. See the **-language** option in the Mixing Languages chapter in either the *Green Hills C* or *FORTRAN User's Guides* for more information.

The syntax for the compiler driver command is:

driver_name [options] filename(s)

*driver_name*The name of the driver for the language that you are using.

*options*Represent any combination of compiler driver options. These options may be placed either before or after *filename(s)* on the command line.

filename(s) Represents the source file or files you wish to compile, assemble, or link. A space is required between each filename.

To build executable programs, support routines in the form of libraries and startup files are often needed in addition to the routines that you have provided. These support routines often perform tasks such as interfacing with an operating system. By default, the compiler driver will include the appropriate libraries and startup files based on the languages, target processor, and compatibility modes specified when compiling and linking.

The compiler driver recognizes certain filename extensions listed in the following table. It determines each file type from the extension and processes the file accordingly.

Extension	Assumed file type
.ada .adb .ads	Ada source file
.c .i	C source file
.cxx .C .cpp .cc	C++ source file
.ii	C++ templates information file
.f .for	FORTRAN source file
.o	object file
.a	library file
.s	assembly language file
.mco	assembly language file with C preprocessor directives
.inf	Inline and dependency intermediate files
.dbo, .dba, .dlo, .dla, .dnm	Debug information

How to Build a C Executable Program

To build an executable from a C source file called **demo.c**, enter the following command:

```
% ccmcore demo.c
```

The driver recognizes **demo.c** as a valid C source file by its **.c** extension and invokes the C compiler. The compiler produces an assembly code file, which is then sent to the assembler, which produces an object file which is sent to the linker and linked with the appropriate libraries selected by the driver.

If no errors occur, an ELF format executable file called **a.out** is created in the current directory. You can rename the output file with the **-o** compiler driver option.

You can rename the output file with the **-o** option. For example:

```
% ccmcore demo.c -o demo
```

This command creates an ELF format file called **demo** in the current directory. The filename must immediately follow the **-o** option.

When a single source file is compiled and linked, all intermediate files are deleted.When more than one file is given to the driver, any object files that are

created in the process are not deleted. This is convenient when only one of the files must be recompiled. For example, suppose that you compile several files:

```
% ccmcore demo.c file1.c file2.c
```

You then discover that **demo.c** must be modified. After editing **demo.c**, you can build the executable with this command:

```
% ccmcore demo.c file1.o file2.o
```

Since **file1.c** and **file2.c** have not been modified, it is possible to use the object modules **file1.o** and **file2.o** created by the previous compilation.

Assembly source files may be input to the compiler driver if the name of the file ends with **.s**. For example:

```
% ccmcore demo.s
```

For assembly source files, the compiler driver first invokes the assembler, then the linker produces the file **a.out**in the current directory.

If you use the **-c** option, the compiler driver stops after creating an object file for each source file on the command line. The following command line produces two relocatable object modules, called **demo.o** and **file.o**, in the current directory:

```
% ccmcore -c demo.c file.s
```

If only one object file is created, you can use the **-o** option with the **-c** option to rename the object file. The new name must contain the suffix **.o**. For example, the following command line creates the relocatable object module **newdemo.o** in the current directory:

```
% ccmcore -c demo.c -o newdemo.o
```

The compiler driver links relocatable object modules into an executable file. If all of the names of the input files to the driver end in**.o**, the compiler driver invokes the linker only. The following command line links the **demo.o**, **file1.o**, and **file2.o** object modules with the necessary startup code and libraries to produce the file **a.out**:

```
% ccmcore demo.o file1.o file2.o
```

How to Build a C++ Executable Program

To build an executable from a C++ source file called **demo.cxx**, enter the following command:

```
% cxmcore demo.cxx
```

The driver recognizes **demo.cxx** as a valid C++ source file by its**.cxx** extension and invokes the C++ compiler. The compiler produces an assembly code file, which is then sent to the assembler, which produces an object file. The object

file is then sent to the linker and linked with the appropriate libraries selected by the driver.

If no errors occur, an ELF format executable file called **a.out** is created in the current directory. You can rename the output file with the **-o** compiler driver option. For example:

```
% cxmcore demo.cxx -o demo
```

This command creates an ELF format executable file called **demo** in the current directory. The filename must immediately follow the **-o** option. When a single source file is compiled and linked in this way, all intermediate **.s** and **.o** files are deleted.

When more than one file is given to the driver, any object files that are created in the process are not deleted. This is convenient when only one of the files must be recompiled. For example, suppose that you compile several files:

```
% cxmcore demo.cxx file1.cxx file2.cxx
```

You then discover that **demo.cxx** must be modified. After editing **demo.cxx**, you can rebuild the executable with this command:

```
% cxmcore demo.cxx file1.o file2.o
```

Since **file1.cxx** and **file2.cxx** have not been modified, it is possible to use the object modules **file1.o** and **file2.o** created by the previous compilation.

Assembly source files may be input to the compiler driver if the name of the file ends with **.s**. For example:

```
% cxmcore demo.s
```

For assembly source files, the compiler driver first invokes the assembler and then the linker to produce the file **a.out**in the current directory.

If the **-c** option is used, the compiler driver stops after creating an object file for each source file from the command line. The following command line produces two relocatable object modules, called **demo.o** and **file.o**, in the current directory:

```
% cxmcore -c demo.cxx file.s
```

If only one object file is created, you may use the **-o** option with the **-c** option to rename the object file. The new name must contain the suffix **.o**. For example, the following command line creates the relocatable object module **newdemo.o** in the current directory:

```
% cxmcore -c demo.cxx -o newdemo.o
```

The compiler driver links relocatable object modules into an executable file. If all of the names of the input files to the driver end in**.o**, the compiler driver invokes only the linker. The following command line links the **demo.o**, **file1.o**,

and **file2.o** object modules with the necessary startup code and libraries to produce the file **a.out**:

```
% cxmcore demo.o file1.o file2.o
```

How to Build Programs with C and C++ Modules

It is possible to combine both C and C++ modules into a single program. You should use the C++ driver, **cxmcore**, to build any executables containing modules written in C++. The C++ driver is designed to handle the special requirements for linking C++ programs.

3

The Toolset

This chapter contains:

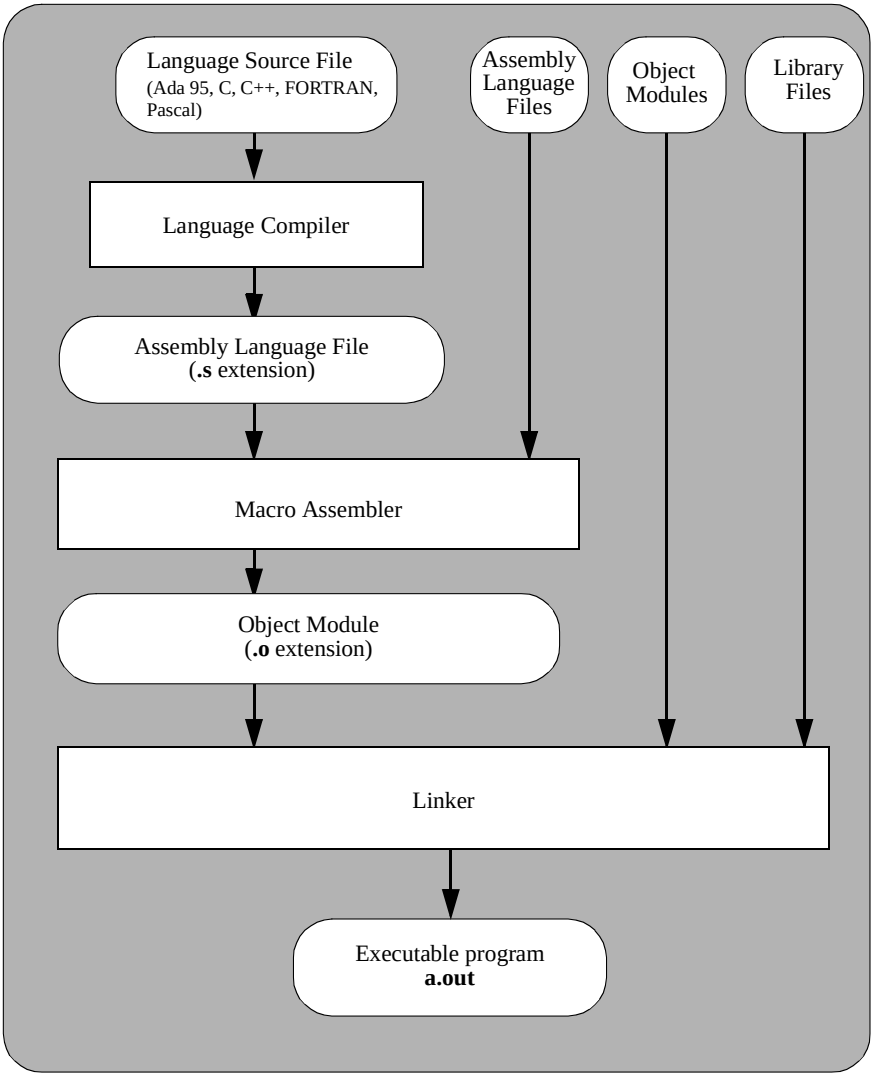
- How to Compile and Link an Executable Program
- Green Hills MCore Cross Compilers
- The MCore Macro Assembler, asmcore
- Object Module Librarian, ax
- The MCore Linker, elxr
- Header Files
- Support Routines and Libraries
- Debugging and Running the Program

The MULTI Builder controls the compiling, assembling, and linking. It depends on many other tools to perform its tasks. These tools include executable programs, support files, and libraries.

How to Compile and Link an Executable Program

Figure 1 illustrates the flow of files through the tool chain when the MULTI Builder is invoked. The builder first calls the appropriate compiler for each source file and produces an assembly code file. The builder then invokes the assembler to produce an ELF object file. The builder then invokes the linker to produce an ELF executable file. MULTI can debug this ELF executable file. Figure 2 illustrates the C++ compilation procedures.

Figure 1 **Flow of Source Files Through the Tool Chain**



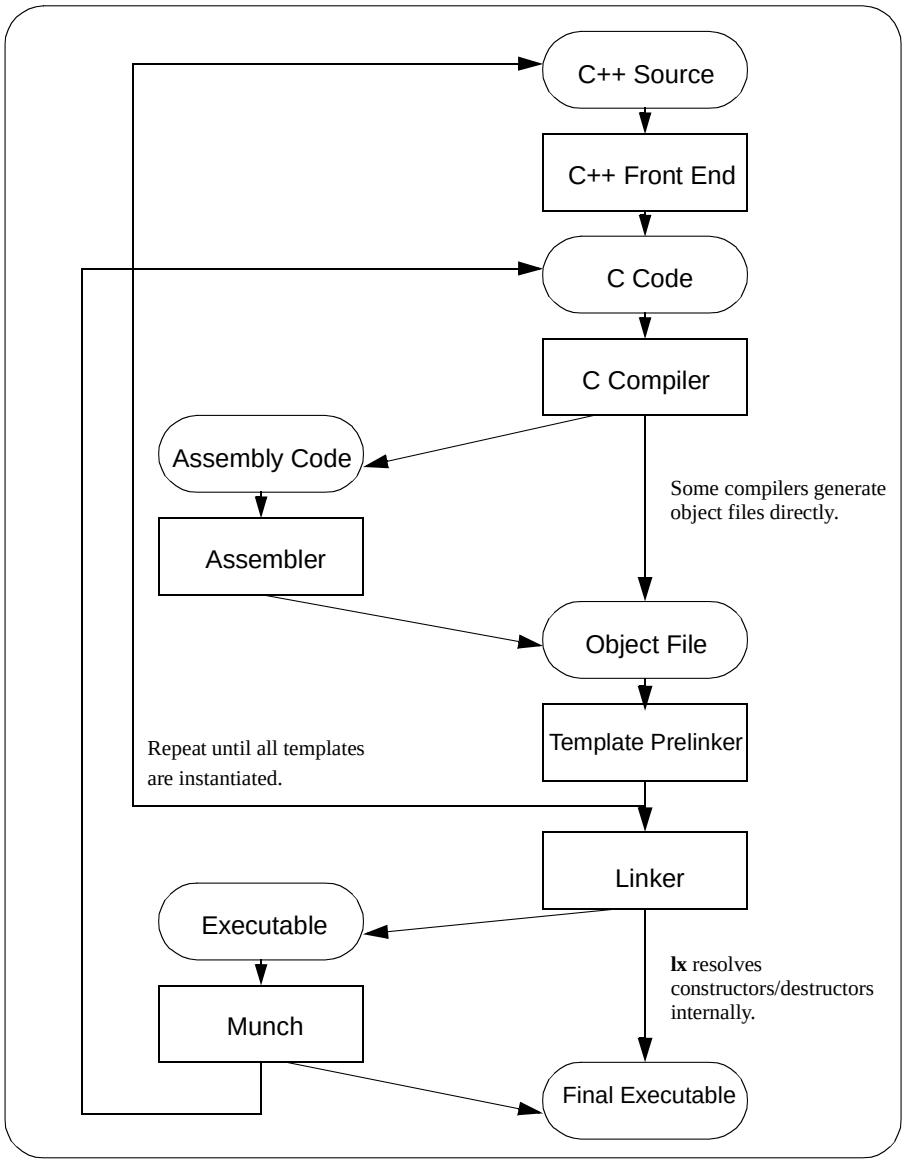


Figure 2 **C++ Compilation Process**

Green Hills MCore Cross Compilers

The Green Hills MCore Cross Compilers are an integrated family of highly optimizing compilers. Each compiler is a combination of a language-specific front end, a global optimizer, and a target-specific code generator. The compilers use compatible subroutine calling conventions. This allows modules written in different languages to be mixed. The output file from any of the compilers is an assembly code file.

Green Hills provides compilers for Ada, C, C++, FORTRAN, and Pascal. Each compiler supports the major dialects of the associated language.

The MCore Macro Assembler, asmcore

While programs executing on a processor are capable of very powerful functions and can work with complex data structures, processors themselves understand only binary sequences of “machine code” and operate only on binary sequences of data. The machine code forms sequences of instructions for the processor to perform; the data is manipulated by these instructions.

Since humans have difficulty working with such binary sequences, each processor type has a human-readable “assembly language.” Usually, there is a one-to-one correspondence between each assembly language instruction and its equivalent machine code form.

This assembly language typically supplies not only a textual representation for each instruction, but also a set of “directives” where the programmer can give instructions to the assembler itself. Directives specify data types, generate data values, specify alignment requirements for the machine code or data, and so on.

The MCore Macro Assembler **asmcore** takes the assembly language statements and directives of the MCore assembly language program presented to it and translates them into the equivalent MCore processor machine code and data formats. The resulting file produced is an object file, or object module. See Chapter 9, “Macro Assembler”, for more information.

Usually, the linker is able to resolve all external references and produce a “fully-linked” output module which is made executable by the operating system. Alternately, if instructed to do so, the linker may simply combine several object modules into a bigger object module, perhaps still with some unresolved references.

The linker is usually instructed to link object modules together with one or more “libraries.” A typical library contains a large number of object modules of its

own. The linker extracts from the library only those modules which it needs in order to resolve the external references in the object modules presented to it. Libraries are useful for providing commonly used modules in an easily accessible format.

Object Module Librarian, ax

The Green Hills MCore Librarian program combines object modules created by the Assembler or Linker into a library file. See Chapter 12, “The Librarian”, for more information. The linker can search library files for components to resolve internal references. A module from a library is only included in a program when it is referenced in the program.

The MCore Linker, elxr

The linker for the MCore toolset is **elxr** for ELF.

The MCore linker takes one or more object modules and combines them into a single executable output module. The relocation section of each given module “resolves” its external text and data references with the module(s) containing the required text and data. See Chapter 13, The ELXR Linker, for more information.

Header Files

Green Hills provides header files for use with C and C++ source files. These files are accessed by placing the **#include** directive in the source file. When the compiler sees **#include** *file*, it first searches in the directory containing the source file, then in the directories specified with the **-I** option, and finally in the default directories.

For C, the default directories are:

/usr/green/mcore/include
/usr/green/ansi
The contents of **/usr/green/ansi** shown below.

assert.h	ctype.h	errno.h	float.h	inline.h	ghcxx.h	interrup.h
interrupt.h	limits.h	locale.h	math.h	setjmp.h	signal.h	stdarg.h
stddef.h	stdio.h	stdlib.h	string.h	strings.h	time.h	varargs.h

For C++, the default directories vary depending on the C++ mode in use.

Support Routines and Libraries

To build executable programs, special support routines are often needed in addition to the routines you provide. These support routines often perform tasks which cannot be done by the user, such as interfacing with the underlying operating system. These routines are kept in libraries.

In addition to libraries, other special files are required by an executable program, such as the startup file and the default linker directive file. The compiler driver automatically specifies these files and libraries when invoking the linker, unless the option **-nostdlib** is specified. This option does not add any default startup files or libraries to the linker command line.

Startup File

When linking a program, the compiler driver normally specifies a startup file such as **crt0.o** before any user specified object files or libraries. This file contains a function named **_start** which is the default entry point for the program. The **_start** function performs initialization and then invokes **main()**. Please see Chapter 15, “Runtime Environment and Library Organization”, for more information.

Libraries

See Appendix C, “C Runtime Libraries”, for a list of functions included in the libraries below.

ANSI C Library, libansi.a

The functions documented in the ANSI C Standard are contained in this library and **libind.a**, described below. After all user files and libraries on the command line, **libansi.a** should always be listed. Only **libind.a** should appear after **libansi.a**.

Language Independent Library, libind.a

All transcendental math functions, such as **sin** and **sqrt**, are in the **libind.a** library. In addition, low-level support routines and system service functions are here. This library should always be the last file on the command line.

Debugging and Running the Program

Once you have created an executable, the next steps are to debug and run the program. Green Hills MULTI development environment provides a source-level debugger used with programs executing on an actual target or executing on a simulated target.

MULTI Debugger

The MULTI Debugger is part of the MULTI Software Development Environment. MULTI runs on the host machine while the application to be debugged is running either under the MCore Simulator, the ROM Monitor, or on a target system interfaced through an In-Circuit Emulator Server.

Simulator

The Green Hills MCore Simulator is a program that executes on the host and simulates the execution of the MCore microprocessor at the instruction level.

In-Circuit Emulator Server

The ICE Server is a program which runs on the host computer with MULTI and acts as an intermediary between MULTI and an in-circuit emulator connected to the host. The ICE Server translates debugging requests, transmits them to the emulator, and returns the responses from the emulator in a format recognizable to MULTI.

4

The MCore Processor

This chapter contains:

- MCore Characteristics
- Compiler Output Format
- Register Usage
- Structure Packing
- Calling Conventions
- Interrupt Processing in C and C++

This chapter describes the MCore target environment.

MCore Characteristics

The MCore processor has the characteristics shown in the following table:

Characteristic	Description
Memory addressing	Byte-addressed with 32-bit addresses.
Bit numbering	Bit 0 is least-significant bit.
Byte ordering	Big endian by default. The most significant byte of a multi-byte value is stored at the lowest address.
Stack alignment	8-byte alignment.
Floating-point format	IEEE 754 format (32 and 64 bits) with the most significant byte at the lowest address.
Character encoding	ASCII.
C/C++ bit field allocation	starts at most-significant bit.
C/C++ maximum bit field size	Four or fewer bytes.
C/C++ struct, union, array alignment	Aligned to the maximum alignment of any of its components.

The following tables list the data type alignments for C, C++, and FORTRAN.

C/C++ Data Type	Size	Alignment
int	32	32
long	32	32
long long	64	64
*	32	32
short	16	16
char	8	8
float	32	32
double	64	64
long double	64	64
unsigned	32	32
unsigned char	8	8
unsigned short	16	16
enum (default)	32	32
enum (option)	8, 16, 32	varies

FORTRAN Data Type	Size	Alignment
REAL	32	32
REAL*8	64	32

FORTRAN Data Type	Size	Alignment
DOUBLE PRECISION	64	32
CHARACTER	8	8
INTEGER*1	8	8
INTEGER*2	16	16
INTEGER	32	32
LOGICAL*1	8	8
LOGICAL*2	16	16
LOGICAL	32	32
COMPLEX	64	32
COMPLEX*8	64	32
COMPLEX*16	128	32
DOUBLE COMPLEX	128	32

Compiler Output Format

By default, the output of the compiler is MCore Assembly Language.

Register Usage

There are 16, 32-bit general purpose registers which can be used for both integer values and single-precision floating point values. There are also a set of control registers. The registers are shown in the following tables:

Register Name(s)	Usage
r0	Stack pointer
r1	Scratch register
r2-r3	Parameter registers, return value
r4-r7	Parameter registers
r8-r13	Permanent registers
r14	Permanent register, frame pointer
r15	Link pointer

Registers r1-r7, r15 are volatile; their contents may be destroyed by a function call. Registers r0, r8-r14 are non-volatile; they will be preserved across function calls.

Name	Usage
PSR	Processor status register

Name	Usage
VBR	Vector base register
EPSR, FPSR, EPC, FPC	Exception shadow registers
SS0-SS4	Supervisor storage registers
GCR	Global control register
GSR	Global status register
PC	Program counter

Structure Packing

The Green Hills compilers always allocate fields of a structure in the order specified in the declaration. It may be necessary for the compiler to insert one or more bytes of padding to ensure that a field begins at an offset from the beginning of the structure which is a multiple of the alignment of that field. The alignment of a field is determined by its type. The maximum alignment of a field is eight bytes. This alignment applies to fields of type **double**, **long double**, and **long long**. Fields of type **float**, **int**, and **long**, and pointer types have four byte alignment. Fields of type **short** have two byte alignment and fields of type **char** have one byte alignment.

Packing is a feature which reduces the maximum padding the compiler inserts between fields in order to gain storage-efficient data structures. If a structure is packed to two bytes, then each field has a maximum alignment of two bytes, and at most one byte of padding will be inserted between fields. The structure itself will also have a maximum alignment of two bytes.

The command line options **-Zp1**, **-Zp2**, and **-Zp4** specify the default packing in bytes for all structures.

In addition, **#pragma pack()** controls the packing of an individual structure. The pragma must appear before the beginning of the declaration which lists the fields of the structure. The pragma should not be used inside of a structure declaration. If 1, 2, or 4 appear between the **()**, the packing in effect changes until the next **#pragma pack()**. If a number is not present between the **()**, packing resets to the default.

Example:

```
struct s {
char c;
int i, j;
} a;
```

```
#pragma pack(2)
struct s x;
struct s2 {
char c;
int i, j;
} b;
#pragma pack()
struct s2 y;
```

The size of **a** and **x** are both 12. Three bytes of padding appear between field **c** and field **i**. #**pragma pack(2)** did not affect the declaration of **x**, since **struct s** was already declared.

The size of **b** and **y** are both 10. One byte of padding appears between field **c** and field **i**. **#pragma pack()** did not effect the declaration of **y**, since **struct s2** was already declared.

Be aware that the use of **#pragma pack** may generate structures in which some of the fields are impossible or inefficient to access. The programmer assumes responsibility for avoiding access to misaligned fields, which may cause fatal compile-time errors or other serious problems.

Calling Conventions

A procedure, subroutine, or function call is assumed to destroy the contents of all registers except **r8** through **r14** unless **#pragma ghs interrupt** is used.

Arguments

Call arguments are evaluated first from left to right, then the remaining non-call arguments are evaluated from left to right.

In C and C++, each scalar argument is extended to a 32-bit value after it is evaluated unless the corresponding formal parameter has a floating point type and, in C and C++, an ANSI prototype is visible. In this case, the argument is converted into either a 32-bit or 64-bit floating point value according to the formal parameter.

In C and C++, each floating point argument is extended to a 64-bit value after it is evaluated, unless the corresponding formal parameter is either single precision floating point or integer type and, in C and C++, an ANSI prototype is visible. If the formal parameter is single precision, the argument is converted to a 32-bit floating point value. If the formal parameter is scalar, the argument is converted to a 32-bit scalar value.

Any further type conversion is performed upon entry to the called procedure.

Arguments are assigned stack offsets from left to right. The first argument is always at offset zero. The size of the first argument is rounded up to a multiple of four bytes and added to its offset to determine the offset of the second argument. If the second argument requires 8-byte alignment and its offset would not otherwise be a multiple of eight bytes, its offset is increased by four bytes. This is repeated until offsets have been assigned to all arguments. If the argument area is larger than 24 bytes, then a space large enough to hold this argument area less 24 bytes is present on the stack immediately before the call.

In general, the arguments are allocated to the stack according to their stack offset unless it is possible to place them in registers.

Arguments with offsets 0 through 20 will be placed in registers **r2** through **r7**, respectively. Thus in C and C++, scalar, pointer, and floating point arguments are eligible to pass in registers, as well as some structures and unions.

A **varargs** function in K&R only has **va_alist** as its arguments, and they are all considered to be passed in memory. Therefore on entry to a varargs function, all the parameter registers (**r2** through **r7**) are first saved on the stack so that all arguments can be accessed from the stack.

For a **stdargs** function in ANSI, all arguments starting from "**...**" are considered to be passed on the stack. On entry to a stdargs function, if "**...**" is in one of the parameter registers (**r2** through **r7**), then that register plus all parameter registers following it are first saved on the stack, so that all arguments starting from "**...**" can be accessed from the stack.

For a **stdargs** function (as described in the MCore Applications Binary Interface), if the address of any of the parameters is taken, all of the parameter registers are saved on the stack. This is for compatibility with legacy code that assumes that all parameters to a **stdargs** function are passed in memory.

A call to a procedure, subroutine, or function uses a **bsr** or **jsr** instruction which saves the return address in the system register **r15**. A return uses the **rts** pseudo-instruction.

Return Values

Return values that are up to 32 bits in size are returned in **r2**, sign- or zero-extended to 32 bits for scalar types smaller than 32 bits. Return values that are between 32 and 64-bits in size are passed in the register pair **r2**/**r3**.

In C and C++, to call a function which returns any type not passable in register (e.g. structures), the address of a temporary of the return type is passed by the caller in **r2**. The function returns the structure value by copying the return value to the address pointed to by this register before returning to the caller.

In FORTRAN, to call a **FUNCTION** which returns a **COMPLEX**, **DOUBLE COMPLEX**, or **CHARACTER** value, the address of a temporary of the return type is passed in **r2**. If a **FUNCTION** returns a **CHARACTER** value, then the size of the temporary, in bytes, is passed in **r3**. The subroutine or function returns the value by copying he return value to the address pointed to by **r2** on entry.

Frame Pointer

If **-ga** is specified on the command line, a frame pointer will be set up in **r14** for use by a symbolic debugger. This option is required for runtime error checking and graph profiling. This is not part of the MCore ABI.

Accesses to parameters or local stack storage are always made relative to the stack pointer, **r0**, even if a frame pointer is set up.

Interrupt Processing in C and C++

Interrupt functions on the MCore use different calling conventions than normal functions. Specifically:

- Interrupt functions must return using the `rte` instruction rather than the `rts` instruction, and
- Normal functions are permitted to destroy the contents of certain temp registers. If the caller wishes to save the contents of these registers, it must be done by the caller before calling the function. Interrupt functions are not permitted to destroy the contents of those registers.

You can make the compiler follow these conventions for a particular function in C or C++ by inserting the `#pragma ghs interrupt` instruction immediately after the opening curly brace.

Alternatively, the keyword **__interrupt** may be placed at the beginning of a function definition:

```
__interrupt void func(void)
```

Non-interrupt routines only save and restore permanent registers used, but an interrupt routine also saves and restores any temporary registers if they are used. If an interrupt routine has a function call, then all temporary and permanent registers will be saved, even if they are not used in the interrupt function.

5

Embedded Features

This chapter contains:

- Program Sections
- Putting Data into ROM
- Reducing Program Size
- Using Linker Switches
- Producing S-Record Output
- Multiple-Section Programs
- Renaming Text Sections
- Japanese Automotive C

Japanese Automotive C Symbolic memory-mapped I/O Embedded developers have many special requirements for controlling how data and code are arranged and accessed.

Program Sections

Program sections are labeled collections of program objects. The simplest program sections are **.text**, **.data**, **.rodata**, and **.bss**. The **.text** section holds program code. The **.data** section holds external variables with explicitly initialized values such as **int i=1;**. The **.rodata** section holds compiler generated constants and read-only variables. The **.bss** section holds variables which are not explicitly initialized. In most systems, the runtime or operating system initializes the **.bss** section to all zeros. This zero initialization is required by the C and C++ languages.

The compiler assigns various data and text objects to the appropriate sections at compile-time. The linker's job is to collect all data for each named section and to locate that section in memory. In doing this, it is guided by a user-supplied or default linker section map. The section map specifies the desired location of each section and also the order of the sections in the final output file.

In addition to the above sections, you can create sections and assign them to specific regions of memory. You can also assign variables to the user-defined sections. You have the flexibility to position variables and other program objects in memory.

The linker provides definitions for several symbols which, if referenced and not defined, are given certain addresses corresponding to the final image. These symbols are constructed by prepending the strings **_ _ghsbegin** and **_ _ghsend** to the name of each section in the final image, with any period (.) in the section names changed to underscores (_). For example, for a section named **.text**, the symbols **_ _ghsbegin_text** and **_ _ghsend_text** would revert to the virtual start and end addresses of that section, respectively.

Putting Data into ROM

The embedded features package includes facilities for various data items to be put into ROM. The assumption is that the program text is in ROM, so these features are designed to put specified data items with the program text.

Putting Initialized Data into ROM

A program located in ROM may need to initialize RAM memory upon power-up or restart. If the program needs to have RAM variables that are initialized to specified values, proper steps must be taken to set up those variables at startup. Normally, executable files created by the linker consist of instructions in **.text**, initialized data in **.data**, and zero-initialized data in **.bss**.

Since all memory variables are in sections, they are initialized one entire section at a time. There are three ways that sections are initialized, depending on the type of section. Read-only sections have all of their initial values in the executable file. The contents of this section are downloaded to the target via the debug server or the contents are burned into ROM. Read-write sections are initialized by creating a read-only section which is a copy of the read-write section. This read-only section is initialized as described above. Then, during program startup, the contents are copied from the read-only image to the read-write section in RAM.

Zero-initialized sections (**.bss** sections) are initialized to zero during program startup. All variables in these sections have been implicitly initialized to zero.

The new ROM linker directives files make these actions largely automatic. By using the Green Hills startup code, a program can be put into ROM, and still have initialized variables (whose values are automatically copied from ROM to RAM at program startup time). In addition, its bss sections are automatically cleared, so that **.bss** variables start proper initialization to zero. “Linker Directives Files” on page 207 provides more information.

How to Copy Data Sections from ROM to RAM and Clear .bss (zero-initialized data)

The Green Hills startup code automatically clears **.bss** sections and copies ROM to RAM; however, you can customize this, using the following pointers:

_ _ghsbinfo_clear

_ _ghseinfo_clear

_ _ghsbinfo_copy

_ _ghseinfo_copy

These pointers reference the **.secinfo** section which contains addresses specifying the areas of memory that need to be copied or cleared. For further information on customizing, on copying ROM to RAM, and to see the code that does the copy or clear, refer to “ind_crt0.c” on page 213.

Verifying Program Integrity

One use of the **_ _psinfo** structure is to checksum sections in memory to verify that they have not changed from the state in which they were created by the linker. For this purpose, the linker calculates a CRC checksum for each **F_TEXT** or **F_DATA** section of non-zero length, and stores it as the last four bytes of the section. Upon initialization, it is possible to scan the **_ _psinfo** table sections of this type, calculate the same CRC on all but the last four bytes, and then compare the result to the stored CRC. A match indicates that the program has the same byte values in memory that it had when the linker created the executable file.

Reducing Program Size

One concern of developers of embedded systems is program size. Given the limitations on the amount of ROM available in typical embedded systems, it is desirable to make programs as small as possible. The Green Hills compiler allows you to create smaller executable files under certain conditions.

Removing Floating-Point Libraries

Large executable files can be the result of library routines which cause code related to floating point operations to be linked in. This can occur even if your program does not use floating point variables.

A prime example of this is the **printf** function. Since **printf** allows various formatting options for floating point values, which are the **%f**, **%e**, and **%g** switches, using **printf** causes floating point handling code to be loaded even if these particular options are not used. This may add considerable size to the executable program.

The **-fnone** switch to the driver is designed to avoid this problem. This switch means that the user program is not using floating point operations. This allows the driver to load special versions of **printf** and other library functions that do not use any floating point code.

The **-fnone** switch has two effects. The compiler will give a fatal error for any floating point constant and for any use of the reserved words **float** and **double**. This prevents any floating point value or operation from appearing in the C source code. At link time, the linker searches a special library which has non-floating point versions of library functions before searching the regular libraries. If any of these functions are used, the non-floating point versions are loaded in place of the floating point ones.

The compiler attempts to position certain program variables in order to minimize the space for padding between variables. Global and external variables are not eligible for this optimization because of the possibility that other modules may make assumptions about their order. Such variables are generally allocated in the order in which they are declared in the source file. However, static variables (both local to functions and of file scope) may be rearranged by the compiler in order to reduce padding space.

The compiler classifies these variables into three categories based on size: single byte variables, larger variables (which are the size of an integer register or less), and variables larger than an integer register. The variables from the first category are allocated, then the second, and the third. Collecting variables of similar sizes together reduces the need for padding.

You can increase opportunities for this optimization by liberal use of static variables wherever possible.

Specifying Program Start Address

The linker normally uses the address of the global symbol **-start** as the start address for the user program. The driver supports an option, **-entry=***sym*, to specify an alternate start address.

For example, use the symbol **newstart** as the program start address with a command line:

```
% ccmcore -entry=newstart file.c
```

Using Linker Switches

The linker, **elxr**, has many switches for embedded system development. When using the driver, linker-specific switches must be preceded by **-lnk=** to be effective. This causes the switches to be passed to the linker unchanged. Please see “Command Line Options” on page 162 for more information.

Producing S-Record Output

The utility program **gsrec** creates Motorola S-record format files from ELF executable files. See Chapter 14, “Utility Programs” for more information.

Multiple-Section Programs

In embedded programming, it is sometimes necessary to place certain variables in specific memory regions. For example, there may be different kinds of RAM memory available, some fast and some slow, and selected variables need to be placed into the fast RAM. The Green Hills C compiler and linker allow you to achieve this and similar goals by grouping variables into program sections and positioning them as desired in memory.

To group program variables and position them in memory, it is necessary to assign a named section to each desired memory region using a linker section map. In addition, information is provided in the program source files to show which variables go into which sections. This is done using the **section** pragma, with the following syntax:

#pragma ghs section[*secttype*="*sectname*"[,*secttype*="*sectname*"]...]
The square brackets enclose optional material, and the **...** indicates that the preceding square-bracketed material repeats zero or more times.

sectname is the user-defined section name, eight letters or less in length, and by convention starts with a period (“**.**”). The word **default** may be used in place of any *sectname*. While normal section names are specified in quotes, the word **default** is not.

secttype tells which kind of data item is affected by the pragma, and may be one of the following:

text Program text.

data Initialized variables.

bss Zero-initialized variables.

rodataConstant variables and/or strings.

Each occurrence of the section pragma specifies a mapping of data types to section names. Each section pragma leaves mappings from earlier pragmas in place except for those which it explicitly overrides. Specifying **default** in place of a quoted section name removes any mapping for that particular *secttype*. The statement removes all mappings and restores the section-assignment rules to their initial state:

#pragma ghs section
Mappings affect variables at the point where they are defined. Each variable’s placement to its section is determined by the mapping in the source file. This places different variables to different sections by interspersing section pragmas among the variable declarations. For each variable, the compiler determines which section it would normally fall into and then checks whether variables of

that type have a mapping. If so, the section specified in the mapping is used in place of the default.

For example, consider the following line of C code:

```
int foo=3;
```

In this example the variable **foo** is normally placed into the **.data** section because it is initialized to an explicit value. However, if this line of code were preceded by the following, then the variable **foo** is placed in the section **.mydata** instead:

```
#pragma ghs section data=".mydata"
```

Here is how three different variables might be assigned to three different sections:

```
#pragma ghs section data=".data1"
int x1 = 0;
      /* Assign x1 to section .data1 */
#pragma ghs section data=".data2"
int x2 = 0;
      /* Assign x2 to section .data2 */
#pragma ghs section data=".data3"
int x3 = 0;
      /* Assign x3 to section .data3 */
#pragma ghs section data=default
                  /* Now we are back to default rules */
```

This allocates variable **x1** to section **.data1**, **x2** to **.data2**, and **x3** to **.data3**.

Renaming Text Sections

You can rename a text section, but there are some restrictions. You can rename a text section only once per program source file, before the very first function in the source file.

For example, the following source file places the function `foo()` into the text section **.mytext**:

```
#pragma ghs section text=".mytext"
void foo(void)
{
}
```

The following examples show incorrect usage:

Bad Example 1

```
#pragma ghs section text=".mytext"
void foo(void)
{
}
```

```
        #pragma ghs section text=".mytext2"
/* wrong: can only have one pragma to rename */
/* text section */
        void bar(void)
        {
        }
```

Bad Example 2

```
void foo(void)

        {
        }
        #pragma ghs section text=".mytext"
/* wrong: must use pragma before the first function */
void bar(void)
        {
        }
```

Japanese Automotive C

Japanese Automotive C is a set of extensions to ANSI C used by Japanese automobile manufacturers. For complete specifications, refer to the *C-Language Specification for Automotive Control (Proposal)* by Toyota Motor Corp., July 29, 1993.

Japanese Automotive C generally conforms to the principles of ISO 9899, equivalent to the ANSI X3.159-1989 standard, with the exception of the "Implementation-defined Behavior" specification of Annex G.3 in ISO 9899. Japanese Automotive C modifies, or extends, this specification to support portability. The method by which it extends the "Implementation-defined Behavior" conforms to the "Common Extension" section of ISO 9899, found in Annex G.5.

To select this version of C, click the **Japanese Automotive C** box in the C options window of the MULTI Builder window. Alternately, enter the **-japanese_automotive_c** command line option.

Selecting Japanese Automotive C enables the following command line options:

-pragma_asm_inline

Enables **#pragma asm**, **#pragma endasm**, **#pragma inline**.

-unsignedchar

Specifies type **char** as unsigned.

-unsignedfield

Specifies that a bit field whose type is **char**, **short**, **int**, or **long** has an unsigned value.

-noshortenum

Specifies that enumerated types are integers.

-asmwarn Prints a warning for each _ _asm() statement.

-noasm Prevents the compiler from recognizing **asm** as a keyword in other modes, allowing a variable or function named **asm** to be declared.

Normally, in strict ANSI C mode, it is a fatal error to declare a bit field with basetype other than **int**, **signed int**, or **unsigned int**. Japanese Automotive C makes this legal, even though it is a minor violation of the ANSI standard.

For example,

```
struct {
   char b:3;
   char c:5;
) s;
```

When the above code is compiled with **-ANSI**, the following error occurs:

```
"x.c", line 3:  Illegal type for bit field
"x.c", line 4:  Illegal type for bit field
```

When the code is compiled with **-ANSI -japanese_automotive_c**, no error or warning occurs.

Interrupt Functions

A function may be declared to be an interrupt function by prepending the **__interrupt** keyword to the function definition. The compiler will generate code for this function that will save all the registers this function uses, including the registers that are normally destroyable across function calls. These functions are intended to be used to handle hardware interrupt and exception conditions; since these events are not part of the normal program flow using a non-interrupt function may modify registers, resulting in incorrect behavior of the interrupted routine. These functions should be of void type and should take no arguments.

Example:

```
__interrupt void handle_clock_interrupt(void)
{
   clock_ticks = clock_ticks + 1;
}
```

#pragma ghs interrupt

Putting **#pragma ghs interrupt** before a function definition is equivalent to declaring the function with the **__interrupt** keyword.

#pragma intvect

For all CPU processors, selecting Japanese Automotive C also enables:

#pragma intvect *function integer_constant*

The purpose of this pragma is to establish interrupt vectors. The compiler arranges for the address of the named *function* to be placed in memory at the address specified by **integer_constant** using a **.org** directive to the assembler.

This feature is only supported when using a Green Hills assembler and is not available in binary code generation mode.

The compiler does not check to see if *function* has been declared in the file, or if it is a legal function of any kind. The compiler also does not verify that *integer_constant* is a legal or unique address.

Selecting Japanese Automotive C also results in the following caveat: in the case of a pointer being cast to an integer, if the pointer and the integer are the same size, no data is lost. If the pointer is cast to a smaller integer, then the data is reduced from the upper bit.

Also, for some CPU processors, selecting Japanese Automotive C enables several built-in functions to control interrupts:

void __DI(*void*);

Disables all interrupts.

void __EI(*void*);

Enables all interrupts.

void _set_il(*int n*);

Sets interrupt level to *n*.

6

Debug Formatting

This chapter contains:

- Basic Debug Formatting Information
- Benefits of .dbo Files
- Backwards Compatibility
- Controlling Generation of the .dnm File

Green Hills tools support a proprietary debug format, called **.dbo** files. These files are generated by the compiler or assembler and have the same name as the object file, but with the suffix changed from **.o** to **.dbo**.

Basic Debug Formatting Information

To debug a program with MULTI or a debugger from another vendor, the compiler or assembler must generate information indicating source line numbers and variable data types. The options **-g** or **-G** can be passed to the compiler to generate this debug information. These options are also available in the MULTI Builder's **File Options** window, labeled **Debug Level**. The selection **Plain** corresponds to **-g**, and the selection **MULTI** corresponds to **-G**.

Before a program can be debugged with MULTI, the **dblink** utility program collects the information in the **.dbo** files into a **.dnm** and **.dla** file. The Builder or Driver usually invokes **dblink** just after the program is linked. However, if MULTI is debugging a program **prog**, MULTI looks for **prog.dnm** in the same directory. If MULTI doesn't find **prog.dnm**, it invokes **dblink** to create the file.

dblink uses the symbol table information in the executable program to find all of the **.dbo** files. If the **.dbo** files have been moved, **dblink** might not be able to locate them. In this case, the option may be passed to **dblink** to indicate additional directories where the **.dbo** files may be found:

-dbopath=dir[;dir][;dir][...]

This option is rarely needed and should only be used if **dblink** indicates that **.dbo** files are not found.

Benefits of .dbo Files

The information contained in the **.dbo** file is more extensive than that contained in any previous debug format supported by MULTI. Because debug information is not contained in the object files, link time is greatly decreased without increasing the **dblink** time substantially.

Backwards Compatibility

In the new mode, debug information is only contained in the **.dbo** files and in the **.dnm** and **.dla** file. There is no debug information in the executable program. Therefore, any utilities which depend on reading debug information

will not work. Also, the default behavior will no longer work with debuggers provided by other vendors.

How to Use DWARF

Users can ask the compilers to generate both **.dbo** files and DWARF debug information. This is recommended for users who have utilities which read the debug information in the object files as well as for users who use both the MULTI Debugger along with other source debuggers. To generate both **.dbo** files and DWARF debug information in an ELF environment, select **Output dual debug formats** in the **Advanced Options** window, or enter **-dual_debug** when using the command line driver.

By default, only the **.dbo** file is generated. However, if **-dual_debug** is specified, both DWARF and **.dbo** files are generated. This mode is provided for compatibility with third party tools that read DWARF debug information. Even if **-dwarf** is specified, only the **.dbo** files are actually used by MULTI to debug the program. Use of the **-dwarf** option will not improve debugging in any way and will slow down assembly and link-time by increasing the size of assembly and object files significantly. See Debugging Options in the Compiler Driver Options chapter, for more information.

Controlling Generation of the .dnm File

By default, **dblink** is invoked after the program is linked if **Debug Level** is **Plain** or **MULTI** and **.dbo** files are in use, or if **Debug Level** is **MULTI** and **.dbo** files are not in use.

Three command line options control the invocation of **dblink** by the driver:

-nodnm
: Prevents the invocation of **dblink** after linking the program.

-nonodnm
: Forces the invocation of **dblink** after linking the program.

-dnm
: Invokes **dblink** on an executable that has already been linked.

Two additional command line options control whether the executable has its symbols stripped after **dblink** is invoked. The executable is never stripped unless the Builder links the program and invokes **dblink**:

-strip

Forces the program to be stripped (by the **gstrip** utility) after **dblink** is run.

-nostrip

Prevents the program from being stripped after **dblink** is run.

By default, the program will only be stripped with **-G -nodbo**.

7

ELF Files

This chapter contains:

- Relocatable and Executable File Organization
- 32-bit ELF Data Types
- ELF Header
- ELF Identification
- Sections
- Symbol Tables
- String Tables
- Program Headers

ELF stands for Executable and Linking Format. This chapter explains the organization of ELF files of all types. Sections of this chapter have been reproduced with permission from UNIX System Laboratories, Inc. For additional information about ELF files, please see *System V Application Binary Interface*, 1993, UNIX System Laboratories, Inc., published by Prentice-Hall, Inc.

An ELF file can be a relocatable object file or an executable file. A relocatable object file holds program code and data and is suitable for linking with other object files. An executable file is a file which holds programs suitable for execution. ELF files are created by the compiler, assembler, and linker.

Relocatable and Executable File Organization

The following two tables show the organization of both types of ELF files, relocatable object files and the executable files:

Relocatable File	Executable File
ELF header	ELF header
Program header table *optional*	Program header table
Section 1	Segment 1
. . .	
Section *n*	Segment 2
. . .	
. . .	. . .
Section header table	Section header table *optional*

An ELF header resides at the beginning of an ELF object file or executable file and serves as the table of contents of the file. All other data and tables in the file may appear in any order. Sections hold the bulk of object file information for the linking view, such as instructions, data, symbol table, and relocation information.

An ELF executable file must have a program header table. A relocatable ELF file does not need one. The program header table tells the system how to load the program.

A section header table contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name and section size. Relocatable files to be linked must have a section header table.

32-bit ELF Data Types

The ELF object file format supports 32-bit architectures with 8-bit bytes. It must be modified for 64-bit architectures. Object files represent some control data with a machine-independent format, making it possible to identify object files and interpret their contents. Part of an object file uses the encoding of the target processor, regardless of the machine on which the file was created:

Type Name	Size	Alignment	Purpose
Elf32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

Table 1 **Object File Types**

All data structures that the object file format defines follow the usual size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4. Data also have suitable alignment from the beginning of the file. For example, a structure containing **Elf32_Addr** will be aligned on a 4-byte boundary within the file.

For portability, ELF data structures use no bit fields.

ELF Header

Some object file control structures can grow, because the ELF header contains their actual sizes. If the object file format changes, a program may encounter control structures that are larger or smaller than expected. An ELF header is set by the following C structure declaration:

```
#define EI_NIDENT  16
typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
```

	Elf32_Addr	e_entry;
	Elf32_Off	e_phoff;
	Elf32_Off	e_shoff;
	Elf32_Word	e_flags;
	Elf32_Half	e_ehsize;
	Elf32_Half	e_phentsize;
	Elf32_Half	e_phnum;
	Elf32_Half	e_shentsize;
	Elf32_Half	e_shnum;
	Elf32_Half	e_shstrndx;
} Elf32_Ehdr;		

e_ident The first bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents.

e_type Identifies the object file type. Possible types are:

Name	Value	Meaning
ET_NONE	0	No file type
ET_REL	1	Relocatable file
ET_EXEC	2	Executable file
ET_LOPROC	0xff00	Processor-specific
ET_HIPROC	0xffff	Processor-specific

e_machine Specifies the target architecture for an individual file:

Name	Value	Meaning
EM_NONE	0	e_machine
EM_SPARC	2	Sun SPARC
EM_386	3	Intel 80386
EM_68K	4	Motorola 68000
EM_88K	5	Motorola 88000
EM_486	6	Intel 80486
EM_ MIPS	8	MIPS
EM_960	19	Intel 80960
EM_PPC	20	Power PC
EM_V800	36	NEC V800 series
EM_FR20	37	Fujitsu FR20
EM_RH32	38	TRW RH32
EM_MCORE	39	Motorola MCORE

EM_ARM	40	ARM
EM_ALPHA	41	Digital Alpha
EM_SH	42	Hitachi SH
EM_TRICORE	44	Siemens TriCore
EM_MIPS_X	51	MIPS-X
EM_COLDFIRE	52	Motorola ColdFire
EM_MMA	54	Fujitsu MMA

e_version Identifies the object file version.

Name	Value	Meaning
EV_NONE	0	Invalid version
EV_CURRENT	1	Current version

The value 1 signifies the original file format; extensions will create new versions with higher numbers. The value of **EV_CURRENT**, though given as 1 above, will change as necessary to reflect the current version number.

e_entry Gives the virtual address to which the system first transfers control upon starting the process. If the file has no associated entry point, this member holds zero.

e_phoff Program header table's file offset in bytes. If the file has no program header table, it holds zero.

e_shoff Section header table's file offset in bytes. If the file has no section header table, this field holds zero.

e_flags Processor-specific flags associated with the file. Flag names take the form **EF_machine_flag**.

e_ehsizeELF header size in bytes.

e_phentsize Size in bytes of one entry in the file's program header table; all entries are the same size.

e_phnum Number of entries in the program header table. The product of **e_phentsize** and **e_phnum** gives the table's size in bytes. If a file has no program header table, **e_phnum** holds the value zero.

e_shentize Section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.

e_shnum Number of entries in the section header table. The product of **e_shentsize** and **e_shnum** gives the section header table's size in bytes. If a file has no section header table, **e_shnum** holds the value zero.

e_shstrndx Section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value **SHN_UNDEF**.

ELF Identification

ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file's remaining contents.

The initial bytes of an ELF header, and an object file, correspond to the **e_ident** member. The identification indexes are tabulated below:

Index Name	Value	Purpose
EI_MAG0	0	File identification
EI_MAG1	1	File identification
EI_MAG2	2	File identification
EI_MAG3	3	File identification
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_PAD	7	Start of padding bytes
EI_NIDENT	16	Size of **e_ident[]**

These indexes access bytes that hold the following values:

EI_MAG0 to **EI_MAG3**

A file's first four bytes identify the file as ELF:

Name	Value	Position
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

EI_CLASS The next byte, **e_ident[EI_CLASS]**, identifies the file's class, or capacity:

Name	Value	Meaning
ELFCLASSNONE	0	invalid class
ELFCLASS32	1	32-bit objects
ELFCLASS64	2	64-bit objects

The file format is portable among machines of various sizes, without imposing the sizes of the largest machine on the smallest. Class **ELFCLASS32** supports machines with files and virtual address spaces up to 4 Gigabytes. It uses the basic types defined above.

Class **ELFCLASS64** is reserved for 64-bit architectures. Its appearance here shows how the object file may change, but the 64-bit format is otherwise unspecified. Other classes are defined as necessary, with different basic types and sizes for object file data.

EI_DATA Byte **e_ident[EI_DATA]** specifies the data encoding of the processor-specific data in the object file. The encodings are:

Name	Value	Meaning
ELFDATANONE	0	Invalid data encoding
ELFDATA2LSB	1	See below
ELFDATA2MSB	2	See below

Other values are reserved and are assigned to new encodings as necessary.

EI_VERSION

Byte **e_ident[EI_VERSION]** specifies the ELF header version number. Currently, this value must be **EV_CURRENT**, as explained above for **e_version**.

EI_PAD This value marks the beginning of the unused bytes in **e_ident**. These bytes are reserved and set to zero; programs that read object files should ignore them. The value of **EI_PAD** changes if currently unused bytes are given meanings.

Encoding **ELFDATA2LSB** specifies 2's complement values, with the least significant byte occupying the lowest address. Encoding **ELFDATA2MSB** specifies 2's complement values, with the most significant byte occupying the lowest address.

Sections

Section Headers

An object file's section header table lets you locate all the file's sections. The section header table is an array of **Elf32_Shdr** structures. A section header table index is a subscript into this array. The ELF header **e_shoff** gives the byte offset from the beginning of the file to the section header table; **e_shnum** tells how many entries the section header table contains; and **e_shentsize** gives the size in bytes of each entry.

Sections contain all information in an object file except the ELF header, the program header table, and the section header table. Object file sections satisfy several conditions:

- Every section in an object file has exactly one section header describing it. Section headers may exist that do not have an associated section;
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file;
- Sections in a file may not overlap. No byte in a file resides in more than one section;
- An object file may have inactive space. The various headers and the sections might not account for every byte in an object file. The contents of the inactive space are unspecified.

A section header has the following structure:

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

sh_name Specifies the name of the section. Its value is an index into the section header string table section, giving the location of a null-terminated string.

sh_type Categorizes the section's contents and semantics. See SHF_GHS_ABS for more information.

sh_flags Sections support 1-bit flags that describe miscellaneous attributes.

sh_addr If the section appears in the memory image of a process, this structure gives the address at which the section's first byte should reside. Otherwise, the field contains zero.

sh_offset Gives the byte offset from the beginning of the file to the first byte in the section.

sh_size Gives the section's size in bytes. Unless the section type is **SHT_NOBITS**, the section occupies **sh_size** bytes in the file. A section of type **SHT_NOBITS** may have a non-zero size, but it occupies no space in the file.

sh_link Holds a section header table index link, whose interpretation depends on the section type.

sh_info Holds extra information, whose interpretation depends on the section type.

sh_addralign

Some sections have address alignment constraints. For example, if a section holds a doubleword, the system may need to ensure doubleword alignment for the entire section. That is, the value of **sh_addr** must be equal to 0, modulo the value of **sh_addralign**. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

sh_entsize Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, this gives the size in bytes of each entry. This structure contains zero if the section does not hold a table of fixed-sized entries.

Special Section Indexes

Symbol table entries index the section table through the **st_shndx** field. See "Symbol Tables" on page 59.

Name	Value
SHN_UNDEF	0
SHN_COMMON	-14
SHN_ABS	-15

SHN_GHS_SMALLCOMMON	-256

SHN_UNDEF
A meaningless section.

SHN_COMMON
Common, or **.bss** symbols are allocated space in this section.

SHN_ABS
Contents of the section are absolute values; they are not affected by relocation.

SHN_GHS_SMALLCOMMON
Not available for all processors. Similar to **SHN_COMMON**, but for a limited number of small variables.

Section Types

A section header's **sh_type** specifies the section's semantics:

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_REL	9
SHT_RELA	4
SHT_NOBITS	8

SHT_NULL
Marks the section header as inactive. It does not have an associated section. Other members of the section header have undefined values.

SHT_PROGBITS
Holds information defined by the program that created the ELF file, whose format and meaning are determined solely by the program.

SHT_SYMTAB
Holds a symbol table. An object file may have only one section of this type, but this restriction may be relaxed in the future. It provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking.

SHT_STRTAB
Holds a string table. An object file may have multiple string table sections.

SHT_REL
Holds relocation entries without explicit addends, such as type **Elf32_Rel** for the 32-bit class of object files. An object file may have multiple relocation sections.

SHT_RELA
Holds relocation entries with explicit addends, such as type **Elf32_Rela** for the 32-bit class of object files. An object file may have multiple relocation sections.

SHT_NOBITS
Occupies no space in the file but otherwise resembles **SHT_PROGBITS**.

Section Attribute Flags

A section header's **sh_flags** holds 1-bit flags that describe the section's attributes.

Name	Value	Abbreviation
SHF_WRITE	0x1	w=writable
SHF_ALLOC	0x2	a=allocated
SHF_EXECINSTR	0x4	e=executable
SHF_GHS_ABS	0x400	b=bits
SHF_MCORE_NOREAD	0x80000000	N/A

SHF_WRITE
Contains data that should be writable during process execution.

SHF_ALLOC
Occupies memory during process execution. Some control sections do not reside in the memory image of an object file, this attribute is off for those sections.

SHF_EXECINSTR
Contains execution machine instructions.

SHF_GHS_ABS
Indicates that these sections are to have an absolute, non-relocatable address.

SHF_MCORE_NOREAD
Indicates that these sections are not readable, but may be executable.

Two structures in the section header, **sh_link** and **sh_info**, hold special information, depending on section type:

sh_type	sh_link	sh_info
SHT_REL SHT_RELA	The section header index of the associated symbol table	The section header index of the section to which the relocation applies.
SHT_SYMTAB	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL.)
other	SHN_UNDEF	0

Section Names

Section names beginning with a period (.) are reserved from general use by application programs, although application programs may use these sections if their existing meanings are satisfactory. Application programs may use names without the leading period to be certain of avoiding conflicts with predefined section names.

Frequently Used Sections

Some sections hold program and control information. Sections in the following list have predefined meaning, with the indicated types and attributes.

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.rel*name*	SHT_REL	see below
.rela*name*	SHT_RELA	see below
.rodata	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	see below
.symtab	SHT_SYMTAB	see below
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

.bss Holds uninitialized data that contributes to the program's memory image. The system initializes the data with zeros when the program begins to run.

.data Holds initialized data that contributes to the program's memory image.

.rel*name*

.rela*name*Both these sections hold relocation information. If the file has a loadable segment that includes relocation, the sections' attributes include

the **SHF_ALLOC** bit; otherwise, that bit will be off. Conventionally, *name* is supplied by the section to which the relocations apply. A relocation section for **.text** has the name **.rel.text** or **.rela.text**.

.rodata Read-only data area. Similar to the **.data** section, but comprised of constant data.

.shstrtab Holds section names.

.strtab Holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section's attributes include the **SHF_ALLOC** bit; otherwise, that bit will be off.

.symtab Holds a symbol table. If the file has a loadable segment that includes the symbol table, the section's attributes include the **SHF_ALLOC** bit; otherwise, that bit will be off.

.text Holds the text, or executable instructions, of a program.

Some sections are specific to Green Hills ELF, listed on the next page. Target processors may support some or all of these sections.

Name	Type	Attributes
.syscall	SHT_PROGBITS	SHF_EXECINSTR + SHF_ALLOC
.secinfo	SHT_PROGBITS	SHF_ALLOC
.fixaddr	SHT_PROGBITS	SHF_ALLOC
.fixtype	SHT_PROGBITS	SHF_ALLOC
.sdabase	SHT_NULL	SHF_ALLOC
.sdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.zdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.sbss	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.zbss	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.heap	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.stack	SHT_NOBITS	SHF_ALLOC + SHF_WRITE

.syscall A program code section to support the Green Hills system call mechanism.

.secinfo A table created by the linker describing actions to be taken on sections as they are loaded for program execution (sections to be cleared, copied from ROM to RAM, etc.)

.fixaddr

.fixtype Tables created by the compiler for Position Independent Code (PIC) and Position Independent Data (PID) static pointer adjustments to be made when the program is loaded for execution.

.sdabase If not in PID mode, the runtime system initializes the Small Data Area (SDA) base register to be the address of this section.

.sdata

.zdata Small data area, similar to the **.data** section but of limited size and more quickly addressed.

.sbss

.zbss Small data area, similar to be the **.bss** section but of limited size and more quickly addressed.

.heap Section describing the area of memory for dynamic allocations through malloc and related functions.

.stack Section describing the area of memory that the program stack will occupy.

Relocation Types

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. Relocatable files must have information that describes how to modify their section contents, thus allowing executable files to hold the right information for a process' program image. Relocation entries contain this information. In the code below, the second example is preferred:

```
typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
} Elf32_Rel;

typedef struct {
        Elf32_Addr      r_offset;
        Elf32_Word      r_info;
        Elf32_Sword     r_addend;
} Elf32_Rela;
```

r_offset Gives the location at which to apply the relocation action. The value is the byte offset from the beginning of the section to the storage unit affected by the relocation.

r_info Gives both the symbol table index with respect to which the relocation must be made, and the type of relocation to apply information for a process' program image. For example, a call instruction's relocation

entry, holds the symbol table index of the function being called. You may find the following macros helpful when reading from or writing to the **r_info** field:

```
#define ELF32_R_SYM(i)   ((i) >> 8)
#define ELF32_R_TYPE(i)   ((unsigned char) (i))
#define ELF32_R_INFO(s,t)   (((s)<<8) + (unsigned char) (t))
```

r_addend Specifies a constant value used to compute the final value to be stored into the relocatable field.

A relocation section references two other sections: a symbol table and a section to modify. The section header's **sh_info** and **sh_link** specify these relationships. Relocation entries for different object files have slightly different interpretations for **r_offset**:

- In relocatable files, **r_offset** holds a section offset. The relocation section itself describes how to modify another section in the file. Relocation offsets designate a storage unit within the second section.
- In executable files, **r_offset** holds a virtual address. To make these files' relocation entries more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation).

Symbol Tables

The symbol table of an ELF object file holds information to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The contents of the initial entry are specified below. A symbol table entry has the following format:

```
typedef struct {
        Elf32_Word          st_name;
        Elf32_Addr          st_value;
        Elf32_Word          st_size;
        unsigned char       st_info;
        unsigned char       st_other;
        Elf32_Half          st_shndx;
} Elf32_Sym;
```

st_name Holds an index into the object file's symbol string table, which holds the character representations of the symbol names. If the value is non-zero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

st_value Gives the value of the associated symbol. Depending on the context, this may be an absolute value, an address, etc.

st_size Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This holds 0 if the symbol has no size or an unknown size.

st_info Specifies the symbol's binding attributes and type, explained in the symbol binding and symbol type sections below. The values and meanings are defined below:

st_other Currently holds 0 and has no defined meaning.

st_shndx Every symbol table entry is "defined" in relation to some section; this holds the relevant section header table index. Some section indexes indicate special meanings. See "Special Section Indexes" on page 53 for more information.

Symbol Binding

A symbol's binding determines the linkage visibility and behavior.

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOPROC	13
STB_HIPROC	15

STB_LOCAL
Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.

STB_GLOBAL
Global symbols are visible to all object files being combined. One file's definition of a global symbol will satisfy another file's undefined reference to the same global symbol.

STB_WEAK
Weak symbols resemble global symbols, but their definitions have lower precedence.

STB_LOPROC through **STB_HIPROC**

Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Symbol Type

A symbol's type provides a general classification for the associated entity:

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13
STT_HIPROC	15

STT_NOTYPE

This symbol's type is not specified.

STT_OBJECT

The symbol is associated with a data object, such as a variable, and array, and so on.

STT_FUNC

The symbol is associated with a function or other executable.

STT_SECTION

The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have **STB_LOCAL** binding.

STT_FILE

Conventionally, this symbol names the source file associated with the object file. A file system has **STB_LOCAL** binding, its section index is **SHN_ABS**, and it precedes the other **STB_LOCAL** symbols for the file, if it is present.

STB_LOPROC through **STB_HIPROC**

Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Symbol Values

Symbol table entries for different object file types have slightly different interpretations for the **st_value** member:

- In relocatable files, **st_value** holds alignment constraints for a symbol whose section index is **SHN_COMMON.**
- In relocatable files, **st_value** holds a section offset for a defined symbol. That is, **st_value** is an offset from the beginning of the section that **st_shndx** identifies.
- In executable files, **st_value** holds a virtual address. To make these symbols more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different object files, the information allows efficient access by the appropriate programs.

String Tables

String table sections hold null-terminated character sequences or strings. The object file uses these strings to represent symbol and section names. An empty string table section is permitted; its section header's **sh_size** contains zero. Non-zero indexes are invalid for an empty string table. If the string table is not empty, you can reference a string as an index into the string table section. The first byte, which is index zero, holds a null character. Likewise, a string table's last byte holds a null character to ensure null termination for all strings. A string whose index is zero, specifies either no names or a null name, depending on the context.

A section header's **sh_name** holds an index into the section header string table section, as designed by the **e_shstrndx** structure of the ELF header. The following figures show a string table with 25 bytes and the strings associated with various indexes:

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	\0	\0	V	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

The string table indexes are:

Index	String
0	none
1	name
7	Variable
11	able
16	able
24	null string

A string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist, and a single string may be referenced multiple times. Unreferenced strings are also allowed.

Program Headers

An ELF executable file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. An object file segment contains one or more sections. Program headers are defined only for executable files. A file specifies its own program header size with the ELF header's **e_phentsize** and **e_phnum**.

```
typedef struct {
        Elf32_Word      p_type;
        Elf32_Off       p_offset;
        Elf32_Addr      p_vaddr;
        Elf32_Addr      p_paddr;
        Elf32_Word      p_filesz;
        Elf32_Word      p_memsz;
        Elf32_Word      p_flags;
        Elf32_Word      p_align;
} Elf32_Phdr;
```

p_type The kind of segment this array element describes and how to interpret the array element's information.

p_offset Offset from the beginning of the file at which the first byte of the segment resides.

p_vaddr Address at which the first byte of the segment resides in memory.

p_paddr This field is currently unused and is set to zero by the linker.

p_filesz Number of bytes in the file image of the segment; it may be zero.

p_memsz Number of bytes in the memory image of the segment; it may be zero.

p_flags Flags relevant to the segment.

p_align Loadable process segments must have congruent values for **p_vaddr** and **p_offset**, modulo the page size. This structure gives the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, **p_align** should be a positive, integral power of 2, and **p_vaddr** should equal **p_offset**, modulo **p_align**.

8

Compiler Driver Options

This chapter contains:

- MCore-Specific Options
- Driver Options Specific to the Assembler
- Library Options
- Driver Options Specific to the ELXR Linker
- General Options
- Data Allocation Options
- Debugging Options
- Optimization Options
- Run-time Error Checking Options
- Ada Compiler Options
- C Preprocessor Options
- C and C++ Preprocessor Options
- C Compiler Options
- C++ Compiler Options
- FORTRAN Language Compiler Options

The Green Hills compiler supports compiler driver options that are mostly host-independent. Most of the options are case sensitive. The compiler driver generates a warning for any unrecognized option. It processes the options on the command line in the order listed. For any conflicting options, the later option overrides the earlier option.

MCore-Specific Options

-fsingle Specify code generation using hardware single-precision floating point instructions.

-cpu=m200 Specifies code generation for the MCore m200 core processor.

-cpu=m300 Specifies code generation for the MCore m300 core processor.

Driver Options Specific to the Assembler

-asm=*options*

Passes the specified *options* to the assembler.

-list[*=file*]Enables source listing. This option may be used in two ways.

-list Listing is saved to a file with the same name as the source file, but with a **.lst** extension.

-list=*file*Listing is saved to file of the specified name.

If no file is specified, the listing will be written to a file with the same name as the object file being created, but with a **.lst** extension. For example, **foo.s** becomes **foo.lst**.

Library Options

-L*directory* The compiler driver passes this option to the linker to specify the directory to search for libraries. There is no space between the **L** and the *directory.*

-l*name* The compiler driver passes this option to the linker to add a library to the link command. (This option is the lowercase letter “l”.) The variable *name* represents the abbreviated notation for the libraries, which is generally derived by removing the **lib** prefix and the filename extension. For example, **-lm** adds the **libm.a** library to the link command. There is no space between **l** and *name*.

This option must follow the input source files to resolve any undefined symbols and must be ordered to resolve any undefined symbols in the

specified library already defined in another library. When you list multiple **-l** options, the libraries are linked in command line order prepended to the default library list.

-nofloation

Set the driver to link in libnoflt.a, a special version of the ANSI library in which floating point I/O is not required. The I/O routines in this library does not contain instructions for floating point support and are much shorter.

-nostartfiles

Supresses the inclusion of crt0.o from the standard library.

-nostdlib

Do not link in the standard libraries. Do not use the standard startup file.

Driver Options Specific to the ELXR Linker

-entry=*sym*

The address of the symbol *sym* specifies the program's entry point. If *sym* is specified as a dash (**-entry=-**), then no entry point is passed to the linker.

-lnk=[*link_option*]

Passes the specified linker options (see Command Line Options) to the linker command line. The option is passed to the linker in approximately the position as it appears on the driver command line. By putting the filename on the driver command line, you can pass a **.lnk** file to the linker. For example:

```
% ccmcore hello.c hello.lnk
```

If no option is specified after **-lnk=**, then the default option to the linker will be suppressed until the necessary option is provided to the linker. **[default]**

-locatedprogram

Forces the default behavior which is to link object files into an executable program. Alternate options include **-relprog**, **-relobj**, **-archive**, and **-shared**, all of which combine object files using either an archiver or linker but do not produce an executable program. **[default]**

-map=*mapfile* is displayed on the standard output.

-map=*filemapfile* is saved to file of the specified name.

-relobj

Generates a relocatable object file instead of an executable file. The resulting file is suitable for being passed as input to another run of the linker.

-sec {*info***}**

This entire option, including everything between the braces {}, is passed unchanged to the linker.

-srec

Generates a Motorola S-record output file as well as a COFF or ELF executable. See the utility gsrec for more information.

-relprog

Retains relocation information in the output file. The resulting file is suitable for execution, or it may be passed as input to another run of the linker to allow further relocation of the program.

There are three major differences between **-relobj** and **-relprog**:

- **-relobj** does not give errors for undefined symbols, but **-relprog** does.
- **-relobj** does not allocate common variables, but **-relprog** does.
- **-relprog** relocates all references to functions and variables so that the program can be executed.

-sreconly

Generates only a Motorola S-record output file. The output filename is specified with a **-o** *filename* option. If one is not provided, **a.run** will be used.

-w

Suppresses linker warning diagnostics.

General Options

-#

Displays each command line to call the compiler, assembler, linker, etc. for processing the input files, without invoking the tools. Same as -dryrun.

@file

Invokes the driver to read options from the named file, separated by spaces, tabs and newlines. All other characters in the file are literal. Note that linker directive files should be passed to the linker by using a known suffix, such as .lnk or .lx, rather than using the @ syntax.

-archive

Invokes the librarian to generate an archive instead of invoking the linker to generate an executable program. This option must be used with the **-o** *filename* option, and the suffix of the output filename must be **.a**. For example:

```
% ccmcore foo.c -archive -o libfoo.a
```

-c

Generates only a relocatable object file for each source input file with a filename of *filename***.o**.

-dryrun

Displays the command line to call the compiler, assembler, linker, etc. for processing the input files, without invoking them. See also **-v** and **-#**.

-errmax=*n*

Limits the number of error messages the compiler prints before quitting to *n*. The default is 100 and the minimum is two.

-fnone

Emits an error when the source file makes use of the floating point. This option also implies **-nofloatio**. See “Reducing Program Size” on page 34.

-H

Displays a list of files opened by an **#include** directive. The output goes to **stderr** rather than **stdout**. This option corresponds to the following: click on **Options**, click **Advanced...**, select **Show Headers**.

-Help

Displays to the standard error a detailed list of the compiler driver options to the standard output. The compiler driver ignores all other arguments.

-help

Displays a list of most of the compiler driver options to the standard output. The compiler driver ignores all other arguments.

-I-

The **-I-** option effects the way **#include** "file.h" and **#include** <system.h> are handled. For example, if both **#include** directives appear in source.c and **-I**here and **-I**there are both on the command line, then the compiler first searches for file.h in the directory containing source.c. Then, it will search in ’here’, and ’there’. The compiler with search for system.h in ’here’, and ’there’ only.

If the **-I-** option is added to the command line like this:

-Ihere -I- -Ithere

then the compiler will search for file.h in 'here', and 'there', and the compiler will search for system.h in 'there' only.

-I- causes the compiler to search for files specified with "" in all of the directories listed with -Idir on the command line, but not in the directory containing the source file. Furthermore, **-I-** causes the compiler to search for files specified with <> only in the

directories listed with **-Idir** _after_ the **-I-** option on the command line.

Use the **-I** option to avoid a fundamental problem with the **#include** " " directive. In large programming projects, you may make local copies of either source files or header files. If you have a local copy of a header file but do not have a local copy of all source files that include that header file with #include " ", then the local header file will not always be used. To solve this:

- Never use **#include** " ", but always use **#include** *file* instead.
- Use the **-I-** option. Once **-I-** appears anywhere on the command line, the **#include** "" option will never cause the compiler to look in the directory containing the source file. Instead, the compiler will look only in the directories listed on the command line, in the following order:
  1) Local directories
  2) Non-local directories containing both source and headers. 3) **-I-** option
  4) Any directories which may be referenced with the **#include** <>.
- NOTE: It may be necessary to list some directories both before and after the **-I-** option in order to ensure that the proper header file is found in all cases.

-keeptempfiles

Does not delete temporary files after they are used, including the **.s** files. If you want to generate the **.s** file in the current directory and not have it deleted, use the **-S** option. Equivalent to MULTI Builder's "Keep Temp Files" option in the Advanced Options window.

-language=cxx

-language=fortran

Informs the driver of all of the languages which are in use in the program being linked. This option may be specified once for each language. The driver always assumes that at least some files are written in C or assembly language, but it needs to know if any files contain C++ or FORTRAN in order to select the correct libraries and perform any special processing at link time. If C++ or FORTRAN source files appear

on the command line, the driver automatically sets this option. If only object files and libraries are on the command line, this option is required.

-ident=*string*

Passes the arbitrary string, *string*, to the object file. This is the same as using **#pragma ident "***string***"** in C. This can be used to place the date of the source file in the object file.

-o *filename*

Names the output file of the current driver command. In the simplest case, the linker output file will be named *filename*. If another file is generated from the linker output file, such as an S-Record file, the **-o** determines the name of the other file as well. If only one source file is named, **-o** can be used with either the **-S** or **-c** option to name the output of the compiler or assembler. The driver enforces certain suffixes for some types of output files.

-object_dir=*foobar*

foobar is a directory, often a subdirectory of the current working directory. The driver puts object files there, along with assembly listings, debug information files, inliner files, and other intermediate files which have the same basename as the object file but with a different suffix. Note that the output of the linker and the output of the archiver is never put into **object_dir**.

-passsource

When used with **-S**, it interleaves your original source code with the generated assembly code. Not every line of the original source code will appear in the output.

-pg

Generates code to collect extended profiling information for use with MULTI. This option functions similarly to the **-p** option, except that a call graph report can be produced with the additional information.

-prefixed_msgs

Inserts the words WARNING and ERROR before every warning and error message encountered during compilation. The following Utility Programs support this option: **gbincmp**, **gnm**, **gstrip**, **gsrec**, and **mtrans**.

--prelink_objects

Causes the driver to invoke the C++ prelink utility to instantiate templates but not to invoke the linker or archiver. The effect is similar to **-c** in that only object files are created, but no link is performed. The difference is that the object files contain all template instantiations required by this set of object files. Therefore, they can be linked later

without concern for template requirements. **--prelink_objects** should not be used with any options that prevent the linker from running, such as **-E**, **-P**, **-S**, or **-c**, nor with the option **-template=no_auto_instantiation**, which prevents running prelink.

-S

Produces only an assembly file from the source file. For each source language file specified, compile the file into an assembly language output file. The default output filename is *basename*.**s** unless **-o** is used.

-stderr=*file*

Redirects all warning and error messages into the named *file*.

-shared

Produce a shared object instead of an executable.

-syntax

Checks syntax but does not generate code.

-T

Truncates all symbol names to eight characters for compatibility with older UNIX compilers and linkers that require this option. This option may affect debugging symbols longer than eight characters.

-V

Causes various programs to print their copyright banner and version number as they are invoked.

-v

Displays the compiler driver command lines to invoke the compiler, assembler, and/or linker as they are executed. See also -dryrun and -#.

-W*x***,*args***

Passes options *args* to the tool specified by the *x* argument as follows. The following list describes valid values for *x*. The first value in the list is the number zero, not the uppercase letter “O”.

0 All compilers

2 Compiler but not inliner

L Librarian

C C compiler

F FORTRAN compiler

a Assembler

l Linker

S Linker
Like **-Wl**, but places arguments before any files in the link line. See **-lnk=**.

-w

Suppresses compiler warning diagnostics.

-Y*x***,***directory*

Specifies the directory containing the executable designated by the *x* option. Valid options are as follows, with the first argument in the list being the number zero, not the uppercase letter “O”:

0 Contains the compiler executable.

I The default directory to search for **#include** files. The compiler driver will not search the standard Green Hills include directories.

L Specifies the linker’s primary search directory.

S Contains the startup module or modules such as **crt0.o**.

U The secondary default library search directory for the linker.

a Contains the assembler executable.

l Contains the linker executable.

Data Allocation Options

-autoregister

Enables automatic allocation of local variables to registers. This is the default.

-globalreg=*n*

n is a number from 0 to 4. Arguments to **-globalreg=** must be positive decimal integers. If this option is used, then the new, formerly illegal syntax is now valid:

```
register int i; /* file scope implies global */
```

However, **register static int i** is still illegal, because it can cause a possible register numbering problem when compiling multiple source files.

The type of global register variable can be any data type **T**, including **struct** and **union**, subject to the following restrictions:

- sizeof(T) less than or equal to sizeof(register), usually 4 bytes,
- sizeof(T) must be a power of 2 (sizeof(T) = 1, 2, 4, 8),
- no floats or doubles within type T if type T is a struct or union,
- no arrays within type T if type T is a **struct** or **union**,
- the global register variable must not be explicitly initialized,
- the address of a global register variable should never be taken.

For example,

```
register int x;
....
return &x;
....
```

On the MCore, the 4 registers 8, 9, 10, 11 are allocated, in that order, up to n<=4 registers, to hold the new global register variables. The MCore ABI designates all of these as permanent (non-volatile) registers, so they are saved and restored by library routines which use them.

These 4 registers are initialized to zero by the startup module. The program can depend on this, since it is necessary for ANSI C conformance.

You must use the same *n* for **-globalreg=***n* for all compilations which are to be linked together.

It is an error to put a global variable into a register in one compilation but not into the same register in another compilation. You must be consistent across compilations.

If you want to share the same global register variables across compilations, consider placing them into a single include file, which is always included first in all compilations. This ensures absolute consistency of order and association of a variable with a given register, and vice versa.

MULTI works properly with both local (automatic) and global register variables, even if they are structs or unions.

Restrictions:

1. No library callbacks. A routine which is called by a library routine cannot reliably access a global register variable, since the contents of the particular register may have been saved by the library routine and the register used for something else.
2. No interrupt routines. A routine which is called by an interrupt routine may have interrupted a library routine, which leads to the same restriction for the same reason given above.

You can work around these limitations by licensing the complete C runtime library source from Green Hills and recompiling the entire library with **-globalreg=***n*.

-noautoregister

Disables automatic allocation of local variables to registers.

-nooverload

Does not allocate more than one variable to a register or a function.

-overload

Allocates more than one variable to a register. During debugging, this option should be turned off by **-nooverload**. **[default]**

Debugging Options

-G

In default mode (without -nodbg), -G causes the compiler to generate debug information in a .dbg file that corresponds to the object file produced by the compilation. See Chapter 6, “Debug Formatting”, for more information on .dbg files. When using MULTI as the Debugger, it is strongly recommended to use the default -G mode.

When using the **-nodbg** option, **-G** causes the compiler to generate extended debug information as DWARF 1.1 stabs. When using **-nodbg** it is recommended to use **-g** rather than **-G**.

-g

In default mode (without **-nodbg**) **-g** causes the compiler to generate debug information in a **.dbg** file that corresponds to the object file produced by the compilation. See Chapter 6, “Debug Formatting”, for

more information on **.dbg** files. When using MULTI as the Debugger, it is strongly recommended to use the default **-g** mode.

When using the **-nodbg** option, **-g** causes the compiler to generate DWARF 1.1 stabs.

-nodbg

Do not use Green Hills proprietary debug information; instead use DWARF style debugging information with ELF.

-dual_debug

Use both the Green Hills proprietary debug information and the alternate form.

Optimization Options

-O

Enables the general optimizations such as peephole optimization, common subexpression elimination, pipeline instruction scheduling, dead code elimination, tail recursion and static address elimination. Option qualifiers are also available to target specific types or areas of code for improved performance: **-OA**, **-OI**, **-OL**, **-OM**, and **-OS**. When you select any of these suboptions, the compiler automatically enables the basic **-O** optimization. You can specify any combination of optimizations in any order with a single **-O** option. For example, **-OLMA** is equivalent to **-O -OL -OM -OA**. If you use **-O** in conjunction with **-ansi** or **-ANSI**, the compiler driver automatically sets the **-OM** option.

-OE

Allocates more temporary registers to expression evaluation and fewer to other purposes. The -OE option should be used with programs that have intensive numeric calculations:

```
rho=exp(-PI*cft/q)+(2*PI*cft*sqrt(1-0.25/q/q));
```

In statements such as the one above, it is desirable to have enough registers available to hold the result of all intermediate calculations. The **-OE** option is intended to make more registers available for this purpose. If code does not have intensive numeric computations, the temporary registers will not be needed and thus the **-OE** option will have the detrimental effect of reducing the number of registers available for other purposes.

Algorithmic

-OA

Enables algorithmic optimizations. Applies algebraic transformations such as associativity. In some cases, the compiler generates instructions that ignore overflow, underflow, or round-off. Due to the delicate nature of these transformations and the relatively rare opportunity for significant improvement, **-OA** *is not recommended for general use*.

Following is an example of C source code and the equivalent code the compiler produces with the **-OA** option enabled. The following source code:

```
if (i-5 < 0) printf("%d\n", i);
```

with algorithmic optimization becomes:

```
if (i < 5) printf("%d\n", i);
```

An example of when **-OA** may produce an unexpected result would be if the variable **i** is within five of its lowest bound. **i-5** "wraps around" and thus is greater than zero, even though **i** starts out as a negative number, less than five.

Inlining

-OI

Enables automatic inlining for all input source files. The compiler inlines each function which it determines is appropriate for inlining.

This is disabled in C++.

Inlining is also performed when a function is called from a file which does not contain its declaration or when the function call appears before the declaration in the same file. This kind of inter-module inlining requires two passes of the compiler.

For example, when compiling the following source modules:

```
% ccmcore -OI main.c prog1.c prog2.c
```

The compiler is invoked twice for each of the three source modules. First, the compiler processes each source file to produce an inline file designated by a **.inf** extension. Then the compiler processes each source using the original source files and the three **.inf** files.

Functions declared with the **__inline** keyword (in C and Pascal), the **inline** keyword (in C++), and **OPTIONS/INLINE** in FORTRAN are also inlined. This works even if **-OI** is not thrown. There are a few properties to be aware of. First, these functions are assumed to be of static scope; therefore, they are not exported or available to be inlined in other modules. Second, these functions must be defined in the module before they are called or included in other inline function; otherwise the callers located before the function definition will not be able to inline them. Third, these functions are not output in closed form if they were successfully inlined by all their callers.

Since the **__inline** keyword defines a static function, a function declared in a single module with the **__inline** keyword cannot be inlined into multiple modules. However, there are two techniques that can be used to inline a function into multiple modules, even when the compiler chooses not to do so automatically. The first is that the function can be declared with **__inline** near the top of each module where it is used; this process can be made easier by putting the **__inline** function into a file that is included by each source module that needs it. The second is to use the **-OI=***funcname* option, described below, without the use of the **__inline** keyword. Users may also consider using the **-OD=***funcname* option in conjunction with this second technique.

-OI=*routines*

Inlines the functions listed on the command line and any functions specifically marked for inlining. No automatic inlining is done. For example, when compiling the following source code:

```
%    ccmcore -OI=sub main.c prog1.c prog2.c
```

The compiler inlines the routine **sub()** when called, along with calls to any routines specifically marked for inlining.

Example 1:

a.c: int a() { return 1; }

b.c: int b() { return 2; }

c.c: extern int a(), b(); int c() { return a() + b(); }

o **ccmcore -OI a.c b.c c.c**

a() and **b()** will be inlined automatically because of **-OI**.
a() and **b()** will both also be output in their defining modules.

o **ccmcore -OI=a a.c b.c c.c**

a() will be inlined automatically because of **-OI=a**.
b() will not be inlined because **-OI** was not thrown.
Both **a()** and **b()** will be output in their defining modules.

o **ccmcore -OI=a -OD=a a.c b.c c.c**

a() will be inlined automatically because of **-OI=a**
b() will not be inlined because **-OI** was not thrown.
a() will not be output in closed form, but **b()** will.

o **ccmcore -OI=a -OI -OD=a a.c b.c c.c**

a() will be inlined automatically because of **-OI=a**
b() will be inlined automatically because **-OI** was thrown.
a() will not be output in closed form, but **b()** will.

Example 2:

a.c:

```
int a_very_big_function()
{
   /* lots of code in here, not shown for brevity... */
   return 0;
}

int b()
```

```
{
   return 2;
}
```

b.c:

```
extern int a(), b();
__inline int c()
{
   return 3;
}

int d()
{
   return a_very_big_function() + b() + c();
}
```

o **ccmcore -OI a.c b.c**:

> **a_very_big_function()** will not be inlined even though **-OI** is thrown because the compiler deems it to be too large.
>
> **b()** will be output because **-OI** is thrown and it is small.
>
> **c()** will be inlined because it was declared with **__inline** in the same module as the caller.
>
> **a_very_big_function()** and **b()** will be output in closed form in module a. **c()** will not be output in closed form because it declared with **__inline** and it was was successfully inlined everywhere.

o **ccmcore -OI=a_very_big_function a.c b.c**:

> **a_very_big_function()** will be inlined because of **-OI=a_very_big_function**

b() will not be inlined because **-OI** was not thrown

c() will be inlined because it was declared with **__inline** in the same module as the caller.

a_very_big_function() and **b()** will be output in closed form in module

a. **c()** will not be output in closed form.

o **ccmcore a.c b.c**:

a_very_big_function() and **b()** will not be inlined because **-OI** was not thrown.

c() will be inlined because it was declared with **__inline** in the same module as the caller.

a_very_big_function() and **b()** will be output in closed form in module a. **c()** will not be output in closed form.

Loop Optimization

-OL

Enables loop optimizations. With this optimization, the compiler concentrates most of its resources to optimizing code in the innermost loops in your source files. Therefore, this option is most effective for code containing many loop structures which are executed frequently. This optimization includes strength reduction, loop invariant analysis, and loop unrolling. Where code size is a priority, you can disable loop unrolling. Also, you can increase the maximum number of times the loop is unrolled. See the other options in this section.

-OL=*routines*

Performs loop optimization only for the functions listed on the command line. For example, the following command line specifies loop optimization only for the function **sub**:

```
% ccmcore -OL=sub main.c prog1.c prog2.c
```

-OLB

Increases maximum for loop unrolling from 4 to 8 times and applies unrolling to larger loops than usual.

-Ounroll8

Increases maximum for loop unrolling from 4 to 8 times. This option requires the **-OL** option.

Memory

-OM

Enables memory option. This option is equivalent to **-O** except it allows the optimizer to assume that memory locations do not change, except by explicit stores, and are not affected by any external sources.

This compile-time option is not safe in applications where memory is externally affected, such as in device drivers, operating systems, and shared memory. This option is also not safe in a non-virtual memory environment when interrupts are enabled.

This option is enabled automatically when **-O** is used with either **-ansi** or **-ANSI**. The **volatile** keyword explicitly indicates any objects which may change without the compiler's knowledge or control. If for some reason you cannot use the volatile keyword, you can use the **-Onomemory** option, which disables memory optimization while enabling other **-O** optimizations.

Space

-OD

Specifies a list of functions whose code generation may be skipped during the compilation phase, thereby reducing code space. For example:

-OD=*foo,bar*

specifies that the compiler should not generate code for the functions *foo* and *bar*. This can be useful if a postprocessing tool determines that these functions are never called.

The **-OD** option can also be useful with inlining. For example:

-OI=*foo,bar* **-OD=***foo,bar*

specifies that *foo* and *bar* will be inlined where ever they are called and because of that, the actual code generation for the functions may be omitted. If for some reason a function listed could not be inlined at one

of the call sites, the linker will not be able to resolve the missing symbol, resulting in a link-time error.

Example 1:

a.c: int a() { return 1; }
b.c: int b() { return 2; }
c.c: extern int a(), b(); int c() { return a() + b(); }

o **ccmcore -OI a.c b.c c.c**

a() and **b()** will be inlined automatically because of **-OI**.
a() and **b()** will both also be output in their defining modules.

o **ccmcore -OI=a a.c b.c c.c**

a() will be inlined automatically because of **-OI=a**.
b() will not be inlined because **-OI** was not thrown.
Both **a()** and **b()** will be output in their defining modules.

o **ccmcore -OI=a -OD=a a.c b.c c.c**

a() will be inlined automatically because of **-OI=a**
b() will not be inlined because **-OI** was not thrown.
a() will not be output in closed form, but **b()** will.

o **ccmcore -OI=a -OI -OD=a a.c b.c c.c**

a() will be inlined automatically because of **-OI=a**
b() will be inlined automatically because **-OI** was thrown.
a() will not be output in closed form, but **b()** will.

Example 2:

a.c:

```
int a_very_big_function()
{
   /* lots of code in here, not shown for brevity... */
   return 0;
}

int b()
{
   return 2;
}
```

b.c:

```
extern int a(), b();
__inline int c()
{
   return 3;
}

int d()
{
   return a_very_big_function() + b() + c();
}
```

o **ccmcore -OI a.c b.c**:

> **a_very_big_function()** will not be inlined even though **-OI** is thrown because the compiler deems it to be too large.
>
> **b()** will be output because **-OI** is thrown and it is small.

c() will be inlined because it was declared with **__inline** in the same module as the caller.

a_very_big_function() and **b()** will be output in closed form in module a. **c()** will not be output in closed form because it declared with **__inline** and it was was successfully inlined everywhere.

o **ccmcore -OI=a_very_big_function a.c b.c**:

a_very_big_function() will be inlined because of **-OI=a_very_big_function**

b() will not be inlined because **-OI** was not thrown

c() will be inlined because it was declared with **__inline** in the same module as the caller.

a_very_big_function() and **b()** will be output in closed form in module

a. **c()** will not be output in closed form.

o **ccmcore a.c b.c**:

a_very_big_function() and **b()** will not be inlined because **-OI** was not thrown.

c() will be inlined because it was declared with **__inline** in the same module as the caller.

a_very_big_function() and **b()** will be output in closed form in module a. **c()** will not be output in closed form.

-OS

Enables space optimizations. This option invokes all the basic optimizations except those that would increase the code size. The compiler generates smaller code, with potential for decreased performance. **-OS** implies **-Onostrcpy**. If **-OS** is specified, calls to

strcpy() will not be inlined. Inlining **strcpy()** takes more space but can improve program speed. **-OS** disables loop unrolling (**-OL**).

-OT

Disables the -OS option when both -OS and -OT are on the command line and -OS is before -OT.

Optimization Control (-Ono)

The options beginning with **-Ono** disable certain optimizations. Each enables the same general optimizations specified by the **-O** option but turns off a specific optimization enabled by **-O**.

-Onoconstprop

Disables propagation of constant expressions. Constant propagation is the replacement of one or more variables with constants over a portion of a variable's lifetime in which the variable's value is known and does not change.

-Onocse

Disables common subexpression elimination.

-Onomemory

Disables memory optimizations.

-Onominmax

For use with C and C++. Suppresses the optimization generating special code for minimum, maximum, and absolute value expressions of the form:

```
(i < j) ? i : j
(i > j) ? i : j
(i < 0) ? -i : i
```

-Onopeep

Disables peephole optimizations.

-Onopipeline

Disables instruction resequencing for pipelined architectures.

-Onostrcpy

For use with C and C++. Suppresses optimizations generating inline code for **strcpy()** and **strcmp()** with constant arguments.

-Onotailrecursion

Disables tail recursion, a general optimization enabled with the -O option. In a recursive procedure, tail recursion optimization replaces the

procedure call with a branch instruction and eliminates the return statement.

-Onounroll

Disables loop unrolling, a part of the **-OL** optimization. If **-OL** is not specified, **-Onounroll** is the same as **-O**. If **-OL** is specified, all loop optimizations except unrolling are performed.

Run-time Error Checking Options

-check=*options*

Generates various run-time checks. *options* is either a single option or a comma-separated list of options (no spaces allowed) from the following list.

These options are not valid in FORTRAN.

all

Turns on all checks.

assignbound

Checks the value in the range of the type when assigning a value to a variable or field which is a small integral type such as a bit field.

bounds

Checks array bound indices.

memory

Causes the compiler to generate extra code to verify every load or store involving a pointer which may point to memory returned by malloc() and its related functions. This option also causes a special version of those library routines to be linked.

(This option differs from **-check=alloc** because **-check=alloc** only links the special library, but does not cause the compiler to generate extra code.)

The verification is performed by reading the memory location to which the pointer points and comparing the contents to a special value. If the memory contains the special value, a library routine, **__ptrchk()**, is called with the address of the memory location. If that address is not within the set of legal addresses currently allocated to the program by malloc, a runtime error will occur.

This check is effective for detecting use of pointers which have been incremented past an allocated buffer.

nilderef
: Generates an error message for dereferences of null pointers.

switch
: Generates a warning if the case/switch expression does not match any of the case labels. This does not apply when a default case label is used or when the case statement is enclosed in an if-then construct.

usevariable
: Generates a warning message during compilation of a function containing any local variables that are read before being written.

watch
: Allows the MULTI debugger to set up one fast watchpoint. See the watchpoint command in the *MULTI Reference Manual*.

zerodivide
: Generates an error message indicating that a divide by zero occurred and then terminates the program execution.

To turn off any of these options, simply precede the option with **no**. You can also use this to turn on every option *except* the indicated flag. For example, to turn on all checks except **zerodivide**, enter:

```
-check=all,nozerodivide
```

The following table lists output error messages when the run-time error checking is enabled. Run-time error checking requires additional code, which increases the size and reduces the speed of the program, but the convenience of automatic checking can be very valuable during debugging.

Some messages are considered fatal errors and cause the program to exit with the specified status. Other messages are warnings and will not cause the program to terminate. However, these warnings can change the eventual exit status.

Ada Compiler Options

For options specific to Ada, refer to the *Green Hills Ada 95 User's Guide and Compiler Reference*.

C Preprocessor Options

-include *filename*

Include the source code of the indicated file at the beginning of the compilation. This can be used to establish standard macro definitions, etc. The filename is searched for in the directories on the include search list.

-Xincludenever

Ignores all **#include** directives.

-Xincludeonce

If a filename appears in more than one **#include** directive during a single compilation, it skips all of the directives except the first one.

-Xnocpperror

During preprocessing, lines inside of false **#if**, **#elif**, **#ifdef**, **#ifndef** are ignored with the exception that a warning or error is given for lines beginning with # which do not contain legal preprocessor directives. This option suppresses these warnings and errors.

-Xnopragmawarn

Suppresses warnings for errors in #pragma which are recognized by the compiler and are incorrect.

-Xredefine

Suppresses the warning or error which is normally given when two **#define** directives have different values for the same preprocessor symbol.

C and C++ Preprocessor Options

-C

Includes comments in the preprocessor output. The default is to strip comments from the output.

-D*name*

Defines the argument *name* for the preprocessor with a default value of one. This is equivalent to placing the following at the top of the first source file, **#define** *name* **1.**

-D*name*=*string*

Defines the argument *name* for the preprocessor with the value of *string*. This is equivalent to placing the following at the top of the source file, **#define** *name string.* There is no space between **D** and *name*.

-E

Invokes the compiler as a preprocessor and places the preprocessed file output on the standard output. This is useful for debugging preprocessor macros and include files.

-P

Similar to the **-E** option. Invokes the compiler as a preprocessor but writes the output to a file which has the name of the input file with its suffix changed to **.i**.

-U*name*

Undefines the preprocessor symbol *name*. Equivalent to placing **#undef** *name* at the top of the source file. This option removes any predefined compiler symbols.

-U-

Prevents the compiler from defining any symbols. Normally, the compiler defines a set of default symbols automatically. See also **-I**, **-I-**, and **-H** under General Options.

C Compiler Options

-ANSI

Sets the compiler in Strict ANSI mode. Strict ANSI mode is 100% compliant with the ANSI X3.159-1989 standard and does not allow any non-standard constructs.

-ansi

Sets the compiler in Permissive ANSI compatibility mode. This mode supports the language features of the ANSI X3.159-1989 standard, while allowing certain useful, but non-compliant, constructs in an ANSI C framework.

-column=*n*

Sets the length of a line for error messages and warnings to *n* characters. The default line length is 80. The compiler will break up errors and warnings into multiple lines, inserting a new line and tab between words. If the length of a single word (such as a filename or symbol name) exceeds the maximum line length, then that word will appear alone on a line, but will not be truncated to fit on the line. If the length is set to zero, then errors and warnings will not break into separate lines regardless of their length. Line lengths less than 40 are ignored.

-noansi

Equivalent to **-Xt**.

-asmwarn
Prints a warning for every **asm** statement encountered. This is the default.

-noasmwarn
Prevents the compiler from printing warning messages for **asm** statements.

-k+r
Interprets the source code as the C version documented in Kernighan & Ritchie, first edition, and compatible with the portable C compiler (PCC). See also **-noansi**.

-gnu_c
Supports Gnu extensions, such as #import, zero size arrays, compound statements as part of expressions, inline functions and the _ _inline_ _ keyword. This is the default in C. Ignored for C++.

-nognu_c
Does not allow Gnu C extensions. This is the default in C++. Ignored for C++.

-shortenum
Allocates enumerated types to the smallest storage possible.

-noshortenum
Does not allocate enumerated types to the smallest storage possible.**[default]**

-shortwchar
Specifies the size of the type **wchar_t** in ANSI C and C++ as 2 bytes.

-noshortwchar
Specifies the size of type **wchar_t** in ANSI C and C++ as 4 bytes. **[default]**

-signedchar
Specifies type **char** as signed.

-unsignedchar
Specifies type **char** as unsigned. **[default]**

-signedfield
Specifies a bit-field whose type is **signed** and is interpreted as a signed quantity.

-unsignedfield
Specifies all bit-fields as an unsigned quantity. This is the default. If the ANSI type declaration **signed** is used, then that bit-field is signed even if the program is compiled with the **-unsignedfield** option.

-signedwchar
Specifies type **wchar_t** in ANSI C and C++ as signed. **[default]**

-unsignedwchar
Specifies type **wchar_t** in ANSI C and C++ as unsigned.

-signedptr
Specifies pointers and addresses as signed.

-unsignedptr
Specifies pointers and addresses as unsigned. **[default]**

-T
Truncates all symbol names to eight characters for compatibility with older UNIX compilers and linkers.

-tmp=*dir*
Causes temporary files to be placed in the directory specified by *dir* instead of **/tmp**. This is useful if **/tmp** is on a small file system which might run out of disk space during compiles with inlining or template processing. On Win32, the default temporary directory is the current directory. This is also set with the **TMPDIR** environment variable. For example:

```
setenv TMPDIR /usr/tmp
```

-Xa
Selects Permissive ANSI compatibility mode. Equivalent to **-ansi**.

-Xansiopeq
Uses ANSI rules for **++** and ***=** in K&R C. This is turned off by the **-Zansiopeq** option.

-Xc
Selects Strict ANSI compatibility mode. Equivalent to **-ANSI**.

-Xconcatcomments
Allows /* */ as concatenation in K&R C. This option may be turned off with the -Zconcatcomments option.

-Xinitextern
Allows variables declared, with the extern storage class in a function, to accept initial values. In the K+R mode, this normally gives a warning and in ANSI C mode, it normally is illegal.

-Xjapanese_automotive_c
Enables a set of extensions to ANSI C used by Japanese automobile manufacturers. This option implies **-Xpragma_asm_inline**.

-Xneedprototype

Generates a fatal error if a function is referenced or called but no prototype is provided for that function. This is an extension to ANSI C and shows that the prototypes exist for all functions.

-Xwantprototype

Generates a warning if a function is referenced or called but no prototype is provided for that function. This is an extension to ANSI C and shows that the prototypes exist for all functions.

-Xnoalias

Adds noalias keyword to C. This option may be turned off with the -Znoalias option.

-Xnoasm

asm inline directive is not recognized. The **_ _asm** directive is recognized; only the **asm** directive without leading underscores is affected by this switch. This switch is enabled with **-ANSI**. See also -[**no**]**asmwarn**.

-Xnooldfashioned

Does not recognize old-fashioned syntax for initializing variables such as **int i 5,** and for assignment operators such as **=+**, **=-**,and **=***. **-Znooldfashioned** turns off this option. If this option is not enabled when compiling in non-ansi mode, the old-fashioned syntax is accepted with a warning message. When compiling in non-ansi mode or using **-Xnooldfashioned** in K&R mode, old-fashioned initializations give the error:

expected: '=' got: constant

and an equal sign followed by the symbols are recognized as two separate tokens: + - * / % & | ^ << >>. This results in a syntax error for the symbols: + / % | ^ << >> but is correct for the following symbols which are legal unary operators in C: - * &.

For example when not compiling in ANSI mode, **-Xnooldfashioned** is required to correctly compile the following lines since no space appears after the equal sign:

```
int i, *p;
i =-3;
p =&i;
i =*p;
```

By default, this option is disabled.

-Xpragma_asm_inline
Enables the following pragmae: **#pragma asm**, **#pragma endasm**, and **#pragma inline**.

-Xs
Equivalent to **-k+r**.

-Xslashslashcomments
Allows C++ style `//` comments.

-Xt
Selects a mode of ANSI C compatibility similar to AT&T C Issue 5.0 transition mode supporting function prototypes and the new ANSI keywords **signed** and **volatile** in a non-ANSI environment.

-Zansiopeq
Does not use ANSI rules for ++ and *= in K&R C. This option is turned off with the option -Xansiopeq.

-Zconcatcomments
Does not allow /* */ as concatenation in K&R C. This option is turned off with the option -Xconcatcomments.

-Znoalias
Turns off the **-Xnoalias** option.

-Znooldfashioned
Turns off the -Xnooldfashioned option.

Compile-Time Error Checking

These options control various forms of error checking performed during compilation.

-strict=comperr
Generates compile-time errors for division by constant zero, overflow of constant expressions, assignment of a constant to a variable where the constant value is outside the range of the variable, passing a constant to a parameter where the constant value is outside the range of the parameter, and constant array index outside of the array bounds.

-strict=compwarn
Provides warnings for unused variables, wrong pragmae, unknown pragmae, and overflow of constant expressions.

-strict
Same as -strict=compwarn plus -strict=comperr. Also performs parameter count checking.

-STRICT=COMPERR

Same as **-strict=comperr**, plus gives an error for any use of a function without an ANSI C prototype.

STRICT=COMPWARN

Same as **-strict=compwarn**, plus gives a warning for any use of a function without an ANSI C prototype.

-STRICT

Same as **-strict**, plus gives an error for any use of a function without an ANSI C prototype. Also performs parameter count checking.

-strict=nocompwarn

Turns off **-strict=compwarn**.

-STRICT=noCOMPWARN

Turns off **-STRICT=COMPWARN**.

-strict=nocomperr

Turns off **-strict=comperr**.

-STRICT=noCOMPERR

Turns off **-STRICT=COMPERR**.

C++ Compiler Options

--alternative_tokens

--no_alternative_tokens

Enables or disables recognition of alternative tokens. These are tokens that make it possible to write C++ without the use of the *{*, **}**, **[**, **]**, **#**, **&**, |, ^, and **~** characters. The alternative tokens include the operator keywords (such as **and**, **bitand**, etc.) and digraphs. Default is **--no_alternative_tokens**.

--array_new_and_delete

--no_array_new_and_delete

Enables or disables support for array, new, and delete. Default is --array_new_and_delete.

-asmwarn

Prints a warning for every **asm** statement encountered. **[default]**

-noasmwarn

Prevents the compiler from printing warning messages for **asm** statements.

--bool

--no_bool

Enables or disables recognition of **bool**. When **bool** is enabled, the preprocessor symbol **_BOOL** is defined, allowing code to determine when a typedef should be used to define the **bool** type. Default is **--bool**.

-dotciscxx

Interprets all files ending with **.c** as proper C++ source files.

--early_tiebreaker

--late_tiebreaker

Selects the way that tie breakers, or cv-qualifier differences, apply in overload resolution. In early tie breaker processing, the tie breakers are considered at the same time as other measures of the goodness of the match of an argument value and the corresponding parameter type. In late tie breaker processing, tie breakers are ignored during the initial comparison, and are considered only if two functions are otherwise equally good on all arguments; the tie breakers can then be used to choose one function over another. The default is --early_tiebreaker.

--enum_overloading

--no_enum_overloading

Enables or disables support for using operator functions to overload built-in operations on enum-typed operands. The default is --enum_overloading.

--exceptions

--no_exceptions

Enables or disables support for exception handling. Default is **--no_exceptions**.

--explicit

--no_explicit

Enables or disables support for the **explicit** specifier on constructor declarations. Default is **--explicit**.

--extern_inline

--no_extern_inline

Enables or disables support for **inline** functions with external linkage. When **inline** functions are allowed to have external linkage as required by the standard, then extern and **inline** are compatible specifiers on a non-member function declaration; the default linkage when **inline** appears alone is external (i.e., **inline** means **extern inline** on non-member functions); and an inline member function takes on the linkage of its class, which is usually external. However, when **inline**

functions have only internal linkage as specified by the ARM, then **extern** and **inline** are incompatible; the default linkage when **inline** appears alone in internal (i.e., **inline** means **static inline** on non-member functions); and **inline** member functions have internal linkage no matter what the linkage of their class. Default is **--extern_inline**.

-gnu_c

Supports Gnu extensions, such as **#import**, zero size arrays, compound statements as part of expressions, inline functions and the **_ _inline_ _** keyword. This is the default in C++.

-nognu_c

Does not allow Gnu C extensions. This is the default in C.

--implicit_extern_c_type_conversion

--no_implicit_extern_c_type_conversion

Enables or disables an extension to permit implicit type conversion in C++ between a pointer to an **extern C** function and a pointer to an **extern C++** function. Default is **--implicit_extern_c_type_conversion**.

--inlining

Enables a reasonable level of function inlining. [default]

--inlining_unless_debug

Enables a reasonable level of function inlining, unless symbolic debug information is requested. If symbolic debug information is requested, then the effect is the same as specifying the **--no_inlining** option.

--keep_gen_c

The C++ source is first translated into a C source file before it is compiled. This file has a **.ic** extension and is normally deleted after compilation. This option causes the **.ic** file to not be deleted.

--long_lifetime_temps

--short_lifetime_temps

Selects the lifetime for temporaries. Short lifetimes refer to the end of full expression. Long lifetimes refer to the earliest of end of scope, end of switch clause, or next label. Short lifetimes are standard C++, and long lifetimes are what cfront uses; the cfront compatibility modes select long lifetimes by default.

--max_inlining

Enables an aggressive level of function inlining.

--max_inlining_unless_debug

Enables an aggressive level of function inlining, unless symbolic debug information is requested. If symbolic debug information is requested, then the effect is the same as specifying the **--no_inlining option**.

--multibyte_chars

--no_multibyte_chars

Enables or disables processing for multibyte character sequences in comments, string literals, and character constants. Multibyte encodings are used for character sets like the Japanese Shift-JIS. Default is **--no_multibyte_chars**.

--namespaces

--no_namespaces

Enables or disables support for namespaces. Default is **--namespaces** for C++, **--no_namespaces** for Embedded C++.

--no_forced_zero_initialization

Does not force global, uninitialized variables to be initialized to zero. This may save initialized data section space for embedded programs.

--no_inlining

Disables inlining of function calls. This may be a useful option for debugging C++ code.

--old_for_init

--new_for_init

Controls the scope of a declaration in a **for-init** statement. The old cfront/pre-ANSI compatible rule means the declaration is in the scope to which the **for** statement itself belongs. The new ANSI standard conforming rules, in effect, wrap the entire **for** statement in its own implicitly generated scope. Default is **--old_for_init**.

--pack_alignment=*align*

Sets the default alignment for packing classes and structs to *align*, which must be a power-of-2 value. The specified alignment is the default maximum alignment for nonstatic data members; it can be overridden by a **#pragma pack** directive.

--restrict

--no_restrict

Enables or disables recognition of the **restrict** keyword. Default is **--no_restrict**.

--rtti

--no_rtti

Enables or disables support for Runtime Type Information (RTTI). Features enabled/disabled are **dynamic_cast**, and **typeid**. Default is **--rtti** for C++, **--no_rtti** for Embedded C++.

-shortenum
Allocates enumerated types to the smallest storage possible.

-noshortenum
Does not attempt to allocate enumerated types to the smallest storage possible. **[default]**

-shortwchar
Specifies the size of the type **wchar_t** in ANSI C and C++ as two bytes.

-noshortwchar
Specifies the size of type **wchar_t** in ANSI C and C++ as four bytes. **[default]**

-signedchar
Specifies type **char** as signed.

-unsignedchar
Specifies type **char** as unsigned. **[default]**

-signedfield
Specifies a bit-field whose type, **signed**, is interpreted as a signed quantity.

-unsignedfield
Specifies all bit-fields as an unsigned quantity. **[default]** If the ANSI type declaration **signed** is used, then that bit-field is signed even if the program is compiled with the **-unsignedfield** option.

-signedptr
Specifies pointers and addresses as signed.

-unsignedptr
Specifies pointers and addresses as unsigned. **[default]**

-signedwchar
Specifies type **wchar_t** in ANSI C and C++ as signed. **[default]**

-T
Truncates all symbol names to eight characters for compatibility with older UNIX compilers and linkers.

-tmp=*dir*
Causes temporary files to be placed in the directory specified by *dir* instead of **/tmp**. This is useful if **/tmp** is on a small file system which might run out of disk space during compiles with inlining or template processing. On Win32, the default temporary directory is the current directory. This is also set with the **TMPDIR** environment variable. For example:

```
setenv TMPDIR /usr/tmp
```

-unsignedwchar
Specifies type **wchar_t** in ANSI C and C++ as unsigned.

--using_std
--no_using_std
Enables or disables implicit use of the std namespace when standard header files are included. Default is **--no_using_std**.

--wchar_t_keyword
--no_wchar_t_keyword
Enables or disables recognition of **wchar_t** as a keyword. Default is **--wchar_t_keyword**.

-Xnoasm
asm inline directive is not recognized. The **_ _asm** directive is recognized; only the **asm** directive without leading underscores is affected by this switch. This switch is enabled with **-ANSI**. See also -[**no**]**asmwarn**.

-Xwantprototype
Generates a warning if a function is referenced or called but no prototype is provided for that function. This is an extension to ANSI C and shows that the prototypes exist for all functions.

-Xneedprototype
Generates a fatal error if a function is referenced or called but no prototype is provided for that function. This is an extension to ANSI C and shows that the prototypes exist for all functions.

C++ Compatibility Options

--anachronisms
--no_anachronisms
Enables or disables anachronisms. The default is **--no_anachronisms**.

--cfront_2.1
-2.1
Enables compilation of C++ with compatibility with cfront version 2.1. This causes the compiler to accept language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront) release 2.1.

--cfront_3.0

-3.0

Enables compilation of C++ with compatibility with cfront version 3.0. This causes the compiler to accept language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (cfront) release 3.0. This option also enables acceptance of anachronisms.

--eel

--eele

Enables extended Embedded C++. Adds templates, namespaces, mutable, new style casts enabled and the Standard Template Library (STL) to Embedded C++.

--old_for_init

--new_for_init

Controls the scope of a declaration in a **for-init** statement. The old cfront/pre-ANSI compatible rule means the declaration is in the scope to which the **for** statement itself belongs. The new ANSI standard conforming rules, in effect, wrap the entire **for** statement in its own implicitly generated scope. Default is **--new_for_init**.

--strict_warnings

--std

Enables strict Standard mode, which provides diagnostic messages when non-Standard features are used, and disables features that conflict with Standard C++. Standard violations will be issued as warnings.

--strict

--STD

Enables strict Standard mode, which provides diagnostic messages when non-Standard features are used, and disables features that conflict with Standard C++. Standard violations will be issued as errors.

C++ Library Selection Options

--eel

Engages extended Embedded C++ library without exceptions.

--eele

Engages extended Embedded C++ library with exceptions.

--el

Engages Embedded C++ library without exceptions.

--ele

Engages Embedded C++ library with exceptions.

--stdl
Engages Standard C++ library without exceptions.

--stdle
Engages Standard C++ library with exceptions.

Error Message Options

--brief_diagnostics

--no_brief_diagnostics
Enables or disables a mode in which a shorter form of the diagnostic output is used. When enabled, the original source line is not displayed and the error message text is not wrapped when too long to fit on a single line.

--diag_suppress *tags*

--diag_remark *tags*

--diag_warning *tags*

--diag_error *tags*
Overrides the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag, or using an error number.

--display_error_number
Displays the error message number in any diagnostic messages that are generated. The option may be used to determine the error number to be used when overriding the severity of a diagnostic message.

--for_init_diff_warning

--no_for_init_diff_warning
Enables or disables a warning that is issued when programs compiled under the new `for-init` scoping rules would have had a different behavior under the old rules. Default is **--for_init_diff_warning**.

--no_use_before_set_warnings
Suppresses warnings on local automatic variables that are used before their values are set. The compiler's algorithm for detecting such uses is conservative and is likely to miss some cases that an optimizer with sophisticated flow analysis could detect; thus, implementation might suppress the warnings from the compiler when optimization has been requested but permit them when the optimizer is not being run.

--remarks
Issues remarks, which are diagnostic messages milder than warnings.

--no_warnings

Suppresses warnings. Errors are still issued.

--wrap_diagnostics

--no_wrap_diagnostics

Enables or disables a mode in which the error message text is not wrapped when too long to fit on a single line. Default is **--wrap_diagnostics**.

Listing Options

--list *lfile*(For C++ only.) Generates raw listing information in the file *lfile*. This information can generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the compiler. Each line of the listing file begins with a key character that identifies the type of line, as follows:

N Normal line of source; the rest of the line is the text of the line.

X Expanded form of a normal line of source; the rest of the line is the text of the line. This line appears following the **N** line, and only if the line contains non-trivial modifications. Comments are considered trivial modifications; macro expansions, line splices, and trigraphs are considered non-trivial modifications.

S Line of source skipped by a **#if** or the like; the rest of the line is text. The **#else**, **#elif**, or **#endif** that ends a skip is marked with an **N**.

L Indication of a change in source position. The line has a format similar to the **#** line-identifying directive output by cpp, as follows:

L line-number filename key

Where *key* is either **1** for entry into an include file, or **2** for exit from an include file, and omitted otherwise. The first line in the raw listing file is always an **L** line identifying the primary input file. **L** lines are also output for **#line** directives (*key* is omitted). **L** lines indicate the source position of the following source line in the raw listing file.

R, **W**, **E**, or **C**

Indication of a diagnostic. The line has the form:

S filename line-number column-number message-text

where *S* is either **R** for remark, **W** for warning, **E** for error, and **C** for catastrophic error. Errors at the end of file indicate the last line of the primary source file and a column number of zero. Command line errors are catastrophes with an empty filename (**“”**) and a line and column number of zero. Internal errors are catastrophes with position information as usual, and message text beginning with **internal error**. When a diagnostic displays a list, such as all the contending routines when there is ambiguity on an overloaded call, the initial diagnostic line is followed by one or more lines with the same overall format. This format is a code letter, filename, line number, column number, and message text, but the code letter is the lowercase version of the code letter in the initial line. The source position in such lines is the same as that in the corresponding initial line.

--xref *xfile*

Generates cross reference information in the file *xfile*. For each reference to an identifier in the source program, a line of the form:

symbol-id name X file line-num column-num

is written. *X* is either **D** for definition, **d** for declaration (that is, a declaration that is not a definition), **M** for modification, **A** for address taken, **U** for used, **C** for changed (but actually meaning “used and modified in a single operation” such as an increment), **R** for any other kind of reference, or **E** for an error in which the kind of reference is indeterminate. *symbol-id* is a unique decimal number for the symbol. The fields of the above line are separated by tab characters. (For C++ only.)

Precompiled Header Options

--create_pch *file*

If other conditions are satisfied, creates a precompiled header file with the specified name. This option has no effect if the option **--use_pch** or **--pch** appears after it on the command line. (For C++ only.)

--pch

Automatically uses and/or creates a precompiled header file. This option has no effect if the option --use_pch or --create_pch appears after it on the command-line. (For C++ only.)

--pch_dir *dir*

Specifies the directory in which to search and/or creates a precompiled header file. This option may be used with any of the other PCH options. (For C++ only.)

--pch_messages

--no_pch_messages

Enables or disables the message display that a precompiled header file was created or used in the current compilation. Default is **--pch_messages**. (For C++ only.)

--use_pch *file*

Uses a precompiled header file of the specified name as part of the current compilation. This option has no effect if the option **--pch** or **--create_pch** appears after it on the command-line. (For C++ only.)

Template Options

-archive

Just like the **--prelink_objects** option, except an archive is created. The library name must be specified with the **-o** option. (For C++ only.)

--auto_instantiation

--no_auto_instantiation

-template=auto

-template=noauto

Enables or disables automatic instantiation of templates. Default is **--auto_instantiation**. (For C++ only.)

--distinct_template_signatures

--no_distinct_template_signatures

Controls whether the signatures for template functions can match those for non-template functions when the functions appear in different compilation units. Default is **--no_distinct_template_signatures**. (For C++ only.)

--guiding_decls

--no_guiding_decls

Enables or disables recognition of “guiding declarations” of template functions. A guiding declaration is a function declaration that matches an instance of a function template but has no explicit definition (since its definition derives from the function template).

For example:

```
template <class T> void f(T) { ...}
void f(int);
```

When regarded as a guiding declaration, **f(int)** is an instance of the template; otherwise it is an independent function for which a definition

must be supplied. If **-no_guiding_decls** is combined with **--old_specializations**, a specialization of a non-member template function is not recognized. It is treated as a definition of an independent function. Default is **--guiding_decls**. (For C++ only.)

--implicit_include

--no_implicit_include

Enables or disables implicit inclusion of source files as a method of finding definitions of template entities to be instantiated. Default is **--implicit_include**. (For C++ only.)

--implicit_typename

--no_implicit_typename

Enables or disables implicit determination, from context, whether a template parameter dependent name is a **type** or a **nontype**. Default is **--implicit_typename**. (For C++ only.)

--instantiation_dir=*directory*

When --one_instantiation_per_object is used, this option can be used to specify the directory into which the generated object files should be put. The default instantiation directory is ./template_dir.

--nonstd_qualifier_deduction

--no_nonstd_qualifier_deduction

Controls whether nonstandard template argument deduction should be performed in the qualifier portion of a qualified name. With this feature enabled, a template argument for the template parameter **T** can be deduced in contexts like **A<T>::B** or **T::B**. The standard deduction mechanism treats these as nondeduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere. The default is **--no_nonstd_qualifier_deduction**.

--one_instantiation_per_object

Puts each template instantiation in this compilation (function or static data member) in a separate object file. The primary object file (the object file corresponding to the original source file) contains everything else in the compilation, i.e. everything that isn't an instantiation. Having each instantiation in a separate object file is useful and recommended when creating libraries, because it allows you to pull in only the instantiations that are needed, thus reducing code size. This is also essential if two different libraries include some of the same instantiations to avoid multiple defined symbol problems.

--old_specializations

--no_old_specializations

Enables or disables acceptance of old style template specializations; that is, specializations that do not use the **template<>** syntax. Default is **--old_specializations**. (For C++ only.)

--prelink_objects

Causes the driver to invoke the C++ prelink utility to instantiate templates, but not to invoke the linker or archiver. The effect is similar to invoking the driver with **-c** in that only object files are created, but no link is performed. The difference is that the object files contain all template instantiations required by this set of object files. Therefore, they can be linked later without concern for template requirements. This option should not be used with any options that prevent the linker from being run, such as **-E**, **-P**, **-S**, or **-c**, nor with the option **-template=noauto** (or **--no_auto_instantiation**) which prevents prelink from being run. This option is used by the **clearmake** process (For C++ only.).

-t*mode*

Controls instantiation of external template entities. External template entities are external (i.e., non-inline and non-static) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition. (For C++ only.)

all Instantiates all template entities whether or not they are used.

local Instantiates only the template entities that are used in this compilation, and force those entities to be local to this compilation.

none Instantiates no template entities. **[default]**

used Instantiates only the template entities that are used in this compilation.

--typename

--no_typename

Enables or disables recognition of **typename**. Default is **--typename**. (For C++ only.)

Virtual Table Control Options

--force_vtbl

Forces definition of virtual function tables where the heuristic used by the compiler provides no guidance in deciding on definition of virtual function tables. See **--suppress_vtbl**. (For C++ only.)

--suppress_vtbl

Suppresses definition of virtual function tables where the heuristic used by the compiler provides no guidance in deciding on definitions of virtual function tables. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline non-pure virtual function of the class. For classes that contain no such function, the default is to define the virtual function table (but to define it as a local static entity). This option suppresses the definition of the virtual function tables for such classes, and the **--force_vtbl** option forces the definition of the virtual function table for such classes. **--force_vtbl** differs from the default behavior in that it does not force the definition to be local. (For C++ only.)

The following table lists single dash equivalents for double dash options that are not documented in this manual.

Double Dash Options	Single Dash Options
--comments	-C
--compile	-c
--debug	-g
--define_macro *name=string*	-D*name=string*
--dependencies	-Make
--driver_debug	-dryrun
--error_limit num	-errmax=*num*
--include_directory	-I
--*instantiate mode*	-t*mode*
--library_directory *dir*	-L*dir*
--no_code_gen	-syntax
--no_line_commands	-P
--no_warnings	-w
--optimize	-O
--output	-o
--preprocess	-E
--signed_chars	-signed_char
--undefine_macro *name*	-U*name*

Double Dash Options	Single Dash Options
--unsigned_chars	-unsigned_char
--version	-V

FORTRAN Language Compiler Options

-C

Generates code that checks for correct array subscripting at run-time. This option slows program execution.

-bigswitch

Allows large computed **GOTO** statements by forcing the compiler to use a 32-bit offset. The default is a 16-bit offset, which is smaller and faster but fails if any of the labels is too far away.

-nobigswitch

Uses a 16-bit offset which does not allow large computed GOTO statements.[default]

-d_line

Compiles lines starting with **d**, **D**, **x**, or **X**. The default is to treat them as comments. This option enables debugging statements.

-dod

Enables the DoD FORTRAN extensions. This is implied by **-vms**.

-nodod

Disables recognition of the DoD FORTRAN extensions. The DoD extensions are also part of the VMS FORTRAN extensions enabled by **-vms**.

-extend_source

Extends source to interpret columns 1 through 132 instead of 1 through 72.

-i2

Sets the type for **INTEGER** to **INTEGER*2**. The default is **INTEGER*4**.

-i4

Sets the type for **INTEGER** to **INTEGER*4**.**[default]**

-namelist

Enables the IBM and VMS compatible **NAMELIST** extensions in FORTRAN. These extensions are enabled by the *-vms* option, and so this option is only needed when **-novms** is active.

-nonamelist

Disables the IBM and VMS compatible **NAMELIST** extensions in FORTRAN. These extensions are enabled by default by the **-vms** option.

-onetrip

Executes at least one iteration of every DO loop. By default, when the lower bound index of a DO loop is greater than the upper bound index, the compiler does not execute the DO loop, for compatibility with the ANSI FORTRAN-77 standard. This option is required for some older FORTRAN-66 programs to operate properly.

-save

Allocates all local variables to permanent memory, equivalent to coding **SAVE** at the start of every subroutine or function for compatibility with older FORTRAN compilers. With this option, variables retain their values between calls to subroutines or functions. **[default]**

-nosave

Allocates local variables to registers or stack, equivalent to coding **IMPLICIT AUTOMATIC (A-Z)** at the start of every subroutine or function. Programs compiled with this option are compliant with ANSI FORTRAN-77 and in some cases execute more quickly.

-U

Does not convert uppercase user-supplied variables to lowercase. By default, FORTRAN is not case sensitive and all FORTRAN names are converted to lowercase. The compiler and library both assume that this translation is performed. This option accesses variables defined in C as uppercase. However when you use this option, all FORTRAN keywords must be in lowercase, making the compiler incompatible with the ANSI FORTRAN-77 standard.

-u

Makes “undefined” the default data type for undeclared variables, equivalent to coding **IMPLICIT UNDEFINED(A-Z)** at the top of the source file.

-vms

Selects VMS compatibility mode.**[default]**

-novms

Selects UNIX F77 FORTRAN compatibility mode.

Symbols

Numerics

A

B

C

D

E

F

G

H

O

P

R

S

T

U

V

W

X

Y

9

Macro Assembler

This chapter contains:

- Macro Assembler Characteristics
- Command Line Options
- Using the Driver
- Macro Assembler Syntax
- Expressions
- Labels

Macro Assembler Characteristics

The Macro Assembler translates ASCII files containing assembly language instructions into binary files containing relocatable, linkable object code.

The features include:

- UNIX V.4 ELF object files
- Very long identifiers (up to 4096 characters)
- Relocatable object modules
- Optional source and generated code listings
- Macro, repeat block, and conditional assembly directives
- Free form assembly input format
- Full symbolic debugger support

Command Line Options

Note: The recommended way to call the assembler is to use the driver to call the assembler. Do not call the assembler directly.

The syntax for the Macro Assembler is:

asmcore [*options*] [*input_files*]

The Macro Assembler combines each specified ASCII *input_file* and produces a single output object module.

As an option, a listing is written to the standard output file. A number of *options* may be specified except where the default is not to perform the function enabled by the option.

-g Outputs line number debug information. This allows MULTI to debug assembly code. This should only be used for hand-coded assembly language files; compiler-generated assembly language files have their own debug information.

-help Prints a help message.

-I*dir* Searches directory *dir* for files specified in **.include** directives.

-list[,[*pagelength*][,*pagewidth*][=][*file*]
Enables the source listing. A page length and page width can be specified for the source listing. If either value is omitted, the defaults are

used. If only one comma and one value are used (e.g. **-list,80=foo**), that value will be the page length and the default page width is used. The default page length is 64, and the default page width is 132.

If = is specified but *file* is not specified, then the listing is displayed on the standard output. If =*file* is specified, then the listing is written to *file*. If neither = nor *file* are specified, then the listing file is written to a file with a **.lst** extension, replacing the current extension (e.g. **.s**).

For example:

-list Save listing to **.lst** file with default page length and line size.

-list,80 Save listing to **.lst** file with page length of 80 and line size.

-list=trax Save listing to file **trax** with default page length and line size.

-list,,110 Save listing to **.lst** file with default page length and line size of 110.

-list,,110=trax
Save listing to file **trax** with default page length and line size of 110.

-list= Display listing on standard output with default page length and line size.

-nogen Disables source listing of macro expansions.

-o *file* Sets the name of the output module to *file*. The default is the name of the assembly language file with a **.o** extension. For example, **foo.o** is produced for **foo.s**.

-r Prints a listing of symbol names in alphabetical order.

-ref Prints a full cross reference of the symbols in alphabetical order, including the symbol name, type, file and line defined, and file and line of each usage.

-V Prints the Macro Assembler version number to the standard output.

Example:

The following Macro Assembler command line produces an object file **example.o** with DWARF 1.1 line number debugging information from the assembler source file **example.s**:

```
% asmcore -g example.s -o example.o
```

Using the Driver

The driver, **ccmcore**, may be used to invoke the assembler and linker. This has several advantages over direct use of the assembler, including the invocation of the preprocessor on the assembly file, if it ends in **.mco**. This allows you take advantage of preprocessor facilities, such as **#include** and **#define**, which are not normally available to the assembly language programmer. Assembly language files generally have one of two suffixes, either **.s** or **.mco**. The preprocessor will only be invoked on the files ending in **.mco**. The general format to use the driver with an assembly file is:

% ccmcore *options* foo.mco

To pass options to the macro assembler, you must use one of two special driver options. Otherwise, the driver interprets the information, and it is not passed to the assembler:

-asm=*assembler_option*
-Wa, *assembler_options*

Only one option is used with each **-asm**. Multiple **-asm** options can be used on one line. Multiple options can be listed after **-Wa**. They must be separated by commas.

For example:

```
% ccmcore -g -Wa,-ref,-r calc.mco -o calc
```

In this example, the options **-ref** and **-r** are sent to the macro assembler. The preprocessor is invoked on the assembly file **calc.mco**. An executable, **calc**, is produced.

```
% ccmcore -asm=-ref -asm=-r stack.s -o stack
```

In this example, the preprocessor is not called because the assembly file ends in **.s** instead of **.mco**. The options **-ref** and **-r** are passed to the macro assembler. An executable file, **stack**, is produced.

At times, the assembly language output of a high-level language is required, often for perusal, and so that alterations can be made before assembly. To do this, use the driver command with the **-S** option:

% ccmcore -S main.c

This causes the C compiler to compile **main.c** and place the assembly language output in the file **main.s**.

Macro Assembler Syntax

Character Set

The Macro Assembler recognizes the standard ASCII character set, consisting of upper and lowercase letters (A-Z, a-z), digits (0-9), and a group of special characters listed in the table below. The Macro Assembler also recognizes the ASCII control characters signifying carriage return, new line, form feed, vertical tab, and horizontal tab.

Character	Name	Character	Name
'	single quote	"	double quote
(	left parenthesis	)	right parenthesis
	blank	%	percent
-	minus sign	+	plus sign
:	colon	!	exclamation mark
,	comma	\	backslash
.	decimal point	*	asterisk
&	ampersand	_	underscore
~	tilde	|	vertical bar
=	equals sign	^	carat
<	less than	>	greater than
/	slash	$	dollar sign
@	at sign	#	number sign
;	semicolon		

Identifiers

Identifiers, or symbols, are composed of letters, digits, and the special characters: decimal point or underscore.The first character of an identifier must be alphabetic or one of these two special characters. Upper and lowercase letters are distinct; the identifier **abc** is not the same as the identifier **ABC**. Characters in reserved symbols, such as directives, machine instructions, and registers, are also case sensitive.

Examples:

The following table shows some valid and invalid identifiers:

Identifier	Validity
*star	Invalid (may not start with *)
123test	Invalid (may not start with digit)

Identifier	Validity
f-ptr	Invalid (hyphen not allowed)
_hello	Valid
LABEL	Valid
test4	Valid

Reserved Symbols

The following identifiers are part of the fundamental assembly language. These symbols and their meanings are shown in the table below. Identifiers in both upper and lower cases are reserved.

Identifier	Meaning
r0 - r15	integer registers
cr0 - cr31	control registers
psr	status register
gbr	global base register
sp	stack pointer (r15)
vbr	vector base register
epsr, fpsr, epc, fpc	exception shadow registers
sso0-ss4	supervisor storage registers
gcr	global control register
gsr	global status register
pc	program counter

Constants

Macro Assembler constants may be numeric, character, or string constants.

Numeric Constants

A sequence of digits defines a numeric constant. By default, constants are in decimal or floating point format, although this can be changed by a directive in a source file. However, constants can be specified in hexadecimal, octal, binary by preceding the number with one of the special prefixes on the following page:

Type	Prefix	Example
hexadecimal	0x	0xb0b
octal	0	0747
binary	0b	0b110011

All integer constants are assigned 32-bit two’s complement values.

String Constants

Some directives take a string constant as one or more of their arguments. A string constant consists of a sequence of characters enclosed in double quotes ("). A string constant can contain any ASCII character including ASCII null, except newline. The null character is not appended to strings by the Macro Assembler, as it is in C or C++.

Character Constants

A character constant can be used in any location where an integer constant is needed. A character constant consists of the following items in this order: either a single ASCII character or a single quote character (‘), a backslash character (\), one of the value escape characters, and a single quote character (‘). A character constant is considered equivalent to the ASCII value of the character or escape sequence. For example, the character constant **a** is equivalent to the decimal integer 97, and the character constant **\r** is equivalent to the decimal integer 13 (see table on the following page).

Character Escape Sequences

Character and string constants consists of ASCII characters. The ASCII backslash (\) is used within character and string constants to escape the quotation marks and to specify certain control characters symbolically. A backslash followed by any non-escape character is equivalent to that character (e.g. **\a** is equivalent to **a** since **\a** is not an escape sequence).

Escape	Character	ASCII Value
\0	null	0 (0x0)
\b	backspace	8 (0x8)
\t	horizontal tab	9 (0x9)
\n	newline	10 (0xA)
\v	vertical tab	11 (0xB)
\f	formfeed	12 (0xC)
\r	return	13 (0xD)
\nnn	octal value nnn	
\’	single quote	39 (0x27)
\"	double quote	34 (0x22)
\\	backslash in a constant or string	92 (0x5c)

Source Statements

A Macro Assembler source statement consists of a series of fields, delimited by spaces and/or horizontal tab. The general format of a statement is:

[*label:*] *operator arguments*[*comments*]

Label Field

The label field is optional. When specified, it starts in column one and is terminated by the first white space character or line terminator detected. The last character of the label field must be a colon. More than one label may be associated with a given statement line, but a line may consist of no more than one label field.

The label is used to associate a memory location or constant value with the symbolic label name. Labels may have the same names as instructions and directives.

Operator Field

The operator field starts with the first non-white space character after the optional label field and is terminated by the first white space character or line terminator encountered after the operator. An operator is any symbolic opcode, directive, or macro call.

Argument Field

The argument field starts with the first non-white space character following the operator field, and ends with a line terminator or the beginning of a comment field. Arguments are entered to the opcode or directive or are macro call specified.

Comment Field

The comment field is optional and begins with a double slash (//) or a pound sign (#). Ignore all characters to the right of the comment character until the end of the line.

Continuation Lines

The Macro Assembler does not support continuation lines.

White Space

White space consists of spaces, form feeds, and horizontal tabs.

Line Terminators

Assembly input lines are terminated by a semicolon, line feed, form feed, or a carriage return.

Expressions

Assignment Statements

An expression is assigned to a symbol by an assignment statement in one of the following general forms:

symbol =[:] *expr*
symbol .equ expr
.set symbol, expr

The expression specifies any addressing mode, which is generated when the symbol becomes an instruction operand. An assignment by = defines a local constant, and an assignment by **=:** additionally specifies that the symbol is global.

For example:

```
a = 1           # a local constant
xyz =: 123                      # a global constant
```

Both **.equ** and **.set** work similarly to =, with the difference that **.equ** only allows you to assign an expression to a symbol once while **.set** allows you to reassign to the same symbol multiple times.

For example:

```
.set 7 stack            # set stack to be 7 (legal)
.set 8 stack     # reset stack to be 8 (legal)
chair .equ 9     # set chair to be 9 (legal)
sofa  .equ 8     # set sofa to be 8 (legal)
chair .equ 5     # try to set chair to be 5 (illegal)
```

An expression is either absolute or relocatable. See “Expression Types” on page 128 for more information about absolute or relocatable expressions.

Scalar Expression Operators

A number of operators are available to form expressions. The operators are listed below. The type **unary** indicates that the function is recognized when the operator has only a right operand; **binary** indicates that the operator has two operands, one to the left of the operator and one to the right.

Operator	Type	Description
~	unary	BITWISE NOT operator
-	unary	Negation
-	binary	Subtraction
+	binary	Addition
*	binary	Multiplication
/	binary	Division

Operator	Type	Description
%	binary	Modulus
&	binary	BITWISE AND operator
^	binary	BITWISE EXCLUSIVE OR operator
|	binary	BITWISE OR operator
=,==	binary	Equality (0 or 1)
!=	binary	Inequality (0 or 1)
>,>=,<,<=	binary	Signed compares (0 or 1); greater than, greater than or equal, less than, less than or equal
UGT, UGE, ULT, ULE	binary	Unsigned compares (0 or 1); greater than, greater than or equal, less than, less than or equal
<<,>>	binary	Shift left and shift right
USHR	binary	Unsigned shift right (shift 0 into high bit)
ROTR, ROTL	binary	Rotate right and rotate left

The following table lists the operators in decreasing order of precedence.The binary operators are associative from left to right:

Operators
~ and - (unary)
*, /, %, <<, >>, USHR, ROTR, and ROTL
+ and - (binary)
=, .equ, ==, !=, <, >, <=., >=, ULT, UGT, ULE, and UGE
&
| and ^

Expressions are grouped with matching square brackets [ ].

Expression Types

The primary expression types are:

manifest

If the value of an identifier or expression is computed by the Macro Assembler when encountered, it is a **manifest** value.

absolute

If the value of an identifier or expression is computed by the Macro Assembler during assembly, it is **absolute**. Essentially, any absolute value is a manifest value with the exception of an **absolute** value derived from the difference between two relocatable values in the same section.

quoted string

A C-style character string delimited by double quotes is used in conjunction with a number of Macro Assembler directives. All string "escape" sequences defined in the C language, such as **\n** for newline, are allowed. These sequences are described in "Character Escape Sequences" on page 125.

relocatable

A relocatable expression or identifier assigns a value relative to the beginning of a particular section. These values are not determined at assembly time. All label identifiers are relocatable values.

undefined

If an identifier is unassigned, its value cannot be determined until link time. This is an undefined external.

Type Combinations

The constant types are combined with all operators, except where a relocatable type was made immediate or absolute. Constant types and relocatable types are combined only by the following:

+ If one operand is constant, the result is the type of the other operand.

- If the second operand is constant, the result is the type of the other. If both operands are selected from the same one of the types text, data, or bss relocatable, then the result is a constant which is the difference of the addresses.

Examples:

```
4           # constant
4 * (5 + 6)                 # constant
label               # relocatable
```

Labels

A statement optionally begins with one or more labels. Each label is either a named label or a temporary label.

Named Labels

Named labels are identifiers followed by one or two colon characters. Labels defined with one colon are not referenced outside the source module. A second colon specifies that the label is made visibly external to the source file that it is in, instead of being local to that file.

Temporary Labels

Temporary labels consist of a non-zero digit followed by a colon. Any number of these labels can be present, even if the value of the constant repeats. A reference to a temporary label consists of the label's constant value expressed as a decimal number followed immediately, with no space, by an **f** or **b**. This reference refers to the nearest statement with the same numeric label either forward of the reference, not including the current source line, specifying an **f**, or backward from the reference, including the source line, specifying a **b**. Matching labels in the non-specified direction are not referenced, even if they are closer.

Example

```
1:  bra    lb          # infinite loop
    nop                # delay slot
```

10

Macro Assembler Directives

This chapter contains:

- Listing of Macro Assembler Directives
- Characteristics of Specific Directives

Macro Assembler directives and specification of instructions are specified in a similar way. Directives control options of the Macro Assembler or format and generate data for the code segments. Certain directives establish or alter the definitions of symbols.

The Macro Assembler directives are symbols with a type of directive and a predefined value specifying the particular directive. Both directives and instructions appear between labels and operands. The Macro Assembler assigns a specific type to such predefined symbols and searches only for this type between labels and operands. Therefore, labels can have the same names as instructions and directives.

Listing of Macro Assembler Directives

ALIGNMENT:

.align Adjusts location counter to a boundary

DATA INITIALIZATION:

.byte Stores values in successive bytes

.short Store values in successive 16-bit words

.long Stores values in successive 32-bit words

.double Stores values as successive 64-bit IEEE-754 floating point

.float Stores values as successive 32-bit IEEE-754 floating point

.**ascii** Quoted string stored in successive bytes

.**asciz** Quoted string stored in successive bytes (null-terminated)

.space Zero *n* bytes of storage

.fill Produces *n* elements with a given value and size

.literals Forces a dump of all accumulated literals

.ident Stores string in a section called **.comment**

SECTION CONTROL:

.text Specifies text segment

.data Specifies data segment

.**set** Controls instruction reordering

.section Specifies named segment

.previous Undoes the most recent **.section**

.org Specifies absolute segment

SYMBOL DEFINITION:

.export External identifier

.import External identifier

.comm Common block

.bss Local identifier

.weak Weak symbols

.lcomm Local common block

FILE INCLUSION:

.include Includes header file

MACRO DEFINITION:

.macro Defines a macro

.endm Ends macro definition

.exitm Exits from current macro

REPEAT BLOCKS:

.rept Repeats the following statements

.endr Ends repeat block

CONDITIONALS:

.if Enters a conditional block

.else Conditional block alternative

.elseif Alternative plus new conditional test

.endif Ends conditional block

LISTING FORMAT:

.warning Emits warning messages

.nowarning Does not emit warning messages

.list Turns listing on. This is the default.

.nolist Turns listing off

.gen Turns listing of macro generation on. This is the default.

.nogen Turns listing of macro generation off

.eject Puts a form feed (**Control-L**) into the output listing

.title Puts a title at the top of each page in the output listing

.subtitle Puts a subtitle at the top of each page in the output listing

.sbttl Puts a subtitle at the top of each page in the output listing

Characteristics of Specific Directives

Alignment

.align *man-expr, value*

This directive advances the location counter to an addressing boundary specified by the manifest expression (constant expression) *man-expr.* The boundary is 1, 2, 4, or 8 bytes corresponding to a *man-expr* of 0, 1, 2, or 3. The location counter advances to the boundary specified by the manifest expression. As the location counter advances, the segment is filled with zeros or with the given *value*.

Data Initialization

.byte *man-expr***,***man-expr***, ...**

.short *man-expr***,***man-expr***, ...**

.long *man-expr***,***man-expr***, ...**

.double *flt-expr***,***flt-expr***, ...**

.float *flt-expr***,***flt-expr***, ...**

.ascii *quoted-string, quoted-string, ...*

.asciz *quoted-string, quoted-string, ...*

.skip *man-expr*

.space *man-expr*

.offset *man-expr*

.fill *man-expr,man-expr,man-expr*

.ident *q-string*

These directives evaluate expressions and produce successive values of the specified type in the assembly output.

.byte Computes the values of the supplied manifest expressions and produces successive bytes. The manifest expressions must be in the range -128 to 255 (-128 to 127 or 0 to 255).

.short Computes the values of the manifest expressions and produces successive 16-bit words. The manifest expressions must be in the range -32768 to 32767.

.long Computes the values of the expressions and produces successive 32-bit quantities. The expressions are absolute expressions, relocatable expressions or undefined externals. The actual values of relocatable and undefined externals are supplied at link-time. The value of each expression must be in the range -2147483648 to 2147483647 or 0 to 4294967295.

.double Computes the values supplied by the floating point expressions and produces successive 64-bit IEEE-754 floating point values. Each floating point expression must be in double-precision range.

.float Computes the values supplied by the floating point expressions and produces successive 32-bit IEEE-754 floating point values. Each floating point expression must be in single-precision range.

.ascii Evaluates a C-style string enclosed in double quotes (") and produces successive bytes. The delimiting double quote characters and terminating null are discarded.

.asciz Evaluates a C-style string enclosed in double quotes (") and produces successive bytes. Only the delimiting double quote characters are discarded.

.skip *n* Generates *n* bytes of zero data.

.space Alternate name for **.skip**.

.fill *n,s,v* Generates *n* values of size *s* bytes and value *v.*

.literals Cause the accumulated literal table for the **lrw**, **jsri**, and **jmpi:** instructions for the current section to be emitted.

.ident This directive stores the C-style string specified in *q-string* into a section named **.comment** in the object file. The delimiting double-quote characters are eliminated and the string is null terminated. A common use for the **ident** directive is to store version and revision information in the object module.

Section Control

.text

.data

.**set**

.**section** "*name*","*attr*"

.previous

.using

.org *man-expr*

These directives direct assembly output into the specified section.

.text Directs assembly output into the text segment.

.data Directs assembly output into the data segment.

.set This controls whether instruction reordering will be done by the assembler. The MCore architecture contains *hazards*, sequences of instructions which lead to indeterminate results.

.section This directive directs assembly output into the section called *name*. The section name should be followed by a string using one of the listed combinations of the letters **a**, **b**, **x**, and **w**. **a** means that the section should have memory allocated for it, that is, not be used solely for debugging or symbolic information. **b** means the section will have BSS semantics. Although normal data directives, such as **.word** and **.byte** are allowed in a **.bss** section, all of the values specified in those directives are discarded by the assembler. Instead, in the ELF output file, the assembler records only the size of the section. The contents of the section are omitted. When the section is downloaded to the target, space is allocated for the section, but no data is downloaded to this section. Instead, the application is responsible for initializing all bytes in the section to zero. **b** is used by the compiler for uninitialized variables in the Zero Data Area and any uninitialized variables in a renamed section. **x** indicates that the section will contain executable code. **w** indicates that the section will be writable. If none of these letters are specified, then none of the corresponding attributes will be set. Setting no attributes is appropriate only for sections containing debugging or other information not intended to be part of the final linked file. Sections which are intended to be part of the final linked output should have at least the **a** attribute. Once attributes have been set they may not be respecified. Due to limitations in the debug file formats, only one section per source file (counting the **.text** section if it is used) may have the **x** attribute, if debugging is intended.

Here are a few examples using the **.section** directive:

```
.section .mytext, ax
```

creates a section called **.mytext** with allocate and execute attributes.

```
.section .data2, a
```

creates a section called **.data2** with the allocate attribute.

.previous Changes the active section back to the one in use before the most recent **.section** directive.**.org** Directs assembly output into an absolute section which starts at address *man-expr*. The section is given a name equal to the eight-character hex representation of that address.

Symbol Definition

.bss *ident*,*man-expr*[,*man-expr*]

.comm *ident*,*man-expr*[,*man-expr*]

.lcomm *ident*,*man-expr*[,*man-expr*]

.weak *ident*

.export*ident*

.import*ident*

These directives define the type and value of the identifier *ident*.

.bss Same as **.comm** except *ident* is not exported.

.export Causes the identifier *ident* to be visible externally. If the identifier is defined in the current program, this directive allows the linker to resolve references by other programs. If the identifier is not defined in the current program, the Macro Assembler resolves it externally.

.comm Causes the specified identifier *ident* to be visible externally. The identifier is assigned to a common area of *man-expr* bytes in length. The linker assigns space for the identifier in the bss section if *ident* is not defined by another relocatable object file. The optional *man-expr* specifies the variable alignment in bytes.

.lcomm Same as **.comm** except *ident* is not exported.

.weak Makes the symbol weak. Linker sets its value to zero if the ident cannot be located.

.import Same as **.export**. It is common for **.export** to make symbols defined in the current module be externally visible, while **.import** explicitly declares symbols defined in other modules. The assembler does not require this usage, however.

File Inclusion

.include *file*

Causes the insertion of *file* at the location of this directive. The file will be searched first in the current working directory and then in any other directories specified by the **-I** command line option. See “Command Line Options” on page 120.

Macro Definition

.macro *name* [*list]*

[**.exitm**]

.endm

The **.macro** directive enters a macro definition. Assembly statements are collected until a matching **.endm** directive is processed. The following is an example of how to invoke a macro:

```
.macro trythis parm1
lwz r1,parm1*2 (r4)
.endm
.macro log_and parm1 parm2
lwz parm1, parm2 (r4)
.endm
.text
trythis 16
log_and r1, 0
log_and r1, 2
```

generates:

```
lwz r1,32(r4)
lwz r1,0(r4)
lwz r1,2(r4)
```

A macro definition assigns a name and a local parameter list to a sequence of assembly statements. The parameter list consists of identifiers separated by commas or white space. The name space of macro names is distinct from the names of other user defined symbols.

After a matching **.endm** directive is processed, the Macro Assembler recognizes the name of the macro and substitutes the saved assembly statements. This procedure invokes the macro and is known as *macro expansion*.

Actual parameters are supplied when the macro is invoked, and there must be the same number of actual parameters as there are identifiers in the parameter list of the macro definition.

The **.endm** directive must be the first symbol on its line; no labels are permitted.

During macro expansion, all references to a parameter of the definition are replaced by the corresponding actual parameter. The >< concatenation operator may be used to concatenate two parameters or a parameter with a symbol. The resulting assembly statement is not scanned for further parameter matches. If one macro calls another, the parameters of the first invocation are hidden from that of the inner one.

A macro may contain macro definitions. In this case the inner definition is processed only when the macro is later expanded. Macros may not call themselves recursively.

During macro expansion, if a **.exitm** directive is encountered, expansion of the macro is terminated. Typically, this directive would be placed within a **.if** directive structure to allow for conditional premature termination of the macro expansion.

When a macro is invoked, the name of the macro appears in the listing. The expansion of the macro and the correspondingly generated object code are then listed. The directive can be used to disable the listing of the macro expansion. See “Listing Format” on page 141 for more information.

Repeat Block

.rept *expr*
. . .
.endr

This directive specifies a block of assembly statements which repeat *expr* times.

The block of instructions is repeated *expr* times. The expression must be constant. Repeat blocks may occur within repeat blocks. In this case the inner repeat block is expanded once for each expansion of the next outer block. The repeat count of an inner block is evaluated at each expansion of the inner block.

The **.endr** directive must be the first symbol on its line; no labels are permitted.

Repeat blocks may be contained within macro definitions, or definitions within blocks, but no other overlap is possible.

Conditional Assembly

.if *expr*
[**.else**]
[**.endif**]
.if *expr*

[**.elseif** *expr]*
.endif

The **.if** directive specifies a block of assembly statements which are to be assembled only if *expr* is non-zero. The reverse condition applies to the **.else** block, and the reverse of the condition plus a new condition applies to an **.elseif** block.

The *expr* is evaluated. It must be constant and defined within Pass 1. If its value is non-zero, the block of statements is assembled normally. Otherwise, the generation of code, the definition of symbols and labels, and the processing of directives is suppressed until a matching **.endif** is processed.

The **.else** directive may be used to reverse the condition and begin assembling statements only if the matching **.if** was false. The **.elseif** directive is equivalent to a **.else** followed by a second **.if**, except that only one **.endif** will be required to terminate the block.

Conditional blocks may occur within conditional blocks.

The conditional block is always listed, but no object code listing will appear for blocks which are not assembled.

Symbolic Debugging and Revision Tracking

.file *q-string*

.ln

These directives are used for symbolic debugging.

.file Stores a source filename, *q-string*, into the object file symbol table. The *q-string* filename must be from 1 to 255 characters in length and delimited by double quotes.

.ln This directive creates a line number table entry in the object file, associating the line number, *line-no*, with a particular memory location, optionally specified by address. If no address is specified, the current location in the current section is used.

Symbol Attribute Operations

The .**def** and .**endef** directive pair is used to create a symbol table entry for the specified identifier and to associate one or more attributes with that identifier. The general format for an .**def**/.**endef** is:

def *identifier*
.... (one or more attribute assignment operations)
.endef

If the **.def**/**.endef** pair defines a function name, a second **.def**/**.endef** pair, which assigns a storage class of -1, must immediately follow the function definition set. This allows the Macro Assembler to calculate function size for use with other tools.

The following attribute assignment operators may be specified within an **.def**/**.endef** pair:

.dim The **.dim** pseudo-op indicates that the identifier is an array. Each dimension of the array is specified by a manifest expression *man-expr*, the total number of dimensions being defined by the number of comma-delimited *man-expr* values supplied.

.line The **.line** pseudo-op is used to associate a line number, *man-expr* with the identifier. In this case the identifier specified by **.def** should be a block symbol. The maximum number is 4.

.scl The **.scl** pseudo-op associates a storage class, specified by *man-expr* with the identifier. The special storage class value of -1 is used to indicate the physical end of a function.

.size The **.size** pseudo-op associates the size specified by *man-expr* with the identifier. If the identifier is a bit field, the size is specified in bits. Otherwise the size is assumed to be in bytes.

.tag The **.tag** pseudo-op associates the identifier with a structure, union or enumeration named string.

.type The **.type** pseudo-op associates the C language type specified by *man-expr* with the identifier.

.val The **.val** pseudo-op assigns the value of *expr* to the identifier. The expression *expr* determines the section with which the identifier will be associated, and may be either an absolute expression, a relocatable expression or an undefined external. If *expr* is **.val**, then the current text section location is assigned.

Listing Format

.warning
.nowarning
.nolist[**.macro**][**.rept**][**.if**][**.include**][**.list**]
.list[**.macro**][**.rept**][**.if**][**.include**][**.list**]
.gen
.nogen
.eject
.title "*title***"**

.subtitle "*subtitle***"**
.sbttl "*subtitle***"**

These options control the format of the informational text listing produced by the assembler. They do not affect the object code generated.

.warning Causes warnings to be emitted to standard error output.

.nowarning Causes warnings to not be emitted to standard error output.

The following options will only be effective if the source listing has been enabled with the **-list** or **-l** command line options.

.nolist Without any arguments, this will turn off listing for the sections following this directive until a corresponding **.list** directive is encountered at which point listing will be reactivated.

.nolist .macro

Identical to **.nogen**. This causes all macros in the sections following this directive to not be expanded in the source listing. Macro expansion listing can be reactivated with a **.list .macro** or a **.gen** directive.

.nolist .rept Does not expand all repeat blocks in the sections following this directive in the source listing. Repeat block expansion listing can be reactivated with a **.list .rept** directive.

.nolist .if Causes the listing of only the branch in conditional blocks (**.if** ...) for which code is generated. The full conditional block listing can be reactivated with a **.list .if** directive.

.nolist .include

Does not display include files in the sections following this directive in the source listing. Include file listing can be reactivated with a **.list .include** directive.

.nolist .list Does not display all **.list** and **.nolist** directives in source listings. The **.list** directive listing can be reactivated with a **.list .list** directive.

.list Without any arguments, this turns on listing for the sections following this directive. This is intended to counteract a previously given **.nolist** directive.

.list .if Controls printing of all **.if .else .endif** directives and the lines skipped due to false if expressions.

.list .macro This is identical to **.gen**. This fully expands all macros in the source listing. This can counteract a previously given **.nolist .macro** directive.

.list .over Controls printing of lines with so much binary output that they overflow onto multiple lines.

.list .rept Controls printing the **.rept** directive itself. This fully expands all repeat blocks in the source listing. This can counteract a previously given **.nolist .macro** directive.

.list .if Lists all branches of conditional blocks (**.if** ...) in the source listing. This is intended to counteract a previously given **.nolist .if** directive.

.list .include Displays include files in the source listing. This is intended to counteract a previously given **.nolist .include** directive.

.list .list Controls printing the **.list** directives itself. This is intended to counteract a previously given **.nolist .list** directive.

.gen Identical to **.list .macro**.

.nogen Identical to **.nolist .macro**.

.eject Puts a form feed (**^L**) into the output listing.

.title Causes a title, given by the specified string, to be included at the top of each page of the source listing.

.subtitle Causes a subtitle, given by the specified string, to be included at the top of each page of the source listing.

.sbttl Identical to **.subtitle**.

11

MCore Macro Assembler Reference

This chapter contains:

- Register Set
- Addressing Modes
- Macro Expansion
- Alphabetical List of MCore Instructions

This chapter gives detailed information on the MCore addressing modes and instruction formats, with a complete alphabetical listing of all MCore instructions. Numerous examples are provided. The processor-specific information in this chapter supplements the more general information provided by the MCore Macro Assembler chapter.

While it is intended as a useful reference for the programmer wishing to write and maintain Green Hills MCore Macro Assembler code, it does not cover these topics exhaustively. For additional information, please refer to the *MCore Programming Manual* which also covers other related topics such as a detailed description of the instruction set and trap handling.

Register Set

There are two types of registers, *general registers* and *control registers*, explained below.

General Registers

There are 16 general purpose registers, each 32 bits wide. They are used to hold source operand data and computation results. Although only **r0** has a special function at the hardware level, a convention has been established whereby the following MCore general registers have reserved functions at the software level:

Register Name(s)	Usage
r0	Stack pointer
r1	Scratch register
r2-r3	Parameter registers, return value
r4-r7	Parameter registers
r8-r13	Permanent registers
r14	Permanent register, frame pointer
r15	Link pointer

The **jsr** instruction overwrites the contents of register **r15** with the return address generated by the call. However, the contents of **r15** may also be overwritten by software if required.

Control Registers

There are 32 control registers, each 32-bits wide:

Name	Usage
PSR	Processor status register
VBR	Vector base register
EPSR, FPSR, EPC, FPC	Exception shadow registers
SS0-SS4	Supervisor storage registers
GCR	Global central register
GSR	Global status register
CR13-CR31	Reserved

Control registers can be loaded and stored from via the **mfcr** and **mtcr**, respectively. For a detailed description of the purpose of each of these registers, please refer to the *MCore Programming Manual*.

Addressing Modes

Introduction

All the addressing modes offered by the MCore processor are supported, and are summarized below:

Addressing Mode	Notation	Example
No arguments		rts
Register	reg1	abs r1
Two registers	reg1,reg2	add r2,r3
Register with 5-bit immediate	reg1,imm5	sub r0,16
Register with 7-bit immediate	reg1,imm7	movi r4,100
Control register	reg1,creg2	mfcr r2,cr1
Register indirect with 4-bit displacement	reg1,(reg2,disp4)	ld.w r3,(r0,16)
Register list	reg1-reg2, (reg3)	stm r2-r15,(r0)
Immediate Indirect	[disp8]	lrw r1,[100]
Branch displacement	disp11	bt-16
Register with 4-bit negative displacement	reg1,disp4	loopt r2,-8

Key:

reg1, **reg2** are general purpose registers

creg1 is a control register

imm5 is an unsigned 5-bit value

imm7 is an unsigned 7-bit value

disp4 is a signed or unsigned 4-bit displacement value

disp8 is an unsigned 8-bit displacement value

disp11 is a signed 11-bit displacement value

Each of these addressing modes is explained below.

Every MCore instruction is two bytes (16 bits), including those instructions containing immediate data.

In line with the RISC architecture, there are relatively few instructions and usually few addressing modes which apply to any given one. Also, a given addressing mode applies to the instruction as a whole (in contrast to some architectures which allow different addressing modes to be specified for each operand). The result is very quick and easy instruction decoding, so the lack of complex instructions and addressing modes is more than compensated for by the attendant increase in performance.

The values in the example boxes are all in hexadecimal, except that **xx** means "don't care what value is present". 32-bit values are written as four 8-bit bytes.

Example:

```
        r2:       DE AD C0 DE
        r3:     xx xx xx xx
DEADB0AC:   xx xx xx xx
DEADB0B0:   C0 1D BE EF
DEADB0B4:  xx xx xx xx
```

This indicates that register **r2** holds the value 0xDEADC0DE, we do not care what value **r3** holds (it holds arbitrary data), the four bytes of memory starting at address 0xdeadb0b0 hold the value 0xC01DBEEF, and the four-byte words above and below each hold arbitrary data.

No Arguments

These instructions do not take any arguments.

Example:

```
rts
```

Register

These instructions use the same register as the source and destination.

Example:

```
abs r1
```

This computes the absolute value of the contents of register **r1** and stores the result in register **r1**.

Two Registers

These instructions have two register fields to specify one or two source registers and one destination register for the instruction.

Example:

```
add r2,r3
```

This adds the contents of register **r3** to register **r2** and stores the result in register **r2**.

Example:

```
mov r2,r14
```

This copies the contents of register **r14** to register **r2**. **r14** remains unchanged.

Register with 5-bit Immediate

This addressing mode has a 5-bit immediate field as the first source operand while one register field specifies both the second source operand and the destination.

Example:

```
sub r0,16
```

This subtracts 16 from the value in register **r0** and stores the result back in **r0**.

Register with 7-bit Immediate

The **movi** instruction has a 7-bit immediate field as a source operand and a register field as the destination.

Example:

```
movi r4,100
```

This stores the value 100 in register **r4**.

Control Register

These instructions copy data between the general registers (**r0**-**r15**) and the control registers (**cr0** to **cr31**).

Example:

mfcr r2,cr1
This copies the contents of control register **cr1** to general register **r2**.

mtcr r3,cr0
This copies the contents of general register **r3** to control register **cr0**.

Register Indirect with 4-bit Scaled Displacement

This addressing mode adds the contents of register **reg1** to the scaled unsigned immediate **disp4** to form an address. The ld instructions load the data at that address to the register **reg2**. The **st** instructions store the contents of register **reg2** to that address.

Example:

ld.w r3,(r0,16)
This copies to register **r3** the 32-bit data at the memory address calculated by adding 16 to the contents of register **r0**. The immediate value 16 must be a multiple of 4 because this is a 4-byte load instruction.

Register List

This addressing mode specifies a contiguous set of registers to transfer to or from the memory location pointed to by the contents of register **reg1**.

Example:

stm r12-r15,(r0)
This stores registers **r12**, **r13**, **r14**, and **r15** in ascending memory locations starting at the address stored in register **r0**.

Scaled 8-bit Immediate Indirect

This addressing mode uses a 32-bit word pointed to by a PC-relative address as a source operand. The address is computed by adding the unsigned 8-bit immediate field, scaled by four, to the value of PC+2. The lower two bits of this address are then masked to 00.

lrw r1,[100]
This loads 32-bit word at address **PC**+2+100 and stores the result in register **r1**.

Scaled 11-bit Branch Displacement

This addressing mode computes a branch address by adding the sign-extended displacement value **disp11**, scaled by two, to the address **PC**+2.

Example:

```
bt -16
```

This instruction branches to the address **PC**+2-16 if the condition code bit is set.

Register with 4-bit Negative Displacement

The loopt instruction uses this addressing mode to specify a register to store a loop counter and a branch address formed by subtracting the 4-bit displacement, scaled by two, from the address **PC**+2.

Example:

```
loopt r2,-8
```

This instruction branches to the address **PC**+2-8 if the loop counter in register **r2** is not zero.

Macro Expansion

The MCore Macro Assembler supports several macro expansions as specified in the MCore Applications Binary Interface. These macros are described below.

clrc Clears the condition code bit. Equivalent to:

```
cmpne r0,r0
```

cmplei *rd,n*Compare if the signed value in *rd* is less than or equal to the constant *n*. *n* is allowed to have the values 0 through 31. Equivalent to:

```
cmplti rd,n+1
```

cmpls *rd,rs*Compare if the unsigned value in *rd* is lower or the same as the unsigned value in *rs*. Equivalent to:

```
cmphs rs,rd
```

cmpgt *rd,rs*Compare if the signed value in *rd* is greater than the signed value in *rs*. Equivalent to:

```
cmplt rs,rd
```

jbsr *label*If the address of the label is between -2048 and +2046 bytes away, this expands to:

```
bsr label
```

Otherwise:

```
jsri label
```

jbr *label*If the address of the label is between -2048 and +2046 bytes away, this expands to:

```
br label
```

Otherwise:

```
jmpi label
```

jbf *label*If the address of the label is between -2048 and +2046 bytes away, this expands to:

```
bf label
```

Otherwise:

```
bt 1f
jmpi label
1:
```

jbt *label*If the address of the label is between -2048 and +2046 bytes away, this expands to:

```
bt label
```

Otherwise:

```
bf 1f
jmpi label
1:
```

neg *rd* Negates the value in *rd*. Equivalent to:

```
rsubi rd,0
```

rotlc *rd*,1Rotates the value in *rd* left by one bit. The carry bit is rotated into the least significant bit while the most significant bit that was rotated out is saved in the carry bit. Equivalent to

```
addc rd,rd
```

rotri *rd*,*imm*Rotates the value in *rd* right by the number of bits specified in *imm*. Equivalent to:

```
rotli rd,32-imm
```

rts Returns from subroutine. Equivalent to:

```
jmp r15
```

setc Sets the condition code bit. Equivalent to:

```
cmphs r0,r0
```

tstle *rd*Test for a negative or zero value in the register *rd*. Equivalent to:

```
cmplti rd,1
```

tstlt *rd*Test for a negative value in the register *rd*. Equivalent to:

```
btsti rd,31
```

tstne *rd*Test for a non-zero value in the register *rd*. Equivalent to:

```
cmpnei rd,0
```

Alphabetical List of MCore Instructions

Instruction	Description	Operands	Opcode(hex)
abs	absolute value	reg1	01e0
addc	unsigned add with carry	reg1,reg2	0600
addi	unsigned add with immediate	reg1,imm5	2000

Instruction	Description	Operands	Opcode(hex)
addu	unsigned add	reg1,reg2	1c00
and	logical AND	reg1,reg2	1600
andi	logical AND with immediate	reg1,imm5	2e00
andn	logical AND NOT	reg1,reg2	1f00
asr	arithmetic shift right	reg1,reg2	1a00
asrc	arithmetic shift right by 1 bit	reg1	3a00
asri	arithmetic shift right immediate	reg1,imm5	3a00
bclri	bit clear immediate	reg1,imm5	3000
bf	branch if false	disp11	e800
bgeni	bit generate immediate	reg1,imm5	3200
bgenr	bit generate register	reg1,reg2	1300
bkpt	breakpoint	-	0000
bmaski	bit mask generate immediate	reg1,imm5	2c00
br	unconditional branch	disp11	f000
brev	bit reverse	reg1	00f0
bseti	bit set immediate	reg1,imm5	3400
bsr	branch to subroutine	disp11	f800
bt	branch if true	disp11	e000
btsti	bit test immediate	reg1,imm5	3600
clrf	clear if condition false	reg1	01d0
clrt	clear if condition true	reg1	01c0
cmphs	compare for higher or same	reg1,reg2	0c00
cmplt	compare for less than	reg1,reg2	0d00
cmplti	compare with immediate for less than	reg1,imm5	2200
cmpne	compare for not equal	reg1,reg2	0f00
cmpnei	compare with immediate for not equal	reg1,imm5	2a00
decf	decrement if condition false	reg1	0090
decgt	decrement and compare greater than	reg1	01a0
declt	decrement and compare less than	reg1	0180
decne	decrement and compare not equal	reg1	01b0
dect	decrement if condition true	reg1	0080
divs	signed divide	reg1,r1	3210
divu	unsigned divide	reg1,r1	2c10
doze	enter doze mode	-	0006
ff1	find first one	reg1	00e0
incf	increment if condition false	reg1	00b0
inct	increment if condition true	reg1	00a0
ixh	index halfword	reg1,reg2	1d00

Instruction	Description	Operands	Opcode(hex)
ixw	index word	reg1,reg2	1500
jmp	jump	reg1	00c0
jmpi	jump indirect	[disp8]	7000
jsr	jump to subroutine	reg1	00d0
jsri	jump to subroutine indirect	[disp8]	7f00
ld.b	load unsigned byte	reg1,(reg2,disp4)	a000
ld.h	load unsigned halfword	reg1,(reg2,disp4)	c000
ld.w	load word	reg1,(reg2,disp4)	8000
ldm	load multiple registers	reg1-r15,(r0)	0060
ldq	load register quadrant	r4-r7,(reg1)	0040
loopt	decrement and loop	reg1,disp4	0400
lrw	load PC-relative word	reg1,[disp8]	7000
lsl	logical shift left	reg1,reg2	1b00
lslc	logical shift left by 1 bit	reg1	3c00
lsli	logical shift left immediate	reg1,imm5	3c00
lsr	logical shift right	reg1,reg2	0b00
lsrc	logical shift right by 1 bit	reg1	3e00
lsri	logical shift right immediate	reg1,imm5	3e00
mfcr	move from control register	reg1,creg2	1000
mov	logical move	reg1,reg2	1200
movf	move if condition false	reg1,reg2	0a00
movi	move immediate	reg1,imm7	6000
movt	move if condition true	reg1,reg2	0200
mtcr	move to control register	reg1,creg2	1800
mult	multiply	reg1,reg2	0300
mvc	move carry bit to register	reg1	0020
mvcv	move inverted carry bit to register	reg1	0030
not	logical NOT	reg1	01f0
or	logical OR	reg1,reg2	1e00
rfi	return from fast interrupt	-	0003
rotli	rotate left immediate	reg1,imm5	3800
rsub	reverse subtract	reg1,reg2	1400
rsubi	reverse subtract with immediate	reg1,imm5	2800
rte	return from exception	-	0002
sextb	sign extend byte	reg1	0150
sexth	sign extend halfword	reg1	0170
st.b	store byte	reg1,(reg2,disp4)	b000
st.h	store halfword	reg1,(reg2,disp4)	d000

Instruction	Description	Operands	Opcode(hex)
st.w	store word	reg1,(reg2,disp4)	9000
stm	store multiple registers	reg1-r15,(r0)	0070
stop	enter stop mode	-	0004
stq	store register quadrant	r4-r7,(reg1)	0050
subc	unsigned subtract with carry	reg1,reg2	0700
subi	unsigned subtract with immediate	reg1,imm5	2200
subu	unsigned subtract	reg1,reg2	0500
sync	synchronize CPU	-	0001
trap	trap to operating system	imm2	0004
tst	test with zero	reg1,reg2	0e00
tstnbz	test for no byte equal to zero	reg1	0190
wait	wait for interrupt	-	0005
xor	logical exclusive OR	reg1,reg2	1700
xsr	extended shift right	reg1	3800
xtrb0	extract high order byte	r1,reg2	0130
xtrb1	extract byte 1	r1,reg2	0120
xtrb2	extract byte 2	r1,reg2	0110
xtrb3	extract low order byte	r1,reg2	0100
zextb	zero extend byte	reg1	0140
zexth	zero extend halfword	reg1	0160

12

The Librarian

This chapter contains:

- Description
- Command Line Options
- Examples

The librarian combines object modules created by the Assembler or Linker into a library file. Below are the command line options and examples to understand this function.

Description

ax [*options*] *archive-file input-files*

The **ax** command creates library archives of **.o** object files for use by the linker.

By convention, archives of object files for use by the linker are given the extension **.a**. The linker can search such library archives and extract only those object files which are needed to provide definitions of undefined symbols. This provides a convenient way to make a number of object files available to the linker while linking in only those which are necessary.

There are two important features supported for version 1.8.9:

1. Filenames longer than 15 characters are now fully supported by **ax** and by all Green Hills tools which operate on archives. This is accomplished by storing the full name of the file in a hidden member of the archive called //. The filename field of the archive header for the file with a long name will have an entry of the form /*nnn* where *nnn* is a decimal integer representing an offset into the // file where the entire filename can be found.
2. A table of contents, also known as a symbol table, is now generated by ax whenever the archive is modified by either the **d** or **r** option. This table of contents is stored in the form of a hidden file named / which is always the first file in the archive. In 1.8.8 the granlib utility was provided to generate this table of contents, but in 1.8.9, the granlib utility is no longer needed, because this operation is integrated within ax. The new options **s** and **S** are provided to force or prevent the creation of a table of contents.

   The lx linker supports archives with or without a table of contents, but the table of contents is necessary for the -rescan option to lx to have any effect. Future linkers will require the table of contents in order to process an archive.

Command Line Options

The command line **ax help** prints the following:

ax d[eSv] archive files...

Deletes named files from an archive file.

ax r[ceSv] *archive files...*

Replaces (or adds) named files in an archive file. Any archive copies of the named files are deleted and new contents of the files are added to the archive. It is not necessary for the file to have previously been in the archive. If the archive file did not previously exist, it will be created and a warning message printed (unless option **c** is also specified).

ax t[esv] *archive [files...]*

List files in the archive. Without **v**, this just lists the names of the files. With **v**, the list includes file sizes and dates.

ax x[ev] *archive [files...]*

Extracts named files from an archive file. The archive will be searched for the specified filenames and the named files will be created and written with the contents of the archived files. The archive is not altered by this command.

ax q[ev] *archive files...*

Quickly appends named files to the end of the archive. **q** is similar to the **r** option except that the files will always be added to the end of the archive, rather than replacing any existing version of the file with the new version. This command will be unsupported in the near future.

ax p[ev] *archive [files]*

Prints files to standard output.

These letters may be used with primary option letters as shown above. Options must all appear as one string without spaces:

c Suppress warning for creation of archive if it doesn't exist.

e Prefix messages with **ERROR** or **WARNING**. Equivalent to the driver option -prefixed_msgs, described in Chapter 8, "Compiler Driver Options".

s Used with **t** to regenerate table of contents.

S Suppress generation of table of contents.

v Verbose mode.

Note: If a minus sign (-) is used as a prefix to the first option in the **ax** command, it is silently ignored. For example:

```
ax -rv libx.a file.o
```
is the same as

```
ax rv libx.a file.o
```

Examples

To create a library archive file of object modules suitable for input to the linker, a command may be used such as:

```
ax cr libmystuff.a myfile1.o myfile2.o myfile3.o
```

To add another object module to the existing library archive:

```
ax r libmystuff.a myfile4.o
```

To delete an object module from the existing library archive:

```
ax d libmystuff.a myfile4.o
```

To replace an object module in the existing library archive:

```
ax r libmystuff.a myfile3.o
```

To extract two object modules from the existing library archive:

```
ax x libmystuff.a myfile1.o myfile2.o
```

To append two object files to an existing library:

```
ax q libmystuff.a add1.o add2.o
```

To print the table of contents of the existing library archive using verbose mode:

```
ax tv libmystuff.a
```

If the archive file consisted of three object files, **foo.bar**, **bar.o**, and **etc.o**, the previous command would produce:

```
rw-rw----111/24          110 Mon Jun 22 09:43:14 1992 foo.o
rw-rw----111/24          141 Mon Jun 22 15:05:41 1992 bar.o
rw-rw----111/24          141 Mon Jun 22 09:47:22 1992 etc.o
```

To extract a single file into a different name, the **p** option may be used:

```
ax p libmystuff.a foo.o > newfoo.o
```

The **v** option must not be used with **p** because the filename, **foo.o**, will also be written to standard output.

Note: The driver option **-archive** to the compiler driver is closely related to the **ax** command (it invokes **ax**). See "General Options" on page 68 for more information about the **-archive** command.

13

The ELXR Linker

This chapter contains:

- Command Line Options
- Program Entry Point
- Section and Memory Maps
- Expressions
- Section Attributes
- Green Hills Specific Linker Features
- Porting Guide from other linkers

Command Line Options

Option Processing

Single-letter options may be followed by an argument with or without whitespace, or following an =. Example: the following are equivalent:

-o*argument*

-o=*argument*

-o *argument*

In the case of ambiguity between a single-letter option with an appended argument, and a multiple-letter option, the multiple letter option takes precedence. Use either of the other single-letter option forms when required.

Multiple-letter options may be preceded by one or two dashes; an argument must be separated by an = or given as the following argument. Example: the following are equivalent:

-option=argument

-option argument

Options

@commandfile **(aliases: -T*commandfile*)**
Additional options are read from file *commandfile*. Within the command file, the pound sign (#) marks the remainder of that line as a comment.

-A *file* **(aliases: -R)**
Read in symbol names and addresses only from object file *file*. The object file's contents will not be relocated or included in the output. This is useful when one linker image must refer to symbols which are located in another separately linked image.

-a
Causes the output file to be relocatable and executable. Relocation is performed, and final link steps are performed (such as C++ constructors creation, common allocation, and special symbol creation), but relocation information is retained in the output file. Implies **-r**.

Some of the final link steps, including but not limited to: C++ constructors and special symbols, are not guaranteed to have relocations, and thus may not be valid if the outfile file is loaded at a different address.

-checksum/-nochecksum **(**default: -nochecksum**)**
Add a 4-byte checksum to the end of every program section. The algorithm used is a standard 32-bit CRC using polynomial `0x10211021`.

-e *symbol*
The program's entry point is set to the address of symbol. See Program Entry Point.

-sections { ... } (aliases: -sec, SECTIONS)
Specifies a section map. See Section and Memory Maps.

-L *directory*
Add directory to those searched for libraries specified by **-l**; may be repeated. All **-L** options on the command-line will be processed before any **-l** options. Directories will be searched in the order which they appear on the command-line.

-l*name*
Look for **libname.a** in directories specified by **-L**.

-memory { ... } **(**aliases: MEMORY**)**
Specifies a memory map. See Section and Memory Maps.

-r
Causes the output file to retain relocation information. The output file may be used as an input file in further link steps. Implies **-undefined**. See also **-a**.

-undefined
Causes **elxr** to not check for undefined symbol references. Any undefined symbols will be given an address of 0. See also **-a**.

Program Entry Point

There are several ways to specify the program entry point. The following list shows (in descending order of precedence) how **elxr** sets the entry point:

- **-e** symbol command-line option, if present
- the value of the symbol **_start**, if present
- the value of the symbol **start**, if present
- the value of the symbol **_main**, if present
- the value of the symbol **main**, if present
- the zero address

Section and Memory Maps

Section Definition

A section map is formatted as follows:

SECTIONS {
...
secname [*start_expression*] [*attributes*] : [{ *contents* }]
...
}
Only *secname* and the **:** (colon) are required. All other entries may be omitted. All sections in input files which participate in memory layout must be referenced in the section map.

start_expression

The value of this expression is used as the starting address of this section. If omitted, the section starts at the current address. In either case, the starting address is further modified to fit alignment constraints of the subsections included by *contents*.

attributes
Any number of attributes from the **Section Attributes**, below, may be specified.

{ *contents* }

Any number of section inclusion commands and expressions may be specified.

Section inclusion commands are of the form *filename*(*secname*), which directs that the section *secname* from *filename* should be included. *filename* may be replaced by ***** to specify sections in all files not specifically mentioned.

If '**{**', contents, and '**}**' or if "{ *contents* }" are omitted, the linker includes sections named *secname* from all files, as if { *(*secname*) } had been entered. To avoid this, specify an empty section as follows: **{ }**.

Expressions may be used to create or modify the value of any symbol. If a non-existent symbol is assigned a value, that symbol is created relative to that section.

The special symbol **.** (dot) may be referred to in expressions; it evaluates to the current position, which is the offset from the beginning of the section. An assignment increasing the value of dot will add padding at the current position. An assignment decreasing the value of dot will result in an error.

All assignments to any symbol are section relative; the numbers involved are offsets from the start address of the section. See the **ABSOLUTE** function to get an address.

Depending on the target and the enabled optimizations, program layout may occur multiple times. Therefore you should avoid using expressions which depend on the number of evaluations. If necessary, you may use the expression **final()** to ensure that an expression is evaluated during the final layout only.

Example:

To set the low bit of a symbol **func** if the symbol already exists:

```
.text : { isdefined(func) ? (func += 1) : 0; } /*incorrect, may increment multiple times */
.text : { final(isdefined(func) ? (func += 1) : 0); } /* correct */
```

Expressions

The following functions are recognized during expression evaluation. Their names are case-insensitive.

absolute(*expr***)**
: Given a section-relative offset value, absolute returns the absolute address by adding the address of the containing section to value. It is an error to use absolute outside of a section contents *section*.

addr(*section***)**
: Returns the memory address of the section named *section*.

sizeof(*section***)**
: Returns the current size of the section name *section*.

align(*expr***)**
: Returns the current position (’.’) aligned to a value boundary. This is equivalent to:

(. + *expr* - 1) & ~(*expr* -1)

pack_or_align(*expr***)**
: This is generally only used as the start_expression for a section map. It returns the current position (**.**) aligned such that the section will not span a page boundary of size value. This is equivalent to:

(. % value) + sizeof(this_section) > value ? align(value) : .

min(*value1*,*value2***)**

max(*value1*,*value2***)**

Returns the minimum or maximum, respectively, of the two values supplied.

error("*string***")**

Generates a linker error, displaying string, as well as the current section's name and address, and the current section offset.

isdefined(*symbol***)**

Returns **1** if a global symbol exists and is defined, **0** otherwise.

final(*finalexpression* [,*earlyexpression=0*])

Section Attributes

ABS

Sets a flag in the output file that indicates this section has an absolute address, and should not be moved. Program loaders and other utilities that manipulate the output image should not include this section in any movement related to position independent code or data.

CLEAR, NOCLEAR

Sets or removes the clear attribute of this section. If the clear attribute is present, an entry is made in the Runtime Clear Table, which is often used by startup code to initialize memory regions to a particular value.

The clear attribute is set by default for any section that includes a COMMON or SMALLCOMMON section, which are by default included by .bss and .sbss, respectively.

Examples:

```
.bss :          /* defaults to { *(.bss) *(COMMON) } - which implies CLEAR */
.sbss NOCLEAR : /* defaults to { *(.sbss) *(SMALLCOMMON)}, but will now
not have a clear entry */
.mysbss NOCLEAR : { file1.o(SMALLCOMMON) }  /* disables default clearing
*/
.stack CLEAR PAD(0x1000) :                  /* this section will now have a clear
entry */
```

PAD(*expr***)**

pad(*expr***)**

The linker will place value bytes of padding at the beginning of this section. This is equivalent to specifying padding at the beginning of the section contents.

The following two examples are equivalent:

```
.stack PAD(0x10000) : {}
.stack : { . += 0x10000; }
```

ROM(*section***)**

This section becomes a ROMmable copy of section. This section inherits the attributes and data of section, while section is modified to reserve address space only (as if it were all padding with no data). An entry is made in the Section-Info section to allow startup code to copy this section from this section (ROM) to section (RAM). See Runtime Copy Table. It is an error to specify section contents for a ROM section, or to have multiple sections ROMming the same section.

MIN_SIZE(expr**)**

Instructs the linker to possibly pad to ensure that this section is at least size bytes in length.

Example:

```
.stack MIN_SIZE(0x400) : { ... }        /* equivalent to the following: */
.stack : { ...  . = max(.,0x400); }
```

MIN_ENDADDRESS(*expr***)**

Instructs the linker to possibly pad to ensure that this section extends to at least address.

Example:

```
.stack MIN_ENDADDRESS(0x10000) : { ... }    /* equivalent to the following: */
.stack : { ...  . = max(ABSOLUTE(.),0x10000) - ADDR(.stack); }
```

MAX_SIZE(*expr***)**

Indicates that an error should be generated if this section exceeds size bytes in length during final layout.

Example:

```
.stack MAX_SIZE(0x4000) : { ... }        /* equivalent to the following: */
.stack : { ...  final(. > 0x4000 ? ERROR("section limit exceeded") : 0; }
```

MAX_ENDADDRESS(*expr***)**

Indicates that an error should be generated if this section extends beyond address during final layout.

Example:

```
.stack MAX_ENDADDRESS(0x10000) : { ... } /* equivalent to the following: */
.stack : { ...  final(ABSOLUTE(.) > 0x10000 ? ERROR("section limit exceeded")
: 0; }
```

Green Hills Specific Linker Features

Section-Info Section (.secinfo)

The **.secinfo** section contains special tables which contain information needed by startup code to clear sections (Runtime Clear Table), and copy sections from ROM to RAM (Runtime Copy Table).

Runtime Clear and Copy Tables

These two tables are contained within the **.secinfo** section; the Clear table is bounded by the symbols __ghsbinfo_clear and __ghseinfo_clear, while the Copy table is bounded by the symbols __ghsbinfo_copy and __ghseinfo_copy. The tables each contain zero or more records detailing the action to be taken at startup. By default, the Green Hills C runtime will perform .bss clearing and ROM copying based on this table without user intervention. The remainder of this section only needs to be referred to if you want to override these default actions.

The clear structure is as follows:

```
void    *base; /* pointer to base of memory to clear */
int     value;  /* value to initialize with (generally zero) */
size_t  length; /* number of bytes to clear */
```

These values are appropriate to be passed directly into the memset() routine within the C runtime library. The default clear code in the C runtime thus is as follows:

```
{
extern rodata_ptr __ghsbinfo_clear, __ghseinfo_clear;

void **b = (void **) __ghsbinfo_clear;
void **e = (void **) __ghseinfo_clear;

while (b != e) {
    void *          t;                  /* target pointer       */
    ptrdiff_t       v;                  /* value to set         */
    size_t          n;                  /* set n bytes          */
    t = (char *)(*b++);
    v = *((ptrdiff_t *) b); b++;
    n = *((size_t   *) b); b++;
    memset(t, v, n);
}
}
```

The copy structure is as follows:

```
void    *dest;  /* pointer to base of memory to copy to */
void    *src;   /* pointer to base of memory to copy from */
```

```
size_t  length; /* number of bytes to copy */
```

These values are appropriate to be passed directly into the memcpy() routine within the C runtime library. The default copy code in the C runtime is as follows:

```
{
extern rodata_ptr __ghsbinfo_copy, __ghseinfo_copy;

void **b = (void **) __ghsbinfo_copy;
void **e = (void **) __ghseinfo_copy;

while (b != e) {
    void *  t;                      /* target pointer       */
    void *  s;                       /* source pointer       */
    size_t  n;                       /* copy n bytes         */
    t = (char *)(*b++);
    s = (char *)(*b++);
    n = *((size_t *) b); b++;
    memcpy(t, s, n);
}
}
```

The actual implementation of the above two routines for your CRT can be found in libsrc/ind_crt0.c in your Green Hills distribution, and differs slightly for PIC/PID support on some targets.

Begin and End of Section Symbols

When the linker is performing final symbol resolution for a non-relocatable output file, certain undefined symbol names are recognized as referring to memory addresses in the final section map. These symbol names are constructed by prepending the strings _ _ghsbegin and _ _ghsend to the name of each section in the output file, with any period (.) characters in the section names changed to underscores (_). For a section named .text the symbols _ _ghsbegin_text would resolve to the virtual address of the start, and _ _ghsend_text to the virtual address of the end, of that section.

Example

For this section map:

```
{
    .text  0x100000         :
    .data    0x300000       :
    .bss1  0x400000         :
    .bss                    :
}
```

And this program:

```
      main.c:
      extern char _ _ghsbegin_bss1[], _ _ghsend_bss1[],      _
_ghsbegin_bss[];
      main() {
            memset(_ _ghsbegin_bss1, 0, _ _ghsend_bss1 -
    _ _ghsbegin_bss1);
            _ _ghsbegin_bss[0] = 0xff;
      }
```

If the size of section .bss1 is 0x100, then the linker will resolve _ _ghsbegin_bss1 to be 0x400000, _ _ghsend_bss1 to be 0x400100, and _ _ghsbegin_bss to be 0x400100.

Porting Guide from other linkers

LX

Section Attributes:

size() use **max_size()** instead (**size** is retained for compatibility)

limit() use **max_endaddress()** instead

Section Renaming:

Use curly braces:

lx style:

.newtext : file1.o(.text) file2.o(.text) ;
elxr style:

.newtext : { file1.o(.text) file2.o(.text) }

14

Utility Programs

The assembler Tool Chain contains more than 20 useful Utility Programs, including functional replacements for the standard UNIX utilities *dump*, *hide*, *nm*, *size*, and *strip*. All Utility Programs work with files generated by any Green Hills development tools.

Utility	ELF/BSD Object Files	ELF/BSD Object Library Files	ELF/BSD Executable Files	Function
gbincmp	Yes	Yes	Yes	Compare two binary files.
gcompare	Yes	Yes	Yes	Compare space or time performance.
gdump	Yes[+]	Yes[+]	Yes[+]	Like UNIX dump; dump/disassemble a file.
gfile	Yes	Yes	Yes	Like UNIX **file**; describe the file type.
gfunsize	Yes	Yes	Yes	Print function's code size.
ghexfile	No	No	Yes[-]	Convert an ELF or COFF to TEXHEX.
ghide	Yes	No	No	Hide global symbols in an object file.
gmemfile	No	No	Yes[-]	Generate binary image suitable for loading.
gnm	Yes	Yes	Yes	Like UNIX **nm**: print object file information.
grun	No	No	Yes	Execute in batch mode.
gsize	Yes	Yes	Yes	Like UNIX **size**: print section sizes.
gsrec	No	No	Yes	Convert to Motorola S-record format.
gstack	No	No	No	Compute the stack size for each task.
gstrip	No	No	Yes	Like UNIX **strip**: remove symbol/debug information.
gsymdump	No	No	No	Dump a **.dbg** or **.sym** file.
gtune	No	No	Yes	Automatically tune a program.
gversion	No	No	Yes	Print version date and time information.
gwhat	No	No	No	Like UNIX **what**.
gzero	No	No	Yes	Zero out proprietary data.

[*] No BSD support.
[+] For selected BSD targets only.
[-] Supports COFF but not BSD.

The gcompare Utility Program

The Green Hills **gcompare** Utility Program compares the code size of two input files and prints a report. An input file can be an ASCII text file, an object file, an

object file library, or an executable file. If an input file is a text file, it consists of lines in the following format:

```
name1  number1
name2  number2
...
```

Each name/number pair is on its own line, separated by blanks or tabs.

If an input file is an object file, an object library file, or an executable file, **gcompare** automatically runs **gfunsize -gcompare -all** (or another command specified by the **-x** option) to produce a text file to compare.

You can mix all input file types with no restrictions, including files for different target CPUs.

The **gcompare** utility first reads both input files. For any name which does not exist in both files, **gcompare** prints a warning, unless **-w** is specified. For each name which exists in both files, **gcompare** compares the code size of the old and new numbers. If the old number is worse (larger is worse unless **-i** is specified) **gcompare** writes a line to the output report file containing the name, old number, new number, and percentage by which the new number is better.

For example:

```
main          32    28    -14%
```

Usage

To use **gcompare**, enter:

gcompare [*options*] *oldfile newfile*

where

*options***gcompare** options, listed on the following page.

oldfile First input file.

*newfile*Second input file.

The **gcompare** *options* include:

-help Display information about all options.

-i Invert comparison. With **-i**, larger is better. Without **-i**, smaller is better.

-r Print a 2-column report, with comparisons sorted both from worst to best and from best to worst.

-l Print a report, with comparisons sorted from worst to best.

-L Format for 132 column landscape mode. Default is 80 column portrait mode.

-v Verbose mode. Print all comparisons. Without **-v**, only print comparisons for which *newfile* is worse than *oldfile.*

-w Suppress warnings.

-x *cmd* Specify the command to execute on a non-ASCII input file. The default **-x** command is:

```
-x "gfunsize -gcompare -all"
```

-z Do not show cases where files are the same.

If the **-r**, **-l**, or **-v** options are used, all comparisons are shown, regardless of the result of the comparison. With **-r**, two reports are shown side by side in two columns. The left column is sorted best first, and the right column is sorted worst first.

Sample **-r** output:

linpack.188.O.s vs linpack.188.OS.a				linpack.188.OS.a vs linpack.188.O.a			
WORSE by:	4172	3746	11%	BETTER by:	3726	4172	10%
linpack.o:_daxpy	152	112	-36%	linpack.o:_dscal	34	34	0%
linpack.o:_matgen	342	258	-33%	linpack.o:_epslon	20	20	0%
linpack.o:_dgesl	294	240	-23%	linpack.o:_idamax	72	72	0%
linpack.o:_dgefa	524	442	-19%	linpack.o:_dmxpy	1516	1598	5%
linpack.o_main	1136	1052	-8%	linpack.o:_main	1052	1136	7%
linpack.o:_dmxpy	1598	1516	-5%	linpack.o:_dgefa	442	524	16%
linpack.o:_idamax	72	72	0%	linpack.o:_dgesl	240	294	18%
linpack.o:_epslon	20	20	0%	linpack.o:_matgen	258	342	25%
linpack.o:_dscal	34	34	0%	linpack.o:_daxpy	112	152	26%

The output, especially in **-r** mode, can require many columns. The **-L** option formats the **gcompare** output for 132 columns instead of the default 80 columns. On a UNIX system, use the command **lpr -L** to print the resulting report in landscape mode.

The gdump Utility Program

The **gdump** Utility Program formats and prints information about a BSD or ELF object file, object library file, or executable file, including:

- the BSD or ELF file header
- program headers
- section headers
- (optional) symbol tables
- (optional) relocation sections
- (optional) **.plt** section
- (optional) **.got** section
- (optional) **.dynamic** section

Usage

To use **gdump**, enter:

gdump [*options*] *filename*

where

*options***gdump** options, listed below.

*filename*BSD or ELF file.

BSD File Options

Options when using **gdump** with BSD format files include:

-help Display information about all options.

-c Print section contents.

-h Print file header.

-r Print relocation entries.

-s Print symbol table.

ELF File Options

Options when using **gdump** with ELF format files include:

-asm Print text sections as pure assembly language (see also -ytext).

-dwarf Print DWARF information only.

-full Dump everything except section contents (see -ysec).

-help Display information about all options.

-load Print ELF header summary.

-map Print section summary.

-N Only print information as indicated by **-y** options.

-raw If **-ysec**, dump text sections in hexadecimal format, not disassembly.

-sx/nx Attempt/do not attempt shorter C++ demangling.

-sym Use symbol names, not numbers, in relocation output.

-v1 Print DWARF version 1. This is the default.

-v2 Print DWARF version 2. This version is not well supported.

-verify_checksum
Indicates to gdump that all non-empty, allocated sections have a 4 byte checksum generated by the GHS linker. The content of each section is compared against the existing checksum and if they do not match, both will be printed.

-print_checksum
Prints the checksum for each appropriate section. If **-verify_checksum** is also specified, checksums are assumed to exist and **-print_checksum** prints them for those sections where the checksum is found to be correct. If the **-verify_checksum** option is not specified, checksums are assumed non-existent in the section and calculates them, using all bytes in the section.

-yd/-nd
Print/do not print DWARF debug information.

-ydynamic/-ndynamic
Print/do not print dynamic linkage information.

-yg/-ng
Print/do not print global offset table.

-yh/-nh
Print/do not print ELF header information.

-yl/-nl
Print/do not print DWARF line number information.

-yr/nr
Print/do not print relocation information.

-yp/-np
Print/do not print procedure linkage table.

-ys/-ns
Print/do not print symbol table information.

-ysec/-nsec
Print/do not print section contents.

-ysh/-nsh
Print/do not print section header information.

-ystr/-nstr
Print/do not print string table information.

-yr/-nr
Print/do not print relocation information.

-ytext
Print contents of text sections only.

The gfile Utility Program

The **gfile** Utility Program is similar to UNIX **file**. The **gfile** Utility prints the file type of each filename argument. It may also display additional information. For example, if a machine supports both Big and Little Endian data ordering, then for an object file, object file library, or executable file, **gfile** displays the machine type and byte order. For unrecognized object files, **gfile** prints **unknown machine type**.

Usage

To use **gfile**, enter:

gfile [-help] *filename1* [*filename2* . . .]

where

-help Display information about all options.

*filename*Filename argument(s), separated by white space.

Examples

Example 1

There is an executable file named **a.out** in the current working directory. This executable was created by the Green Hills Tool Chain for an SH (Super Hitachi) CPU. Running **gfile** on **a.out** produces the following:

```
gfile a.out
```

```
a.out:  SH big endian
```

Example 2

The current host system is a SPARC workstation running the Solaris 2.x operating system, which uses the ELF format for executable files:

```
gfile /bin/od
/bin/od:        SPARC big endian executable ELF
```

The gfunsize Utility Program

The **gfunsize** Utility Program prints the code size of one or more named functions or all functions in an ELF object file, object library file, or executable file. For ELF, the code size of each function is part of the ELF symbol information.

MCore does not always give useful sizes because of literal pools. The compiler does not emit literals after every function, but defers emitting literals for as long as possible so that duplicate literals can be merged.

Usage

To use **gfunsize**, enter:

gfunsize [*options*] *filename*

where

*options***gfunsize** options, listed below.

*filename*Name of the ELF file.

The **gfunsize** *options* include:

-help Display information about all options.

-all Print the code sizes of all functions. This is the default.

-func=*name*Print code sizes of the specified function(s).

-addr Print function(s) addresses.

-hex Print function code sizes and addresses in hexadecimal.

-sectnum=*n* Only recognize functions in section number *n.*

-sect=*name*Only recognize functions in section *name.*

-gcompare Print the output in a format suitable for use as input to the **-gcompare** Utility Program.

-w Suppress warnings.

-file Print filename before each function.

-nounderscores
Strip leading underscores from function names.

The ghexfile Utility Program

The **ghexfile** Utility Program converts an ELF or COFF executable file to an extended Tektronix hexadecimal (TEXHEX) output file.

Usage

To use ghexfile, enter:

ghexfile [*options*] *input_file*

where

*options***ghexfile** options, listed below.

*input_file*Name of ELF or COFF executable file to be converted.

The **ghexfile** *options* include:

-help Display information about all options.

-cmd *file*
When **-cmd** is specified, the converter takes the command input from the given filename. **-cmd** options may be nested up to 4 levels deep. More than one **-cmd** option may appear on the command line. Command files are processed in the order in which they are encountered. C-style comments are accepted in the command input. Comments begin with "/*" and are terminated with "*/".

-length *n*
The -length option sets the maximum length of a TEKHEX block. The argument *n* must be a minimum of 40 and a maximum of 252. Any values outside of this range will cause an error message to be displayed. The default maximum size of a TEKHEX block is 80 bytes.

There is a certain number of bytes of overhead for each TEKHEX block. Larger block sizes require less blocks in the TEKHEX file, thus reduces the overhead and speeds up the time it takes to download the file.

-nodata
The option -nodata causes ghexfile to not output data blocks. The symbol formatter was used in the past to produce a TEKHEX file

containing symbols only. The option -nodata exists for backwards compatibility with the symbol formatter.

-nolocals

Do not emit local symbols to the TEKHEX output file. Local symbols are useful when debugging but extend download time. This has the same function as the linker map file switch -l.

-o *filename*

The -o option sets the name of the TEKHEX output file. If -o is not specified on the command line, the output filename is formed by removing the path and the extension of the ELF or COFF input file and adding the extension .tek. For example:

ghexfile /tmp/file.cfe

produces the TEKHEX output file named file.tek

-old

Produce output similar to what is produced by the linker with the -k option. Limit TEKHEX data blocks to 42 bytes per block. Limit TEKHEX symbol blocks to contain one symbol per block.

-skip *name*

The -skip option with a section name, will not translate data in the specified section. If the section that you specify is not in the ELF or COFF input file, the switch has no effect. If you want to skip more than one section, you must enter the command once for each section. For example: -skip .text -skip .data2

-y

The -y option suppresses printing the ghexfile banner.

To produce COFF, run the linker with the -z option. The TEKHEX support which exists in the linker with the -k option is still available, but cannot be used with linker directives nor COFF input files.

Features of ghexfile

The functionality of **ghexfile** is similar to the support in the linker and the symbol formatter, but not identical. Several new features have been added to make this utility more useful. Major features include:

- Control over the length of a TEKHEX block. In the linker, TEKHEX data blocks are limited to 42 bytes. In **ghexfile**, the maximum bytes per block can be specified using the command option -length. The default is 80 bytes per block. The maximum bytes per block is 252. The minimum number of bytes per block is 40. There is a certain amount of overhead for each TEKHEX block. Larger block sizes require less blocks in the TEKHEX

file, and thus reduces the overhead and speeds up the time it takes to download the application.

- **Multiple symbols per block**: The symbol formatter was limited to a block size of 80 thus could place several symbols in each symbol block. The symbol blocks produced by **ghexfile** can be up to 252 bytes, and thus contain 3 times as many symbols as the symbol formatter and many times more than the linker. The linker only emitted one symbol per symbol block resulting in delays caused by excessive overhead in downloading TEKHEX files to emulators.
- **Option not to translate data:** Some users download data as S Records and symbols as TEKHEX. The symbol formatter was used to produce a TEKHEX file containing symbols only. The option -nodata exists for backwards compatibility with the symbol formatter.
- Local symbol control: To reduce download time there is an option to not emit local symbols to the TEKHEX output file. Local symbols are useful when debugging, however they extend download time. This has the same function as the linker map file switch -l.
- **Local symbols in the TEKHEX output file are fully supported**. By default, any local symbols in the ELF or COFF input file will be converted to TEKHEX local symbols. Local symbols are present in the ELF or COFF output files produced by the Green Hills compilers in ELF or COFF mode.
- **For assembly language source files, the assembler does not output local symbols to the object file by default (*68K COFF only*).** If the user needs local symbols from assembly language source files, the options -g and -O:DLOCAL must be specified on the command line when assembling.

  For example:

  ```
  a30 file-g-O:DLOCAL=file.asm
  ```
- Unlimited number of symbols per file: There is no limitation on the number of symbols in the TEKHEX output file. In the symbol formatter, a large number of symbols caused symfmt to fail.
- User friendly command interface: On UNIX systems, there is a UNIX-like command syntax. The command options have meaningful names, and as a result, are easier to remember. The -help option will display the command syntax and a summary of the command options with a brief description.
- Option not to translate data in a section: Many users require the ability to discard data in a section. Linker directives can be used to discard data in a section. In some cases, users do not need to download all section data, but

want to avoid doing multiple links, particularly with large applications. The -skip option avoids translating data in the given section.

The ghide Utility Program

The **ghide** Utility Program modifies the symbol table of an object file to convert all global symbols to local symbols, except for a specified *retain list* of global symbols which remain global.

Usage

To use **ghide**, enter:

ghide *retain_list object_file*

where

-help Display information about all options.

*retain_list*File containing symbol names separated by white spaces, which are spaces, tabs, or new line characters.

object_file Output of an assembler or linker. It may be either relocatable or not relocatable, but it should contain a symbol table, otherwise **ghide** will have no effect.

The **ghide** Utility Program modifies the symbol table of file *object_file* so that all global symbols in *object_file* not listed in *retain_list* become local. The *retain_list* file contains the global symbols in *object_file* to be retained.

Example

An embedded application system consists of a kernel and several application tasks, all developed independently. Some global functions in the kernel are for kernel use only. Other global kernel functions can be called from the application tasks. First the kernel is linked into a single file using the linker's **-r** option to retain relocation information. Then, **ghide** is run on the kernel using a list of the functions which should remain visible to the application tasks. Finally, the application tasks are linked with the kernel file, producing a complete executable file. The use of **ghide** ensures that only the desired global symbols in the kernel are visible to the application tasks. This also prevents duplicate symbol errors in the linker if any application should happen to have a global symbol with the same name as an internal kernel-only function.

The gmemfile Utility Program

The **gmemfile** Utility Program reads a fully linked COFF or ELF executable and produces a binary image of the final executable as it would appear in memory. This binary image file is suitable for raw download to a target.

Usage

To use gmemfile, enter:

gmemfile [*options*] *executable_file*

where

executable_file is the name of the COFF or ELF executable from which you wish to generate a download image.

The **gmemfile** *options* include:

-help
: Display information about all options.

-o filename
: Specifies the output filename. If this option is not given, the output will be written to a filename similiar to the *executable_file*, with the suffix ".bin" added (or substituted if *executable_file* includes a "." suffix already). For example, "foo" becomes "foo.bin", and "a.out" becomes "a.bin".

-s
: Use ELF section headers instead of ELF program headers. Applies only to ELF files.

-z
: Write trailing zero-filled sections (e.g., ".bss")

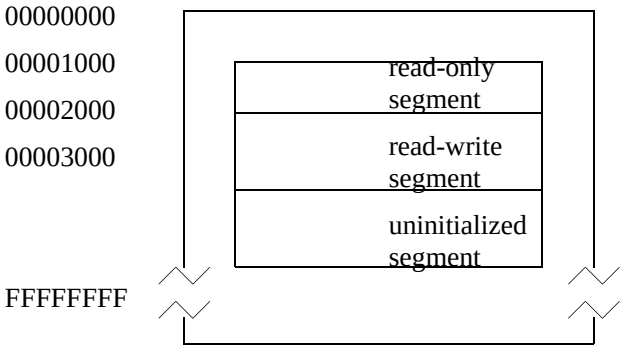
Uninitialized Segments (ELF only)

A segment identified by an ELF program header, which is allocated in the final executable image, but which contains no initialized data is defined by the ELF standard to contain all zeros. The actual clearing of this portion of memory must be performed by the loader, the operating system, or the executable's runtime (e.g., crt0). It is also legal for a segment to be partially initialized. The remaining uninitialized portion of a segment must likewise be filled in with zeros.

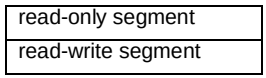
By default, trailing uninitialized segments are omitted from the binary image file generated by **gmemfile**. This can be a big savings in both download time

and file size. However, if your application depends on these segments being zeroed, and no other facility will clear these segments for you (e.g., crt0 or your target operating system), you may specify the -z option to include trailing zeros in your binary image file.

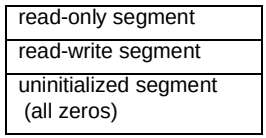
For example, if your program image looks like this:



Then without -z, the output file will look like this:



And with -z, the output file will look like this:



The same type of truncation will occur if the last segment is only partially initialized.

The gnm Utility Program

The **gnm** Utility Program prints the symbol table of an ELF object file, object library file, or executable file created by the Green Hills development tools.

Usage

To use **gnm**, enter:

gnm [*options*] [*files*]

where

-help Display information about all options.

options **gnm** options for ELF are listed below.

files Name(s) of ELF file(s).

ELF File Options

Options when using **gnm** with ELF format files include:

-a Prints special symbols which are normally suppressed.

-h Does not print headers.

-l Prints an asterisk (*) after symbol type for WEAK symbols (**-p** mode only).

-n Sorts output by symbol name.

-o Prints value and size of symbols in octal.

-p Three column output format.

-r Prepends filename to each line of output.

-u Prints undefined symbols only.

-v Sorts output by symbol value.

-x Prints value and size of symbols in hexadecimal.

-V Prints **gnm** version.

Default Output Format

By default, a seven column listing is produced similar to the following:

Index	Value	Size	Type	Bind	Other	Shndx	Name
37	0x00000100	0x00000098	FUNC	GLOBL	0	.text	_ _start
205	0x00000ed0	0x00000000	FUNC	GLOBL	0	.syscall	_ _dotsyscall

Index Position of symbol in the symbol table.

Value The value or address of the symbol.

Size Size of the symbol (e.g. a 4-byte integer would show a size of four)

Type One of the following:

NOTY Typeless symbol.

FILE Filename symbol.

SECT Section name symbol.

OBJT Data symbol.

FUNC Code symbol.

Bind One of the following:

LOCL Local symbol (e.g. C/C++ static).

GLOB Global symbol.

WEAK Weak global symbol (value resolves to zero if undefined).

Other Reserved field, generally zero.

Shndx The name of the section where symbol is defined, for example:

.text Code-defined in the **.text** section.

.data Data-defined in the **.data** section.

ABS No section (e.g. a filename symbol).

COMMON Common variable whose section is not yet determined.

Name Symbol name.

Alternate 3 Column Output Format with -p

The alternate output format has only three columns and is provided for backwards compatibility with tools which can read only this format. This easily parsable format is enabled with the **-p** option:

```
00000000 T __start
         U ___dotsyscall
```

The first column is the value or address of the symbol. The second column is the kind of symbol as shown in the following table. The third column is the name of the symbol.

A External absolute.

a Local absolute.

B External zeroed data.

b Local zeroed data.

C Common variable (same as **B** except not yet assigned to a section).

D External initialized data.

d Local initialized data.

G External initialized SDA.

g Local initialized SDA.

S External zeroed SDA.

s Local zeroed SDA.

T External text.

t Local text.

U External undefined.

In **-p** mode, it is not possible to accurately describe all sections and all storage classes. In particular, user-defined SDA symbols and all other symbols in special sections are shown with letters **SDA = GgSs**.

The grun Utility Program

The **grun** Utility Program remotely executes a program using a MULTI debug server to control the execution environment.

Usage

To use **grun**, enter:

grun [*options*] *dbserv_cmd* **--** *program* [*arguments*]

where

*options*Specific **grun** options, listed below.

*dbserv_cmd*Name of a MULTI debug server.

-- Double dash separates debug server name from program name.

*program*Name of executable program.

*arguments*Optional program command line arguments.

The **grun** *options* include:

-help
Display information about all options.

-text *addr*
Specifies *addr* as the starting address of the program's text (code). This is appropriate for Position Independent Code (PIC) programs.

-data *addr*
Specifies *addr* as the starting address of the program's data. This is appropriate for Position Independent Data (PID) programs.

-stack *addr*
Specifies *addr* as the initial value for the program's stack pointer.

-detach

Causes **grun** to immediately terminate after downloading the program to the target system. **grun** usually engages in communication with the debug server before exiting, which may halt an executing target program. This switch is frequently used with various target monitors or Boot ROMs when the program being downloaded will take the control of the target system and terminate the target monitor.

-download

Causes the program to be downloaded, but does not start it running.

-pro

Like **-profile**, but also translates the profiling data.

-profile

Executes the target program with profiling enabled.

The **grun** utility downloads and starts the executable program, subject to the options above. If the **-bail** flag is specified, **grun** exits immediately after starting the program. Otherwise, **grun** waits for target program completion before exiting. **grun** will wait until 10 minutes with no I/O. After that time, or if **grun** is interrupted by the user, it halts the target program and exits. While **grun** is running, its standard input and output are copied to and from the executing program, redirecting the program's I/O to the user's terminal.

The gsize Utility Program

The **gsize** Utility Program analyzes ELF object files, object library files, or executable files, and for each file displays the size of each section in bytes. If more than one file is named, or if an object library is named, **gsize** prints the name of the file, with the section name and totals for each section.

Usage

To use **gsize**, enter:

gsize *options filename*

where *options* include:

-help
Display information about all options.

-table
Prints the output in a slightly different format.

-nototals
Suppresses the summary information.

-all
Causes empty sections and unallocated sections to be displayed.

filename
The file name of an ELF object file, object file library, or executable file.

The gsrec Utility Program

The **gsrec** Utility Program converts an ELF executable file into a Motorola S-record format file. Motorola S-Records are an ASCII representation of binary data. Many simulators, In-Circuit Emulators (ICEs), PROM programmers and debuggers use S-Records as a program download format.

S-Record Output Format

An S-record file contains ASCII text which can be displayed or edited. There are ten kinds of S-records, numbered from **S0** to **S9**:

- An S0 header record which identifies the program.
- A number of data records which represent the binary data in the program.
- An S5 data count record which contains the count of data records in the S-record file.
- A termination record which contains the address to begin execution of the program.

Not all S-record reader programs behave the same way. Some programs require certain types of data records. Some target environments do not accept S5 data count records and some require certain types of termination records. Options are provided to handle many of these cases.

Usage

To use **gsrec**, enter:

gsrec [*options*] *input_file* [**-o** *srec_file*]

where

*options***gsrec** options, listed below.

*input_file*Name of ELF executable file to be converted.

-o *srec_file***-o** option and a name for the S-record format file.

The **gsrec** *options* include:

-help
Display information about all options.

-auto
Determine byte order from the file header. This is the default.

-B
Input file is Big Endian.

-L
Input file is Little Endian.

-bytes ***n***
Set the maximum count of unpaired data bytes/records (min 4, max 28). The default is 28.

-e ***addr***
Set entry point in the termination record to given address.

-end ***addr***
End address in object file.

-eol cr
Tells **gsrec** to terminate each record with a **\r** character.

-eol crlf
Tells **gsrec** to terminate each record with a **\r\n** combination. This is the default in Windows.

-eol lf
Tells **gsrec** to terminate each record with a **\n** character. This is the default in UNIX.

-fill1 ***n1 n2 v***
Fill memory from address *.n1* to address *n2* with the one byte value *v.*

-fill2 ***n1 n2 v***
Fill memory from address *n1* to address *n2* with the two byte value *v.*

-fill4 ***n1 n2 v***
Fill memory from address *n1* to address *n2* with the four byte value *v.*

-interval ***n*[:*m*]**
Place only those data bytes from the input file which occur within the specified interval in the output file. Legal values for *n* are 1 through 8. The default is one, indicating that every byte will be output. If an *m* value

is specified, it tells how many bytes will be output for each interval. For example, 4:2 outputs two consecutive bytes of every four.

-noS5

Does not produce an S5 block count record.

-o ***filename***

Specifies the output filename. If this option is not given, the S-record will be sent to standard output.

-romaddr *addr*

Start address in ROM to place data. The default is the same address as in the input file.

-S1

Produces S1 data records (16-bit addresses).

-S2

Produces S2 data records (24-bit addresses).

-S3

Produces S3 data records (32-bit addresses). This is the default.

-S5

Produces an S5 record. This is the default.

-S5old

Produces an S5 record with a 16-bit address.

-S7

Produces an S7 end record (32-bit entry point. This is the default.

-S8

Produces an S8 end record (24-bit entry point).

-S9

Produces an S9 end record (16-bit entry point).

-skip s

Does not output data for section *s*.

-start *addr*

Starts address in object file for outputting when **-interval** is used.

Data Record

A data record contains the address where the data is loaded, followed by the data itself. There are 3 varieties of data records: S1, S2, and S3. The only difference between the data records is the size of the load address as follows:

- S3 records contain 32-bit load addresses. This is the default.
- S2 data records contain 24-bit load addresses. Use the option -S2 to get S2 data records.
- S1 data records contain 16-bit load addresses. Use the option -S1 to get S1 data records.

By default an S5 data count record is emitted. If your S-record loader does not understand S5 records, use the option -noS5 to avoid outputting an S5 data count record.

Termination Record

A termination record contains the entry point, the address where program execution begins. There are 3 types of termination records: S7, S8, and S9.

- The S7 record contains a 32-bit entry point. This is the default.
- The S8 record contains a 24-bit entry point. Use the -S8 option to get S8 records.
- The S9 record contains a 16-bit entry point. Use the -S9 option to get S9 records.

Some S-record readers will only accept data records up to a certain length. The option -bytes *size* can be used to set the maximum length of the data records.

If you forget to specify the entry point address when linking, -e *address* can be used to set the entry point to the given *address*.

Data Splitting

In some hardware implementations, the width of the bus differs from the width of the PROMs. Depending on how the bus and PROMs are connected, it may be necessary to store even bytes in one PROM and odd bytes in another PROM. The technique of dividing the data into even and odd bytes, or worse, is called data splitting.

Data splitting can be done in **gsrec** using a few options. The **-start** option specifies the starting address of the data in the object input file to output to the S-record output file. The **-end** option specifies the last address of data in the object input file to output. The **-interval** option specifies the distance between

bytes in the input file to output. A value of 2 for **-interval** outputs every other byte. Sometimes it is necessary to relocate the data to address zero in the S-record output file for programming PROMS. The **-romaddr** option specifies the start address of the data bytes in the S-record output file. Examples 5 and 6 illustrate separating even and odd bytes into two different S-record files.

Examples

All of the examples in this section use the following program file:

file: **prog1**

sections:

1: .text, address: 0x1000, size: 0x100
2: .data, address: 0x2000, size: 0x200
3: .bss, address: 0x3000, size: 0x200 (No Load Section)
4: .data2, address: 0x4000, size: 0x200
The program was linked using the following command:

```
lx -T0x1000 myprog.o -o prog1
```

Given a relocatable input file **myprog.o**, this command causes the linker to create a program file called **prog1** with a **.text** base address of `0x1000`.

Example 1

```
gsrec prog1
```

gsrec translates the data in the input file **prog1** and writes the S-records to the standard output. Since the contents of section **.bss** are not loaded, no S-records are output for its data.

Example 2

```
gsrec -S1 -S9 prog1 -o  prog1.run
```

gsrec translates the data in the input file **prog1** and writes the S-records to the specified output file, **prog1.run**. Data is output as S1 records which have a 16-bit address space. The start address is output as an S9 record which has 16-bit address space.

Example 3

```
gsrec -start 0x1000 -end 0x1080 prog1 -o prog1.run
```

gsrec translates the data in the input file **prog1** starting at address `0x1000` and ending at address `0x1080` to the output file, **prog1.run**.

EXAMPLE 4

```
gsrec -start 0x1000 -end 0x1080 -romaddr 0 prog1 -o prog1.run
```

gsrec translates the data in the input file **prog1** starting at address `0x1000` and ending at address `0x1080`, relocates the data starting at address zero, and copies the resulting S-records to the output file **prog1.run**.

Example 5

```
gsrec -start 0x1000 -end 0x1080 -romaddr 0 -interval 2 prog1
```

gsrec translates the even data bytes in the input file **prog1** starting at address `0x1000` and ending at address `0x1080`, relocates the data to start at address zero, and copies the resulting S-records to the standard output.

EXAMPLE 6

```
gsrec -start 0x1001 -end 0x107F -romaddr 0 -interval 2 prog1
```

gsrec translates the odd data bytes in the input file **prog1** starting at address `0x1001` and ending at address `0x107F`, relocates the data to start at address zero, and copies the resulting S-records to the standard output.

Example 7

```
gsrec -start 0x1002 -end 0x107F -romaddr 0 -interval 4:2 prog1
```

gsrec translates two out of every four data bytes in the input file **prog1** starting at address `0x1002` and ending at address `0x107F`, relocates the data to start at address zero, and copies the resulting S-records to the standard output.

The gstack Utility Program

The **gstack** Utility Program analyzes a program to report the maximum stack size each task may need during execution and the call chain which would produce this maximum stack size.

The **gstack** utility relies on information produced by the Green Hills compilers and by the **mtrans** program. See the *MULTI Reference Manual* for a description of **mtrans**.

To produce the information required for **gstack**, all files comprising the program must be compiled with **-G**.

Usage

To use **gstack**, enter:

```
gstack [prog | prog.sym | prog.dnm]
```

where

prog

prog.sym

prog.dnm Name of the executable file.

The **gstack** options include:

-a Add functions or connections to the call graph. **-a fun1:fun2, fun3** adds **fun1, fun2, fun3** to the call graph with **fun2** and **fun3** being callers of **fun1**.

-c *func*Print all callers of *func*.

-f *func*=*size*Specify or change function stack frame *size*.

-g Print the call graph.

-j Print all functions and frame sizes.

-s *func*Print the maximum stack size for the program starting with *func*.

-u Print all functions who have no callers.

-help Print information on all options.

Example

For the file **test.c**:

```
main()   { int arr[1000];fun1();fun0();fun2();
         fun3(); }
fun0()   { fun1(); fun3(); }
fun1()   { int arr[20]; fun2(); fun3(); }
fun2()   { int arr[10]; fun3 (); }
fun3()   { }
```

```
% ccmcore test.c -G -Xstaticcalls
% gstack a.out main
```

Task main 4136 byte stack produced by the call chain of:

Framesize	Function
4004	main
4	fun0
84	fun1
44	fun2

Caveats

- **gstack** cannot work if there are potential direct or indirect recursive calls in the program, as it cannot predict how many times the recursion will occur. **gstack** will print a warning message if it detects a possible recursion in the call graph.
- **gstack** prints a warning message if it detects a call to a function for which there is no stack frame size information, or for which there is no call information. Both of these situations might be caused by compiling the function without using the **-G** option, or the function may be an assembly language routine.
- **gstack** does not understand function calls through pointers. No warning is printed.
- **gstack** cannot differentiate between static functions of the same name from different files.

The gstrip Utility Program

The **gstrip** Utility Program can remove line number, symbol table, and debug information from an ELF executable file to reduce its file size on disk.

The **gstrip** utility processes executable files created by the Green Hills Software linker as well as those created by some native linkers.

Usage

To use **gstrip**, enter:

gstrip [*options*] *filename*

where

*options***gstrip** ELF options, listed below.

*filename*Name of executable file.

ELF File Options

Options when using **gstrip** with ELF format files include:

-help
Display information about all options.

-l
Strips line number information only; do not strip the symbol table or debug information.

-V
Prints the version number of **gstrip** to standard error output.

-x
Does not strip the symbol table; debugging and line number information may be stripped.

The gsymdump Utility Program

The **gsymdump** Utility Program prints symbolic debug information from a **.dbg** file or a **.sym** file to the standard output.

The Green Hills compilers can create files with the file extension **.dbg**. These **.dbg** files contain the symbolic debug information for the object file with the corresponding basename (**cookie.dbg** contains the symbolic debug information

for file **cookie.o**). The **mtrans** Utility Program reads **.dbg** files and produces **.sym** files, which are read by the MULTI Debugger.

Usage

To use **gsymdump**, enter:

gsymdump [*options*] [*object-file***.dbg**] | *executable***.sym**

where

options **gsymdump** options, listed below.

object_file**.dbg**

Name of file with symbolic debug information.

*executable***.sym**

Name of file to be read by MULTI Debugger.

The **gsymdump** *options* include:

-help

Display information about all options.

-c

Perform internal consistency checks.

-C

Perform internal consistency checks, but ignore warnings.

-da

Dump raw auxiliary table.

-dc

Dump static call information.

-dd

Dump the **#define** table.

-df

Dump the file name table.

-dh

Dump the **.sym** file header.

-dp

Dump the proc table.

-ds

Dump the symbol table.

-dt

Dump the **typedef** table.

-dx

Dump the section table.

-nx

Do not demangle C++ names.

-sx

Shorten long C++ demangled names.

-v

Perform internal consistency checks only. Do not display symbol information.

-V

Same as **-v**, but ignore warnings.

If no options are specified, then all of the above information will be displayed with the exception of an auxiliary table. C++ names will be demangled.

The gtune Utility Program

The Green Hills **gtune** Utility Program can tune your program automatically. It generates compiler options to inline functions that are only called once and to delete functions that are never called.

gtune uses cross reference information produced by the compiler and stored in **.dbg** and **.sym** files. In order for the compiler to produce this information, you

must either compile your source code with the **-G** options on the compiler command line. **gtune** requires debugging to be turned on.

Usage

To use **gtune**, enter:

gtune *executable*[**.sym**] [**-help**] [**-a**] **[-k** *func*] ... [**tune.opt**]

where

*executable***.sym**Executable file name and suffix for the **.sym** file associated with it.

-help
Display information about all options.

-a
Considers all functions in tuning output, regardless of whether the call graph information exists for each function.

-k func
Does not include *func* in tuning the output.

tune.opt
Writes tuning output to file *tune.opt*, or to *stdout* if *tune.opt* is not specified.

-t
Do not inline functions.

The **gtune** output file has lines of the form:

```
-OI=func1          (inline function func1)
-OD=func1                    (deletes the out-of-line copy of function func1)
-OD=func2                    (deletes function func2)
```

The output file is in a format suitable to be used as input to the compiler driver, as **@tune.opt**. The **@** character tells the compiler driver to expand **tune.opt** as if the contents of the file had been explicitly entered on the compiler command line.

Example

File **test.c**

```
void unused() { printf (“nobody calls me\n”); }
void unused_also() { printf (“nobody calls me either\n”); }
int inline_me_too() { return 7; }
int inline_me() { return 6 + inline_me_too(); }
main() { printf (“%d\n”, inline_me()); }
```

First, compile the program with call graph information, by entering:

```
% gcc -G test.c
```
Then, run **gtune** to produce the options for the compiler optimizer, by entering:

```
% gtune a.out tune.opt
```
The contents of **tune.opt** will be:

```
-OI=inline_me_too
-OD=inline_me_too
-OD=unused_also
-OD=unused
-OI=inline_me
-OD=inline_me
```
Now compile the program using the information generated by **gtune**, by entering:

```
% gcc test.c @tune.opt
```
As a result, the functions **unused** and **unused_also** are deleted from the program, since they are never referenced. The functions **inline_me** and **inline_me_too** are compiled inline, that is, no call is made, since only one call is made to each of these functions. The out-of-line copy of each of these functions is deleted.

The gversion Utility Program

The **gversion** Utility Program extracts and prints date and time information from an executable file. By default, **gversion** prints out the revision date and the release date of the program.

Usage

To use **gversion**, enter:

gversion [*-all*] [*slot#*] [*file1*] [*file2*] ...

where

-all
: Print all non-zero dates, marked [0]..[9].

slot#
: Single digit number of *slot* to print. The default is to display all time slots. In the examples on the following page, since **ccommcore** only has one time slot set and the slots are filled in increasing numerical order, the command **gversion 0 ccommcore** yields the same result as **gversion ccommcore** which is

```
ccommcore:    Green Hills Software, release 1.8.8
```

```
ccommcore:      Revision Date Fri Dec 19 13:14:29 1997
```

However, if asked to print the value of slot 1 which is unstamped, **gversion** will display an appropriate error message:

```
ccommcore: Green Hills Software, release 1.8.8
ccommcore has not been time stamped
ccommcore:          [m] Fri Dec 19 13:14:30 1997
```

Note that **gversion** will display the file modification date for informational purposes when no valid time stamp corresponding to the slot requested can be found.

In **-all** mode, each date will be preceded by a digit in square brackets called a time stamp. The Revision Date is preceded by **[0]** and the Release Date is preceded by **[6]**. For example,

```
/usr/green/lx:  [0] Thu Jul 06 22:48:50 1996
/usr/green/lx:  [6] Mon Aug 14 14:32:11 1996
/usr/green/lx: Revision Date Thu Jul 11 21:45:45 1996
/usr/green/lx: Release Date Mon Oct  7 07:47:37 1996
```

The Revision Date reflects the date of the last modification of the source code that was used to build the program. The Release Date is the date that the particular tape or disk image containing the program was created. All programs on the same tape should have the same Release Date.

Example 1

```
gversion /usr/green/lx
/usr/green/lx: Revision Date Thu Jul 11 21:45:45 1996
/usr/green/lx: Release Date Mon Oct  7 07:47:37 1996
```

Example 2

```
gversion all /usr/green/gversion
/usr/green/lx:  [0] Thu Jul 06 22:48:50 1996
/usr/green/lx:  [6] Mon Aug 14 14:32:11 1996
```

The **gversion** options include:

-help

Display information about all options.

-date

Print the value as a date string. This is the default.

```
gversion ccommcore
ccommcore:    Green Hills Software, release 1.8.8
ccommcore:      Revision Date Fri Dec 19 13:14:29 1997
```

-value

Print the value without converting to date. For example:

```
gversion -value ccommcore
ccommcore: Green Hills Software, release 1.8.8
ccommcore: [0] 882566069
ccommcore: [1]
ccommcore: [2]
ccommcore: [3]
ccommcore: [4]
ccommcore: [5]
ccommcore: [6]
ccommcore: [7]
ccommcore: [8]
ccommcore: [9]
```

-mtime

Always print the file modification date. This option must be used with **-all**. For example:

```
gversion -mtime -all ccommcore
ccommcore: Green Hills Software, release 1.8.8
ccommcore:          [0] Fri Dec 19 13:14:29 1997
ccommcore:          [m] Fri Dec 19 13:14:30 1997
```

-nomtime

Never print the file modification date.

-quiet

Suppress errors for unstamped files.

```
gversion foo
foo cannot be time stamped

gversion -quiet foo
```

15

Runtime Environment and Library Organization

This chapter contains:

- Introduction
- Multiple Language Runtime Support
- MCore Library Structure
- Linker Directives Files
- How to Create a Customized Linker Directives File
- Special Sections in Linker Directives Files
- Source Files Available for Customization
- Incorporating Your Changes into the Libraries

This chapter describes the basic structure of the Green Hills runtime environment and how to modify and customize it. This chapter also describes changes from previous versions of the Green Hills toolset.

Introduction

In an embedded environment, runtime support requirements differ, depending on the application and target configuration. For example, if user C code calls high level I/O routines such as **printf**, **scanf**, or **malloc**, then low-level system call routines **write**, **read**, and **sbrk** need to be implemented, perhaps with modifications appropriate to your specific hardware environment. Green Hills provides a default implementation for these interfaces which work with the Green Hills target environments; for example, Green Hills instruction set simulators, ROM monitors, or real-time operating systems.

The Green Hills version 1.8.9 cross development library system includes two low-level libraries, **libsys.a** and **libarch.a**. The **libsys.a** library contains low-level system routines which implement memory initialization and system calls. The **libsys.a** library automates the function of copying sections of a program's initialized data from ROM to RAM. It automatically relocates initializers which are runtime located (PIC/PID). The **libarch.a** library contains utility routines specific to a particular architecture; for example, **libarch.a** might include an integer divide routine to be used by the compiler system when a processor architecture does not specify an integer divide instruction. The startup module, **crt0.o**, performs some basic initializations.

The source code to **libsys.a** and **crt0.o** is fully contained in a separate directory, **libsrc**, under the usual installation directory (for example, **/usr/green**) of a Green Hills cross development distribution. This directory also contains example project build files to build these libraries using the MULTI Development Environment. This source code, as well as the low-level library object modules themselves, are a guide for development of an embedded system.

While these object modules and object libraries enable you to get up and running quickly under the initial default Green Hills configuration, you may want to omit or modify these low-level interfaces for a particular system. Also, not all functions contained in the libraries have default implementations; a few routines, such as fstat, are entry points for user-supplied implementations, if needed.

Multiple Language Runtime Support

The Green Hills C Library implements all the standard ANSI C library functions. For a complete list of these supported functions, please refer to the Appendix. The libraries for other Green Hills languages, C++, Ada, FORTRAN, and Pascal provide full support for those languages, including:

- ANSI and K&R C
- ANSI C++ and EC++
- MILSTD FORTRAN DoD supplement to ANSI X3.9-1978 with VMS FORTRAN extensions
- ISO Pascal Level 1
- Ada 95

MCore Library Structure

Each Green Hills cross development toolset includes a set of runtime library directories that are located beneath the main install directory. For example, if your main install directory is **/usr/green** (or on a PC, **C:\green**), then you may have a set of library directories **/usr/green/***directname*, where *directname* can be:

mcore MCore ELF

mcore_sdMCore ELF with single precision hardware floating point.

The compiler driver option **-fsingle** changes the library directory from **mcore** to **mcore_sd**.

Linker Directives Files

The linker directives file contains a description of the program sections. The following example linker directives file will be referred to in the descriptions that follow:

```
-sec
{
   .text                    :
   .syscall                 :
   .secinfo                 :
   .fixaddr                         :
   .fixtype                         :
   .rodataalign(4):
```

```
    .romdata    ROM(.data):
    .romsdata   ROM(.sdata):
    .sdabase align(4) :
    .sbss                   :
    .sdata              :
    .rosdata                :
    .data       :
    .bss                    :
    .heapalign(4) pad(0x100000):
    .stackalign(4) pad(0x80000):
}
```

How to Create a Customized Linker Directives File

It is strongly suggested that you consult the existing default linker directives examples provided in the target library directories of the distribution. When you do not specify a linker directives file for a program build, these default directives files will be used. The default linker directives filenames are:

default.lnk Used for normal, absolutely-located programs

pic.lnk Appropriate for PIC (Position Independent Code) programs

pid.lnk Appropriate for PID (Position Independent Data) programs

picpid.lnk Appropriate when using both PIC and PID

Not all target library directories will contain all of these linker directives files, although **default.lnk** exists in the default target library directory for all targets. A suggested method for customizing a linker directives file is to copy the default link map and make appropriate changes to it, such as altering the addresses of text and data, specifying additional user-defined sections, changing the sizes of the stack (via **.stack**) and heap (via **.heap**) areas, specifying different sections to be placed in ROM, and removing unnecessary sections. Before removing a section from the linker directives file, please read the following section on Special Sections. Starting from a default section map reduces the risk of failing to include a required section.

Special Sections in Linker Directives Files

Several sections must reside in the linker directives file for proper operation. A list of these sections, a brief description of their functions, and an indication of when they might not be needed follows. More detailed descriptions are located throughout the rest of this chapter and in the **libsrc** source code.

.heap Specifies the size and location of the runtime heap. It is required when using the Green Hills runtime libraries for dynamic memory allocation. A reference to the predefined linker symbol

_ _ghsbegin_heap which denotes the beginning of the **.heap** section is located in the **ind_heap.o** module of the **libsys.a** library. If the Green Hills runtime libraries are not being used at all, then the **.heap** section can be omitted. Otherwise, failure to include a **.heap** section in the linker directive specification when the program uses dynamic memory allocation (i.e. when those library modules handling this feature are linked in with the program) will cause one or more linker errors such as the following:

```
[elxr] error: undefined: '__ghsbegin_heap' referenced in
'/usr/green/mcore/libsys.a(ind_heap.o)'
```

The Green Hills runtime libraries ensure that the dynamic memory allocation will not overrun the specified **.heap** area by returning an error code from **sbrk** which, in turn, will cause **malloc()** and other memory allocation routines to return a NULL pointer. User code should always check for erroneous return values when calling dynamic memory allocation routines. Also, the Green Hills **sbrk()** routine, located in **ind_heap.c**, can be customized to avoid use of the **.heap** section.

.sdabase Required when not using PIC/PID. Essentially, for all normal, absolutely located programs, the **.sdabase** section MUST occur in the linker directives files. The Green Hills debug servers will likely have to initialize a PID/SDA base register before execution of a normal program can begin. Debug servers will initialize this base register with the address of the **.sdabase** section (a dummy, zero-sized section). Failure to include an **.sdabase** section will cause the base register to be left uninitialized and many programs to fail. If the target does not support PID or SDA, or if customized user code startup code sets up the base register, then the **.sdabase** may not be needed but should still be included for completeness. The **.sdabase** section is typically located just prior to the start of the sections that make up the SDA (typically the small data area consists of the **.sdata** and **.sbss** sections). When small data area is being used, location of the **.sdabase** is critical since only small offsets from the SDA base register are typically allowed for addressing data within the SDA. If data in the SDA resides too far from the **.sdabase** section, a linker relocation overflow error will result. If SDA is not being used, then location of the **.sdabase** may not be significant since PID references generally allow a full 32-bit offset from the base register.

.stack This section specifies your intended stack area and size, although there is no general mechanism for ensuring that the stack stay within the limits specified by the **.stack** section. Green Hills startup code (when running in standalone mode) and debug servers will initialize the initial stack pointer of a process based on this **.stack** section. When building programs to run under an operating system which does not allow user-specification of the stack area, an empty **.stack** section can still be included in the section map in order to resolve references in the startup code. The reference to the **.stack** area in the Green Hills startup code (**crt0.o**) is used only when programs execute standalone, that is, not downloaded and run with the MULTI Debugger. Use of the **.stack** section in the startup code is also customizeable by editing and rebuilding the **crt0.mco** assembly module. The Green Hills debuggers also allow the initial stack pointer to be specified differently with each download of a program via the special **_INIT_SP** variable; use of **_INIT_SP** supersedes use of a **.stack** section to locate the initial stack pointer.

.**romdata**

.romsdata Section names beginning with **.romXXX** reference sections that are the shadow copies of initialized data sections for romming code (and hence have **ROM()** directives as well). Many of the Green Hills **default.lnk** section maps use ROM sections even though the entire program may be downloaded to RAM, allowing you to see examples of how to specify ROM sections. When the linker sees a ROM directive, it will create a copy of the RAM section that is readonly and hence rommable. Then, the Green Hills startup code (**ind_crt0.c**) will automatically copy the data from the **.romXXX** section to the **.XXX** section as needed when copying initialized data from ROM to RAM.

If the ROM linker directives are used, but the Green Hills automatic ROM-RAM initialization code is not desired (i.e. custom ROM-RAM copy code is used), then you should ensure that the ROM-RAM copy mechanism in **ind_crt0.c** is disabled by modifying or deleting that code.

If no ROM-RAM copying of code is necessary at all, then the .romXXX sections can be omitted from the section map.

The **.romXXX** names are a convention used in the default section maps. Names can be arbitrary. The **ROM()** directives provide the automatic romming capability. You can find the actual code that accomplishes the automatic ROM copying by looking in the **ind_crt0.c** file.

.syscall Text section containing runtime library code for Green Hills emulation of system calls. This is a rommable section since it contains only text.

When using the Green Hills runtime libraries for system call emulation, this section is required. The **.syscall** source code (assembly) is located in the file **ind_dots**. When system call emulation is not being handled by the **.syscall** mechanism, use of the **.syscall** code in this module can be removed from the program (by customizing the **libsys.a** library which contains the **.syscall** code), or a dummy **.syscall** section can be included in the section map to prevent it from being appended to the end of the section map by the linker. If the **.syscall** section is empty, it should be left out of the section map.

.secinfo This is a special section output by the Green Hills linker. It contains information on the section layout of programs. The startup code in **libsys.a** (**ind_crt0.c**) uses this information to determine which sections need to be cleared (**bss** sections) and which need to be copied (ROM sections). This is a read-only and thus a rommable section. Failure to include this section in the link map will cause the linker to append it to the end of the section map. There are three flags that can be used here. **x** means executable, **a** means allocate it to be downloaded, and **w** means writable. For example:

```
.section ".far", "aw"
.section ".bar", "ax"
```

Consult the **ind_crt0.c** file for further documentation on the automatic **copy/clear** feature.

.rodata Many of the Green Hills compilers will place readonly data (such as data declared with the C **const** specifier and string constants) into a read-only section called **.rodata**. The convention used in the default linker directives files is to place **.rodata** with other rommable sections. Failure to include a **.rodata** section in the section map may cause the linker to append it to the end of the section map.

With the exception of **.rodata** and the **.romXXX** ROM sections, the special sections described above are created for and maintained by the Green Hills runtime environment system. You should not explicitly add to them. Contents of these sections are generated by one of the following methods:

- compiler (**.fixtype**, **.fixaddr**)
- linker (**.secinfo**)
- runtime library code (**.syscall**)
- linker directives (**.sdabase**,**.stack**, **.heap**)

Any attempt to explicitly place text or data into these special sections will produce undefined and potentially fatal results. When creating custom named sections, you must take care to not use any of the names of these special sections.

More detailed descriptions of the functionality corresponding to the extended linker directives and special sections can be found in the next section, Source Files Available for Customization, as well as in the source code located in the **libsrc** directory.

Source Files Available for Customization

With the Green Hills runtime libraries, you need to customize just a few low-level source files and routines to implement or enhance the runtime environment for a particular hardware system. A description of each file follows. Each source file, including files with only assembly language, are fully commented to provide more detailed documentation.

The linker directives files, which specify the location and, sometimes, the size of program sections, are also improved. Some of the improved functionality in the low-level libraries require use of some of these linker directives.

crt0.mco

The startup module **crt0.mco** is assembled from a small architecture-specific assembly language file and contains only a minimal amount of code to setup a C program environment before calling into a high level language C function. This minimal assembly code is located in a function named **_start**, the default entry point for programs in the Green Hills environment. On program startup, an initial system call is made to determine whether a Green Hills debug agent (or debug server) is controlling execution of the program (as opposed to the program running standalone, without being connected to a debug server). If the system call is successful, some required register initialization is assumed to have been accomplished by the debug server; otherwise the code in **crt0.mco** may do some more initialization. The source code in **crt0.mco** should be consulted since the actual register manipulation varies widely across different architectures.

The most commonly required register initialization is that of the processor stack pointer, since a valid stack is generally required before a high level language can be called. Most **crt0.o** modules reference the special symbol **_ _ghsend_stack** when initializing the stack pointer. **_ _ghsbegin_***section* and **_ _ghsend_***section*, where *section* is the name of your section preceded by an

underscore (_) and not a period (.), are special linker-defined symbols which reference the respective start and end of each user program section.

The **.stack** section is an improved method for specifying the location of the runtime stack. You can place a **.stack** section into the linker directives file to specify the location and size of the runtime stack. The Green Hills startup code sets up the stack pointer to point to the end of this section if the stack grows downward. Otherwise the stack pointer is written with the address of the start of the **.stack** section. You can change the location or size of the stack by changing the linker directives file.

Green Hills debug servers automatically detect the existence of the **.stack** section and automatically initialize the stack pointer. An example **.stack** section specified in the preceding example directives file is:

```
.heap  align(16) pad(0x100000) :
.stack  align (16) pad(0x80000):
```

This specifies that the stack starts on the first 16-byte aligned address following the **.heap** section in memory. Also, the stack is configured to be 0x80000 bytes in size.

Using the **.stack** to specify the stack configuration permits stack checking code is to be easily built into a program; code need only compare the current value of the stack pointer register with the value of **_ _ghsbegin_stack** to determine whether the available stack area is exhausted.

Finally, **_start** calls the function **_ _ghs_ind_crt0**, the architecture-independent startup routine located in **libsys.a**.

ind_crt0.c

As described above, this module contains machine-independent startup code. In particular, **ind_crt0.c** clears the uninitialized data sections (**.bss** and **.sbss**, if present), and copies initialized data sections from their location in ROM to their final location in RAM. Whether a program requires this ROM to RAM copy depends on the use of the linker directives file for that program. In the example linker directives file above, the ROM directives specify that the sections **.romdata** and **.romsdata** are “shadow” or ROM copies of the actual **.data** section in RAM. The ROM sections are copied to RAM by the startup code in this module. You do not have to write code to clear or copy sections at startup; **ind_crt0.o** does it automatically. The **ind_crt0.c** module should be customized and rebuilt should you want to do this initialization without the aid of the library.

When your program requires PIC/PID, the **ind_crt0.o** module will relocate initialized pointers to any position independent object. For example, consider the following C code:

```
extern int foo();
int (*ptr)() = &foo;
```

This declares a global function pointer which is initialized to the address of the function `foo`. This declaration is not valid with other compilers when utilizing position independent code because the address of `foo` is unknown at link-time and hence unresolvable by the linker/loader. Green Hills, however, supports this. The compiler emits a small amount of data which describes each relocated initialization in the program. For example, when compiling for PIC, the compiler generates data to inform **ind_crt0.o** that the initializer of the variable `ptr` requires a runtime modification specifying the runtime location of the program's code, as desired. After **ind_crt0.o** finishes, all initialized pointers contain valid runtime addresses.

Finally, **ind_crt0.c** calls into the user code, that is, to **main()**.

ind_call.mco

ind_dots.mco

These modules contain architecture-specific language and handle lowest level system call capability. The routine **_ _ghs_syscall** is called from various system call routines, such as **open**, **close**, **read** and **write**. The **_ _ghs_syscall** routine transfers control to a special address, the start address of the special **.syscall** section (see the earlier linker directives example.).

Debug servers or monitors can key in this special address to accomplish the emulation of system calls in the Green Hills environment. For example, many MULTI debug servers set a special breakpoint on this address; then, when the breakpoint is encountered during execution, the debug server knows that a system call occurred. The arguments are then retrieved and the system call is emulated on the host by the debug server. This mechanism provides a more generic system call interface and brings system call capability to some targets where this functionality was previously unavailable.

ind_mcpy.c

ind_mset.c

These modules contain the C runtime library functions **memcpy** and **memset**. They are needed to clear and copy data sections during initialization. You can

modify and change these routines if desired. For some architectures, either or both of these functions are provided in assembly code. On other architectures, high level C functions are used and compiled into the **libsys.a** library with the highest optimization level.

ind_mcnt.mco

ind_gcnt.mco

ind_bcnt.c

ind_mprf.c

ind_gprf.c

These modules provide profiling support. Two routines are used: **__ghs_mcount** for call count profiling and **_ _ghs_gcount** for call graph profiling. The module **ind_bcnt.c** contains the profiling routine, **__ghs_bcount**, which handles calls that are emitted by the compiler to implement code coverage analysis. The **ind_mprf.c** and **ind_gprf.c** modules contain routines that accomplish profiling functions such as starting a profiling timer (only on certain processors) and writing profile data to disk.

ind_heap.c

The **ind_heap.c** module contains the dynamic memory allocation routines, in particular the system call routine **sbrk**. The **.heap** section specifies the location, size, and alignment of the heap in the linker directive file.

For example, in the preceding linker directives example, the heap section is specified as follows:

```
. . .
.bss :
.heap align(16) pad(0x100000) :
.stack align(16) pad(0x80000) :
. . .
```

This specifies that the heap starts on the first 16-byte aligned address following the **.bss** section in memory and is `0x100000` bytes in size. The **.stack** section then follows the heap area. The pad directive instructs the linker that the heap is uninitialized on startup. This directive enables you to place the runtime heap anywhere in memory; the runtime library automatically allocates heap memory where you place this section.

In addition, the ability to hardcode a size for the heap assures you that the program does not use more heap memory than desired or expected. Attempting to allocate memory at an address which is higher than that specified by the size in the linker directive will cause **sbrk** to fail and return an error value.

ind_io.c

ind_io.c, **ind_gmtm.c**, **ind_sgnl.c**, **ind_stat.c**, **ind_time.c**, **ind_exit.c**, **ind_tmzn.c**, **ind_renm.c**, and **ind_syst.c** contain a basic set of UNIX-like operating system routines. Each routine is documented to show the function it performs and the values it returns. These routines allow the linker to resolve an operating system's symbols, referenced by the Green Hills runtime libraries. The **ind_io.c** system routines most likely to be needed for a basic UNIX-like implementation are:

```
int open(const char *filename, int mode, . . . );
int creat(const char *filename, int prot);
int close(int fno);
int read(int fno, void *buf, int size);
int write(int fno, const void *buf, int size);
int unlink(char *name);
void _exit(int code);
```

These system calls enable the embedded system program to open, read, write, and close files. In many embedded systems, this basic support is not required. For example, on some systems, the application never ends and hence does not need an exit routine.

Other system calls, enabling a more robust interface and used in the default Green Hills runtime environment, include:

```
int brk(void *addr);
void *sbrk(int size);
long lseek(int fno, long offset, int end);
int fcntl(int fno, int cmd, int arg);
int getpid(void);
int isatty(int fno);
void _enter(void);
```

The **_enter** routine is called at startup and can be used as a separate routine to initialize the I/O system, initialize caching options, etc. Also, a few of the above listed routines only provide rudimentary support; you should consult the implementation before using them. Many of the Green Hills **ind_io.c** and related system call modules filter down to calls to the generic system call interface routine, **_ _ghs_syscall**, described above.

ind_exit.c

_enter Called up at startup and is used as a separate routine to initialize the I/O system, initializing caching options, etc.

_ghs_at_exit Registers functions that need to be called upon **_exit** (similar to the ANSI **atexit()**.)

_exit Calls cleanup routines registered by the program via

_ _ghs_at_exit() or **atexit()** and allows the program to terminate gracefully via an exit system call.

FORTRAN Runtime Support

The system routines needed specifically for FORTRAN are tabulated below. These routines are used by the Green Hills FORTRAN runtime library, **libf.a**. Descriptions of the routines are in the noted source files.

Source File	Routine
ind_trnc.c	int truncate(const char *path, int length);
ind_stat.c	int fstat(int fno, struct stat *statptr); int stat(char *name, struct stat *statptr);

Other Low-Level Functions

Source File	Routine
ind_gmtm.c	struct tm *gmtime(const time_t *timer);

ind_sgnl.c	int raise(int sig);
	void (*signal(int sig, void (*func)(int)))(int);
	unsigned int ualarm(unsigned int value, unsigned int interval);
	unsigned int alarm(unsigned int seconds);

ind_stat.c	int access(char *name, int mode);
ind_time.c	time_t time(time_t *tptr); int times (struct tms *buffer);

ind_tmzn.c	struct tm *localtime(const time_t *timer);
	void tzset(void);
	int __gh_timezone(void);

ind_renm.c	int rename(const char *old, const char *new);
ind_syst.c	int system(const char *string);

Incorporating Your Changes into the Libraries

Included in each MCore library directory is a **default.bld** file to help you recompile and replace the customized object modules. For example, after

changing directories into one of the target library directories, do the following steps:

1. Create a subdirectory **objs**, if it has not already been created.
2. Rename or remove the file (**libsys.a** or **crt0.o**) that you wish to rebuild. It might be useful to keep the old file around as a backup.
3. Run **multi**. It should open up on **default.bld**.
4. Remove the unused **libghs.bld** from project by single clicking on it and hitting the **remove** button, if this has not already been done.
5. Enter either the **libsys** or **crt0** project with a double click.
6. Click on **build**. The library should be built.
7. If you rebuilt **crt0.o**, it should be copied from out of the **objs** subdirectory back into the target directory. On UNIX systems, you may, instead, create a symbolic link with **ln -s objs/crt0.o**.

Consult the *MULTI Reference Manual* on using the Builder to change options and rebuild. Options should be changed with care, as some of them are required for proper operation. For example, the **default.bld** project causes the preprocessor symbol `EMBEDDED` to be defined when compiling; this symbol is required for some modules in the libraries. Under **default.bld**, the subproject **libsys.bld** builds the **libsys.a** library and **crt0.bld** builds the startup module, **crt0.o**. The **libghs.bld** project should be ignored.

16

MCore Simulator

This chapter contains:

- The MCore Simulator Command Line Options
- The Simulator as a MULTI Debugger Target
- ROM Mode
- Unsupported Features

You can use **simmcore** as a standalone simulator to run MCore programs or use it as a debugging target with MULTI to interactively debug MCore programs. See the *MULTI Reference Manual* for more information on the MULTI Debugger.

The MCore Simulator Command Line Options

Command Line Options

You can run the simulator from the command line using the following syntax:

simmcore [*options*] *image-file*
where *image-file* is an ELF style executable file.

The **simmcore** *options* include:

-help
Prints the options list.

-fpu
Enables hardware floating point support.

-ieee
Enables IEEE-style operating mode with floating point exception handling.

-V
Print version of simulator.

The Simulator as a MULTI Debugger Target

The simulator can be used in conjunction with MULTI to interactively debug programs that may be difficult to debug. Invoking the simulator either from the command line or as a result of the MULTI **remote** command, will bring up two other windows, one labeled **TARGET** and the other labeled **IN/OUT**.

The IN/OUT window is where all output to **stdout** goes, and all input from **stdin** comes from. This separates communication with the remote process from communication with the simulator itself.

The TARGET window provides an interface to the simulator. Target commands provide a comprehensive view of the internal state of the simulated processor. Some of these commands are only useful in the simulator's ROM mode. The

commands currently available in the MCore simulator are listed in the table found below.

Command	Meaning
help	Show all of commands with their brief descriptions.
reset	Reset processor manually.
softreset	Send soft reset signal to processor.
intr *num*	Send interrupt *num* to processor.
fintr *num*	Send fast interrupt *num* to processor.
show timing	Show number of cycles executed.
zerotime	Reset cycle time counter.
timer *count int*	Send interrupt *int* every *count* cycles.
alloc *addr size*	Allocate *size* bytes of memory starting at address *addr* in memory.

All of these commands can also be reached from the MULTI Debugger command pane by using the **target** command.

OS Simulation Mode

The simulator is able to deal with number of system calls. The following table lists these system calls.

exit	close	stat	link
read	access	fstat	unlink
write	creat	brk	getpid
open	lseek	time	alarm

A large portion of library functions use a combination of these calls to achieve their goal. These system calls are unavailable to you in ROM simulation mode.

ROM Mode

ROM mode was added to the Green Hills simulators to facilitate creation of embedded programs for real-time systems. The MCore simulator in ROM mode will not simulate various system calls available in OS simulation mode. Instead, only the minimum hardware is simulated, such as the CPU and memory systems.

In ROM mode the executable is loaded into simulator memory using the physical addresses specified in the object file instead of virtual addresses. Then execution begins at the address stored in the vector table (0x0), as if the CPU

were just turned on. Thus, the object file acts like programmable ROM where your system code can reside.

Other features of ROM mode:

- Exception handling is left to the user.
- All memory pages are allocated on use. No segmentation faults are generated.
- No arguments are passed to the running program.
- Stack can be set anywhere desired by user (the value stored at 0x4 is used initially).

ROM mode is useful for writing and testing exception handlers and parts of an operating system. Also, you will have to write at least some start-up code in pure assembly language since this mode gives you such a bare-bones processor.

Unsupported Features

The simulator does not support the following hardware features:

- instruction pipelining for external memory
- low power mode
- instruction timings

A

Enhanced asm Facility

This appendix contains:

- Introduction
- Definition of Terms
- asm Macros
- MCore asm procedures
- Writing asm Macros

Introduction

Although the ability to write portable code is one reason for using the C language, sometimes it is necessary to introduce machine-specific assembly language instructions into C code. This need arises most often within operating system code that must deal with hardware registers that would otherwise be inaccessible from C. The **asm** facility makes it possible to introduce this assembly code.

In earlier versions of C, the **asm** facility included a line that looked like a call on the function **asm**, which took one argument, a string:

```
asm("assembly instruction here");
```

Unfortunately this technique has shortcomings when the assembly instruction needs to reference C language operands. You have to guess the register or stack location into which the compiler would put the operand and encode that location into the instruction. If the compiler's allocation scheme changed, or, more likely, if the C code surrounding the **asm** changed, the correct location for the operand in the **asm** would also change. You'd have to be aware that the C code would affect the **asm** and change it accordingly.

The new facility presented here is upwardly compatible with old code, since it retains the old capability. In addition, it allows you to define **asm** macros that describe how machine instructions should be generated when their operands take particular forms that the compiler recognizes, such as register or stack variables.

Although this enhanced **asm** facility is easier to use than before, you are strongly discouraged from using it for routine applications because those applications will not be portable to different machines. The primary intended use of the **asm** facility is to help implement operating systems in a clean way.

The optimizer (**ccmcore -O**) may work incorrectly on C programs that use the **asm** facility, particularly when the **asm** macros contain instructions or labels that are unlike those that the C compiler generates. Furthermore, you may need to rewrite **asm** code in the future to maximize its benefits as new optimization technology is introduced into the compilation system.

Definition of Terms

asm macro An **asm macro** is the mechanism by which programs use the enhanced **asm** facility. The **asm** macros have a definition and uses. The definition includes a set of pattern/body pairs. Each pattern describes the

storage modes that the actual arguments must match for the **asm** macro body to be expanded. The uses resemble C function calls.

storage mode

The **storage mode**, or **mode**, of an **asm** macro argument is the compiler's idea of where the argument can be found at run-time. Examples are "in a register" or "in memory."

patternA **pattern** specifies the modes for each of the arguments of an **asm** macro. When the modes in the pattern all match those of the use, the corresponding body is expanded.

asm macro body

The **asm macro body**, or **body**, is the portion of code that will be expanded by the compiler when the corresponding pattern matches. The body may contain references to the formal parameters, in which case the compiler substitutes the corresponding assembly language code.

asm Macros

The enhanced **asm** facility allows you to define constructs that behave syntactically like static C functions. Each **asm** macro has one definition and zero or more uses per source file. The definition must appear in the same file with the uses (or be **#included**), and the same **asm** macro may be defined multiply (and differently) in several files.

The **asm** macro definition declares a return type for the macro code, specifies patterns for the formal parameters, and provides bodies of code to expand when the patterns match. When it encounters an **asm** macro call, the compiler replaces uses of the formal parameters by its idea of the assembly language locations of the actual arguments as it expands the code body. This constitutes an important difference between C functions and **asm** macros. An **asm** macro can therefore have the effect of changing the value of its arguments, whereas a C function can only change a copy of its argument values.

The use of an **asm** macro look exactly like normal C function calls. They may be used in expressions and they may return values. The arguments to an **asm** macro may be arbitrary expressions, except that they may not contain uses of the same or other **asm** macros.

When the argument to an asm macro is a function name or a structure contained in memory, the compiler generates code to compute a pointer to the structure or function, and then resulting pointer is used as the actual argument of the macro. Structures contained in registers are passed directly to the function. Addresses

are loaded into a temporary variable before being passed along; these will usually be allocated to a register. The following example shows how passing addresses work:

Example

```
asm void make10(addr)
{
%reg addr
   li r3, 10
   stw r3, 0(addr)
%nearmem addr
   lwz r4, addr
   li r3, 10
   stw r3, 0(r4)
%error
}

void func()
{
   int i, array[10];
   make10(&i);
   make10(array);
   make10(&array[3]);
}
```

Definition

The syntactic descriptions that follow are presented in the style used in “C Language Compilers.” The syntactic classes *type-specifier*, *identifier*, and *parameter-list* have the same form as in that chapter. A syntactic description enclosed in square brackets ([ ]) is optional, unless the right bracket is followed by +. A + means “one or more repetitions” of a description. Similarly, * means “zero or more repetitions.”

asm macro:
    asm [*type-specifier*]*identifier* ([*parameter-list*])
    {
    [*storage-mode-specification-line*
            asm-body] *
    }

An **asm** macro consists of the keyword **asm**, followed by what looks like a C function declaration. Inside the macro body there are one for more pairs of *storage-mode-specification-line(s)* (patterns) and corresponding *asm-body(ies)*. If the *type-specifier* is other than void, the **asm** macro should return a value of the declared type.

storage-mode-specification-line:
    % [*storage-mode* [*identifier* [, *identifier*]*];]+

A *storage-mode-specification-line* consists of a single line (no continuation with \ is permitted) that begins with % and contains the names (*identifier(s)*) and *storage mode(s)* of the formal parameters. Modes for all formal parameters must be given in each *storage-mode-specification-line* (except for error). The % must be the first character on a line. If an **asm** macro has no *parameter-list*, the *storage-mode-specification-line* may be omitted.

Storage Modes

These are the storage modes that the compiler recognizes in **asm** macros.

treg A compiler-selected temporary register.

ureg A C register variable that the compiler has allocated in a machine register.

farmem A location in memory that cannot be accessed with a single load instruction. These may need to be broken up into multiple load instructions, and the steps for doing this may depend on the compilation mode.

nearmem A location in memory that can be accessed with a single load instruction (such as nearby stack variables or variables in a small data area).

reg A **treg** or **ureg**.

con A compile time constant.

mem A **mem** operand matches any allowed machine addressing mode, with the exception of **reg** and **con**.

lab A compiler-generated unique label. The *identifier(s)* that are specified as being of mode **lab** do not appear as formal parameters in the **asm** macro definition, unlike the preceding modes. Such identifiers must be unique. Example:

```
asm void check_for_bit_set(r)
{
%reg r %lab endlab
    b_if_not_bit_set r, endlab
    software_trap
endlab:
%error
}

void check_status(int i, int j)
{
   /* Using the macro twice would cause the "endlab" */
   /* label to be defined twice if it was not specified
```

```
    /* as a unique label */
    check_for_bit_set(i);
    check_for_bit_set(j);
}
```

error Generate a compiler **error**. This mode exists to allow you to flag errors at compile time if no appropriate pattern exists for a set of actual arguments.

farsprel A location on the stack that is too far away to be accessed in a single instruction.

Note: For an asm macro that does not take any arguments, use a blank storage mode (%).

asm Body

The **asm** body represents (presumed) assembly code that the compiler will generate when the modes of all of the formal parameters match the associated pattern. Syntactically, the **asm** body consists of the text between two pattern lines (that begin with %) or between the last pattern line and the } that ends the **asm** macro. C language comment lines are not recognized as such in the **asm** body. Instead they are simply considered part of the text to be expanded.

Formal parameter names may appear in any context in the **asm** body, delimited by non-alphanumeric characters. For each instance of a formal parameter in the asm body the compiler substitutes the appropriate assembly language operand syntax that will access the actual argument at run time. As an example, if one of the actual arguments to an **asm** macro is **x**, an automatic variable, a string like **4(%fp)** would be substituted for occurrences of the corresponding formal parameter. An important consequence of this macro substitution behavior is that **asm** macros can change the value of their arguments. This differs from standard C semantics.

For **lab** parameters a unique label is chosen for each new expansion.

If an **asm** macro is declared to return a value, it must be coded to return a value of the proper type in the machine register that is appropriate for the implementation.

An implementation restriction requires that no line in the **asm** body may start with **%**.

The MCore compiler also supports the following primitives in the body of an **asm** statement.

%SPOFF(*m*)

Given that *m* is of the farsprel storage class, this expands to an integer containing its offset from the stack pointer.

Example:

```
%farsprel m
lrw  r3, %SPOFF(m)
add  r3,r0
ld.w r3,(r3,0)
```

MCore asm procedures

```
#include <stdio.h>

asm int mulword(a,b)
{
%con a %con b
   lrw        r2,a
   lrw        r3,b
   mult       r2,r3
%con a %reg b
   lrw        r2,a
   mult       r2,b
%reg a %con b
   lrw        r2,b
   mult       r2,a
%reg a %reg b
   mov        r2,a
   mult       r2,b
%con a %nearmem b
   ld.w       r2,b
   lrw        r3,a
   mult       r2,r3
%nearmem a %con b
   ld.w       r2,a
   lrw        r3,b
   mult       r2,r3
%nearmem a %nearmem b
   ld.w       r2,a
   ld.w       r3,b
   mult       r2,r3
%con a %farsprel b
   lrw        r2,%SPOFF(b)
   addu       r2,r0
   ld.w       r2,(r2,0)
   lrw        r3,a
   mult       r2,r3
%farsprel a %con b
   lrw        r2,%SPOFF(a)
   addu       r2,r0
   ld.w       r2,(r2,0)
```

```
    lrw         r3,b
    mult        r2,r3
%farsprel a %farsprel b
    lrw         r2,%SPOFF(a)
    addu        r2,r0
    ld.w        r2,(r2,0)
    lrw         r3,%SPOFF(b)
    addu        r3,r0
    ld.w        r3,(r3,0)
    mult        r2,r3
%farmem a %farmem b
    lrw         r2,a
    ld.w        r2,(r2,0)
    lrw         r3,b
    ld.w        r3,(r3,0)
    mult        r2,r3
%error
}

short x = 30;
short y = 2;
int z = 10;

func()
{
    int tmp1 = x, tmp2 = y;
    int ztmp = z;
    return mulword(10,20) + mulword(tmp1,tmp2) + mulword(ztmp,30) +
            mulword(ztmp,ztmp) + mulword(10,ztmp) + mulword(x,y);
}

func()
{
    int tmp1 = x, tmp2 = y;
    int ztmp = z;
    return mulword(10,20) + mulword(tmp1,tmp2) + mulword(ztmp,30) +
            mulword(ztmp,ztmp) + mulword(10,ztmp) + mulword(x,y);
}

func2()
{
    int tmp1 = x, tmp2 = y;
    int ztmp = z;
    return (10*20) + (tmp1*tmp2) + (ztmp*30) +
            (ztmp*ztmp) + (10*ztmp) + (x*y);
}

main()
{
    if (func() != func2())
        printf("ASMPROC FAIL: %d (wrong) != %d (correct)\n",func(),func2());
    return 0;
```

```
}
```

Writing asm Macros

Here are some guidelines for writing **asm** macros.

1. Know the implementation. You must be familiar with the C compiler and assembly language with which you are working. You can consult the Application Binary Interface for your machine for the details of function calling and register usage conventions.
2. Observe register conventions. You should be aware of which registers the C compiler normally uses for scratch registers or register variables. An **asm** macro may alter scratch registers at will, but the values in register variables must be preserved. You must know in which register(s) the compiler returns function results.
3. Handle return values. **asm** macros may "return" values. That means they behave as if they were actually functions that had been called via the usual function call mechanism. **asm** macros must therefore mimic C's behavior in that respect, passing return values in the same place as normal C functions. Float and double results sometimes get returned in different registers from integer-type results. On some machine architectures, C functions return pointers in different registers from those used for scalars. Finally, structs may be returned in a variety of implementation-dependent ways.
4. Cover all cases. The **asm** macro patterns should cover all combinations of storage modes of the parameters. The compiler attempts to match patterns in the order of their appearance in the **asm** macro definition.

If the compiler encounters a storage mode of error while attempting to find a matching pattern, it generates a compile time error for that particular **asm** macro call.

5. Beware of argument handling. **asm** macro arguments are used for macro substitution. Thus, unlike normal C functions, **asm** macros can alter the underlying values that their arguments refer to. Altering argument values is discouraged, however, because doing so would make it impossible to substitute an equivalent C function call for the **asm** macro call.
6. **asm** macros are inherently non-portable and implementation-dependent. Although they make it easier to introduce assembly code reliably into C code, the process cannot be made foolproof. You will always need to verify correct behavior by inspection and testing.

7. Debuggers will generally have difficulty with **asm** macros. It may be impossible to set breakpoints within the inline code that the compiler generates.

8. Because the optimizers are highly tuned to the normal code generation sequences of the compiler, using **asm** macros may cause optimizers to produce incorrect code. Generally speaking, any **asm** macro that can be directly replaced by a comparable C function may be optimized safely. However, the sensitivity of an optimizer to **asm** macros varies among implementations and may change with new software releases.

B

Viewpathing

This appendix contains:

- Theory of Operation
- Limitations
- Environment Variables

Viewpathing is a simple, lightweight implementation of workspaces. This feature gives our tools a hierarchical method for searching multiple directories for requested input files. For example, with this feature the compiler could first look in the current developer's local source directory for a source file, then, in the development group's group source directory, and finally, in a multi-group global source directory.

Theory of Operation

When a viewpath is specified, by means of the environment variable NVPATH, the tools treat all input filenames as viewpath-relative. To create a viewpath-relative filename, the tools first determine the difference between the first element of the NVPATH variable and the current working directory. For example, if `NVPATH=/1:/2:/3` and your current working directory is `/1/subdir/`, then the difference is `subdir/`.

This difference is appended to each node in the viewpath, in order to generate the path prefix to append to the relative filename provided to the tool. In the previous example, if the tool is given `src/file.c,` then the searched locations will be, in order:

/1/subdir/src/file.c

/2/subdir/src/file.c

/3/subdir/src/file.c

For effective viewpathing, the developer must run all tools from a current working directory which is a subdirectory of the first element in the NVPATH, referred to as the **root node**.

Undesirable behavior may result if "." is specified as the **root node**.

Viewpathing does not affect the locations of output files. All temporary files will be unaffected and all other output files (such as executables, object files, etc.) will be created relative to the **root node**.

If a file located in a directory down the viewpath, for example, in the 2nd or 3rd node, is opened for modification, such as adding files to an archive or editing a text file, then the original file will first be copied into a location relative to the **root node**. Then, it will be opened for modification. In this way, a developer's modification will not affect the development group's files.

Limitations

When creating a file for output in the **root node**, the required intermediate directories will not be created in the **root node**, even if the corresponding directory path exists in a node down the viewpath.

Environment Variables

The following environment variables control viewpathing behavior of the tools. These environment variables must be set to a non-null value to take effect.

NVPATHEnable viewpathing and should be a colon separated list of pathnames.

GHS_VP_DEBUG

Enable the output of diagnostic information from programs that use viewpathing.

GHS_VP_NONE

Disable viewpathing, even if NVPATH is set.

GHS_VP_SLOW

Disable a performance optimization where files which are opened for creation and not updating are not copied down the viewpath if they already exist.

Example

This example requires three existing empty directories:

```
        /test/local
/test/group
/test/global
</test>: pwd
   /test
</test>: ls -Ag *
   global:
   total 2
   -rw-rw-r--   1 green         7 Feb 11 15:56 file1.c
   group:
   total 2
   -rw-rw-r--   1 green        15 Feb 11 15:58 file2.c
   local:
   total 2
   -rw-rw-r--   1 green        34 Feb 11 15:55 file3.c
</test>: cd local
</test/local>: setenv NVPATH "/test/local:/test/group:/test/global"
```

```
</test/local>: setenv GHS_VP_DEBUG 1
</test/local>: ax crv archive.a file1.c file2.c file3.c
   ax: info: Viewpathing support is ON (FAST).
   ax: info: Adding viewpath node 1: /test/local
   ax: info: Adding viewpath node 2: /test/group
   ax: info: Adding viewpath node 3: /test/global
   a - file1.c
   ax: info: Located file: /test/global/file1.c
   a - file2.c
   ax: info: Located file: /test/group/file2.c
   a - file3.c
</test>: cd ..
</test>: ls -Ag *
   global/:
   total 2
   -rw-rw-r--   1 green         7 Feb 11 15:56 file1.c
   group/:
   total 2
   -rw-rw-r--   1 green        15 Feb 11 15:58 file2.c
   local/:
   total 4
   -rw-rw-r--   1 green       246 Feb 11 16:05 archive.a
   -rw-rw-r--   1 green        34 Feb 11 15:55 file3.c
</test>: cd local
</test/local>: ax tv archive.a
   ax: info: Viewpathing support is ON (FAST).
   ax: info: Adding viewpath node 1: /test/local
   ax: info: Adding viewpath node 2: /test/group
   ax: info: Adding viewpath node 3: /test/global
   rw-rw-r-- 4025/28        7 Feb 11 15:56 1998 file1.c
   rw-rw-r-- 4025/28       15 Feb 11 15:58 1998 file2.c
   rw-rw-r-- 4025/28       34 Feb 11 15:55 1998 file3.c
</test/local>: echo "Completely new and larger file 1" > file1.c
</test/local>: ls -Ag
   total 6
   -rw-rw-r--   1 green       246 Feb 11 16:05 archive.a
   -rw-rw-r--   1 green        32 Feb 11 16:09 file1.c
   -rw-rw-r--   1 green        34 Feb 11 15:55 file3.c
</test/local>: ax r archive.a file1.c
   ax: info: Viewpathing support is ON (FAST).
   ax: info: Adding viewpath node 1: /test/local
   ax: info: Adding viewpath node 2: /test/group
   ax: info: Adding viewpath node 3: /test/global
</test/local>: ax tv archive.a
   ax: info: Viewpathing support is ON (FAST).
   ax: info: Adding viewpath node 1: /test/local
   ax: info: Adding viewpath node 2: /test/group
   ax: info: Adding viewpath node 3: /test/global
   rw-rw-r-- 4025/28       32 Feb 11 16:09 1998 file1.c
   rw-rw-r-- 4025/28       15 Feb 11 15:58 1998 file2.c
   rw-rw-r-- 4025/28       34 Feb 11 15:55 1998 file3.c
```

C

C Runtime Libraries

This appendix contains:

- Built-in Functions
- Reentrancy
- libansi.a data structures and functions
- libind.a functions
- Less Buffered I/O

To use the Green Hills C Library, the user needs a standard Green Hills compiler license. Under this license, unlimited distribution of programs linked with the Green Hills C Library object code is permitted without charge. However, distribution of the Green Hills C Library source code or object code is not permitted.

Built-in Functions

The Green Hills C and C++ compilers implement certain built-in functions. A C or C++ built-in function name begins with two underscores: _ _. A built-in function is recognized by the compiler as a special function. It usually generates optimized inline code, often using special instructions which do not correspond to standard C and C++ operations.

_ _MULUH, _ _MULSH

C and C++ function prototype:

extern unsigned int __MULUH(unsigned int a, unsigned int b);

extern signed int __MULSH(signed int a, signed int b);

The **__MULUH(a,b)** built-in function takes as arguments two 32-bit unsigned integers and returns the high 32-bit half of their 64-bit unsigned product. This built-in function generates inline code for most targets.

The **__MULSH(a,b)** built-in function takes as arguments two 32-bit signed integers and returns the high 32-bit half of their 64-bit signed product.

The **__MULUH** and **__MULSH** built-in functions generate inline code for the following targets:

MIPS	PowerPC	960	TriCore
V800	486	Sparc	

_ _CLZ32

C and C++ function prototype:

externunsigned int __CLZ32(unsigned int a);

The **__CLZ32(a)** built-in function takes a 32-bit integer argument and returns the count of leading zeros, which is a number from 0 to 32.

The **__CLZ32** built-in function generates inline code for the following targets:

PowerPC	960	TriCore	MCore

_ _DI, _ _EI

C and C++ function prototype:

externvoid __DI(void);/* disable interrupts */

externvoid __EI(void);/* enable interrupts */

The **__DI()** built-in function disables interrupts.

The **__EI()** built-in function enables interrupts.

These two built-in functions can be used together as a pair to bracket a critical section of code which must not be interrupted.

The **__DI** and **__EI** built-in functions generate inline code for the following targets.

MIPS	PowerPC
SH	V800
TriCore	FR
MCore	

For targets other than those listed, the **__DI()** and **__EI()** are undefined in the library.

Reentrancy

A reentrant function can be interrupted, suspended, and called again, then resumed from its suspended state. Generally, reentrant functions do not write to global variables or local static data structures, and call only other reentrant functions.

In the tables on the following pages, the code letters listed below are used to identify the reentrancy of the library functions:

Y Function is reentrant

N Function is not reentrant

I Function does I/O; for most purposes it is not reentrant.

E Function writes to the global variable **errno,** otherwise is reentrant. It may be possible to modify the library source so writing to **errno** is a reentrant operation. See the functions

_ _gh_set_errno() and **_ _gh_get_errno()**. The complete source code is **libsrc/ind_errn.c**.

libansi.a data structures and functions

Variable	Source Module	Declaration
_CTYPE	ccctype.c	unsigned char _CTYPE[]
sys_errlist	ccsyserr.c	char *sys_errlist[]
tolower	ccctype1.c	short _tolower_[]
toupper	ccctype1.c	short _toupper_[]

Function	Source Module	Reentrant?	Arguments/Return Value
abort	ccabort.c	Y	void abort(void)
abs	ccabs.c	Y	int abs(int x)
asctime	ccstrftm.c	N	char *asctime(const struct tm *t)
_assert	ccassert.c	I	void _assert(const char *problem, const char *filename, int line)
assert	ccassert.c	I	void assert(int value)
atexit	ccatexit.c	N	int atexit(void (*func)(void))
atof	ccatof.c	Y	double atof(const char *str)
atoi	ccatoi.c	Y	int atoi(const char *str)
atol	ccatol.c	Y	long atol(const char *str)
bcmp	ccbcmp.c	Y	bcmp(char *b1, char *b2, int length)
bcopy	ccbcopy.c	Y	bcopy(char *from, char *to, int n)
bsearch	ccbsrch.c	Y	void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))
bufcpy	ccbufcpy.c	Y	bufcpy(char *to, char *from, int n)
bzero	ccbzero.c	Y	bzero(char *pt, int n)
calloc	cccalloc.c	N	void *calloc(size_t num, size_t size)
cfree	cccfree.c	N	void cfree(char *item)
clearerr	ccclrerr.c	I	void clearerr(FILE *file)
clearn	ccclearn.c	Y	void clearn(int n, char *pt)
clock	ccclock.c	Y	clock_t clock(void)

Function	Source Module	Reentrant?	Arguments/Return Value
ctime	ccctime.c	N	char *ctime(const time_t *timer)
difftime	ccdifftm.c	Y	double difftime(time_t time1, time_t time0)
div	ccdiv.c	Y	div_t div(int number, int denom)
__docvt	ccdocvt.c	Y	internal use only
_doprnt	ccvprintf.c	I	int _doprnt(const char *format, va_list args, FILE *stream)
_doscan	ccscanf.c	I	_doscan(const char *format, va_list args, FILE *stream)
ecvt	ccecvt.c	N	char *ecvt(double value, int ndig, int *decpt, int *sign)
eprintf	cceprntf.c	I	int eprintf(const char *format, ...)
execl	ccexecl.c	Y	int execl(const char *name, const char *args, ...)
execle	ccexecle.c	Y	int execle(const char *name, const char *args, ...)
execv	ccexecv.c	Y	int execv(const char *name, char *const *argv)
exit	ccexit.c	N	void exit(int val)
fabs	ccfabs.c	Y	double fabs(double x)
fclose	ccfclose.c	I	int fclose(FILE *file)
fcvt	ccecvt.c	N	char *fcvt(double value, int ndig, int *decpt, int *sign)
feof	ccfeof.c	Y	int feof(FILE *stream)
ferror	ccferror.c	Y	int ferror(FILE *stream)
fflush	ccfflush.c	I	int fflush(FILE *file)
ffs	ccffs.c	Y	int ffs(int i)
fgetc	ccfgetc.c	I	int fgetc(FILE *file)
fgetpos	ccfgetps.c	I	int fgetpos(FILE *file,fpos_t *pos)
fgets	ccfgets.c	I	char *fgets(char *str, int n, register FILE *file)
_filbuf	ccfilbuf.c	I	_filbuf(register FILE *file)
filln	ccfilln.c	Y	void filln(int n, char *pt, int fill)
_flsbuf	ccflsbuf.c	I	_flsbuf(int ch, FILE *file)
fdopen	ccfopen.c	I	FILE *fdopen(int fno,const char *mode)
fopen	ccfopen.c	I	FILE *fopen(const char *name, const char *mode)
fprintf	ccprntf.c	I	int fprintf(FILE *stream, const char *format, ...)
fputc	ccfputc.c	I	int fputc(int ch,FILE *file)
fputs	ccfputs.c	I	int fputs(const char *str,FILE *file)
fread	ccfread.c	I	size_t fread(void *ptr,size_t size,size_t nitems,FILE *file)
free	ccmalloc.c	N	void free(void *ptr)
freopen	ccfopen.c	I	FILE *freopen(const char *name, const char *mode, FILE *file)
frexp	ccfrexp.c	Y	double frexp(double value, int *eptr)
frexpf	ccfrexpf.c	Y	float frexpf(float value, int *eptr)
fscanf	ccscanf.c	I	int fscanf(FILE *stream, const char *format, ...)

Function	Source Module	Reentrant?	Arguments/Return Value
fseek	ccfseek.c	I	int fseek(FILE *stream, long int offset, int ptrname)
fsetpos	ccfsetps.c	I	int fsetpos(FILE *file,const fpos_t *pos)
ftell	ccftell.c	I	long ftell(FILE *stream)
fwrite	ccfwrite.c	I	size_t fwrite(const void *ptr,size_t size,size_t nitems, register FILE *file)
gcvt	ccgcvt.c	N	char *gcvt(double value, int ndig, char *buf)
getc	ccgetc.c	I	int getc(FILE *f)
getchar	ccgetchr.c	I	int getchar(void)
getenv	ccgetenv.c	Y	char *getenv(char *np)
getl	ccgetl.c	I	long getl(FILE *file)
gets	ccgets.c	I	char *gets(char *str)
getw	ccgetw.c	I	int getw(FILE *file)
index	ccindex.c	Y	char *index(const char *str, const char ch)
isalnum	ccfuncs.c	Y	int isalnum(int c)
isalpha	ccfuncs.c	Y	int isalpha(int c)
iscntrl	ccfuncs.c	Y	int iscntrl(int c)
isdigit	ccfuncs.c	Y	int isdigit(int c)
isgraph	ccfuncs.c	Y	int isgraph(int c)
islower	ccfuncs.c	Y	int islower(int c)
isprint	ccfuncs.c	Y	int isprint(int c)
ispunct	ccfuncs.c	Y	int ispunct(int c)
isspace	ccfuncs.c	Y	int isspace(int c)
isupper	ccfuncs.c	Y	int isupper(int c)
isxdigit	ccfuncs.c	Y	int isxdigit(int c)
labs	cclabs.c	Y	long labs(long x)
ldexp	ccldexp.c	E	double ldexp(double value, int exp)
ldexpf	ccldexpf.c	E	float ldexpf(float value, int exp)
ldiv	ccldiv.c	Y	ldiv_t ldiv(long int number, long int denom)
localeconv	cclocale.c	Y	struct lconv *localeconv(void)
longjmp	ccsetjmp.xxx (xxx is the target)	Y	void longjmp (jmp_buf env, int val)
malloc	ccmalloc.c	N	void *malloc(size_t size)
mblen	ccmblen.c	Y	int mblen(const char *s, size_t n)
mbstowcs	ccmbswcs.c	Y	size_t mbstowcs(wchar_t *pwcs, const char *mbs, size_t n)
mbtowc	ccmbtowc.c	Y	int mbtowc(wchar_t *pwc, const char *s, size_t n)
memchr	ccmemchr.c	Y	void *memchr(const void *s, int c, size_t n)
memcmp	ccmemcmp.c	Y	int memcmp(const void *s1, const void *s2, size_t length)

Function	Source Module	Reentrant?	Arguments/Return Value
memmove	ccmemmov.c	Y	void *memmove(void *s1, const void *s2, size_t n)
mktemp	ccmktemp.c	N	char *mktemp(char *str)
mktime	ccmktime.c	Y	time_t mktime(struct tm *timeptr)
modf	ccmodf.c	Y	double modf(double value, double *iptr)
on_exit	ccatexit.c	N	int on_exit(void (*func)(void), char * arg)
perror	ccperror.c	I	void perror(const char *str)
printf	ccprintf.c	I	int printf(const char *format, ...)
putc	ccputc.c	I	int putc(int ch, FILE *f)
putchar	ccputchr.c	I	int putchar(int ch)
putl	ccputl.c	I	long putl(long l, FILE *file)
puts	ccputs.c	I	int puts(const char *str)
putw	ccputw.c	I	putw(int w, FILE *file)
qsort	ccqsort.c	Y	void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *))
rand	ccrand.c	N	int rand()
realloc	ccmalloc.c	N	void *realloc(void *old, size_t new_size)
remove	ccremove.c	I	int remove(const char *filename)
rewind	ccrewind.c	I	void rewind(FILE *stream)
rindex	ccrindex.c	Y	char *rindex(const char *str, const char ch)
scanf	ccscanf.c	I	scanf(const char *format, ...)
setlocale	cclocale.c	Y	char *setlocale(int category, const char *locale)
setbuf	ccsetbuf.c	N	void setbuf(FILE *stream, char *buf)
setjmp	ccsetjmp.xxx (xxx is the target)	Y	int setjmp (jmp_buf env)
setlinebuf	ccsetlbf.c	N	int setlinebuf(FILE *stream)
setvbuf	ccsetvbf.c	N	int setvbuf(FILE *stream, char *buf, int mode, size_t size)
sprintf	ccsprntf.c	E	int sprintf(char *s, const char *format, ...)
srand	ccrand.c	N	void srand(int val)
sscanf	ccscanf.c	N	int sscanf(const char *str, const char *format, ...)
strcat	ccstrcat.c	Y	char *strcat(char *s2, const char *str1)
strchr	ccstrchr.c	Y	char *strchr(const char *str, int ch)
strcmp	ccstrcmp.c	Y	int strcmp(const char *str1, const char *str2)
strcoll	ccstrcol.c	Y	int strcoll(const char *s1, const char *s2)
strcpy	ccstrcpy.c	Y	char *strcpy(char *s2, const char *str1)
strcspn	ccstrcsp.c	Y	size_t strcspn(const char *s1, const char *s2)
strerror	ccstrerr.c	Y	char *strerror(int errnum)

Function	Source Module	Reentrant?	Arguments/Return Value
strftime	ccstrftm.c	Y	size_t strftime(char *start, size_t maxsize, const char *format, const struct tm *timeptr)
strindex	ccstridx.c	Y	int strindex(char *str, char *sub)
strlen	ccstrlen.c	Y	size_t strlen(const char *str)
strncmp	ccstrncm.c	Y	int strncmp(const char *str1, const char *str2, register int n)
strncpy	ccstrncp.c	Y	char *strncpy(char *s2, char *str1, register int n)
strpbrk	ccstrpbr.c	Y	char *strpbrk(const char *s1, const char *s2)
strrchr	ccstrrch.c	Y	char *strrchr(const char *str, int ch)
strrindex	ccstrrdx.c	Y	int strrindex(char *str, char *sub)
strsave	ccstrsav.c	N	char *strsave(char *str)
strspn	ccstrspn.c	Y	size_t strspn(const char *s1, const char *s2)
strstr	ccstrstr.c	Y	char *strstr(const char *str, const char *sub)
strtod	ccstrtod.c	E	double strtod(const char *str, char **endptr)
strtok	ccstrtok.c	N	char *strtok(char *s1, const char *s2)
strtol	ccstrtol.c	E	long strtol(const char *str, char **ptr, register int base)
strtoul	ccstrtul.c	E	unsigned long strtoul(const char *str, char **ptr, register int base)
strxfrm	ccstrxfm.c	Y	size_t strxfrm(char *s1, const char *s2, size_t n)
swab	ccswab.c	Y	swab(char *from, char *to, int nbytes)
tmpfile	cctmpfil.c	I	FILE *tmpfile(void)
tmpnam	cctmpnam.c	N	char *tmpnam(char *s)
tolower	ccfuncs.c	Y	int tolower(int c)
toupper	ccfuncs.c	Y	int toupper(int c)
ungetc	ccungetc.c	I	int ungetc(int ch,FILE *file)
vfprintf	ccvprntf.c	I	int vfprintf(FILE *stream, const char *format, va_list args)
vfscanf	ccscanf.c	I	vfscanf(FILE *stream, const char *format, va_list args)
vprintf	ccfprntf.c	I	int vprintf(const char *format, va_list args)
vscanf	ccscanf.c	I	vscanf(const char *format, va_list ap)
vsprintf	ccsprntf.c	I	int vsprintf(char *s, const char *format, va_list ap)
vsscanf	ccscanf.c	I	vsscanf(const char *str, const char *format, va_list ap)
wcstombs	ccwcsmbs.c	Y	size_t wcstombs(char *s, const wchar_t *pwcs, size_t n)
wctomb	ccwctomb.c	Y	int wctomb(char *s, wchar_t wchar)

libind.a functions

Function	Source Module	Reentrant?	Arguments/Return Value
acos	indacos.c	E	double acos(double x)
acosf	indacosf.c	E	float acosf(float arg)
acosh	indacosh.c	E	double acosh(double x)
asin	indasin.c	E	double asin(double x)
asinf	indasinf.c	E	float asinf(float arg)
asinh	indasinh.c	E	double asinh(double x)
atan	indatan.c	E	double atan(double x)
atan2	indatan2.c	E	double atan2(double y, double x)
atan2f	indatn2f.c	E	float atan2f(float y, float x)
atanf	indatanf.c	E	float atanf(float x)
atanh	indatanh.c	E	double atanh(double x)
cabs	indcabs.c	E	double cabs(struct complex z)
ceil	indceil.c	Y	double ceil(double x)
cos	indcos.c	Y	double cos(double f0)
cosf	indcosf.c	Y	float cosf(float f0)
cosh	indcosh.c	E	double cosh(double x)
coshf	indcoshf.c	E	float coshf(float x)
erf	inderf.c	Y	double erf(double x)
erfc	inderf.c	Y	double erfc(double x)
exp	indexp.c	E	double exp(double x)
expf	indexpf.c	E	float expf(float x)
fabs	indfabs.c	Y	double fabs(double x)
floor	indfloor.c	Y	double floor(double x)
fmod	indfmod.c	Y	double fmod(double x,double y)
gamma	indgamma.c	E	double gamma(double x)
_gh_va_arg	indvaarg.c	Y	char *_gh_va_arg(p, align, regtyp, size)
hypot	indhypot.c	E	double hypot(double x,double y)
isinf	indisinf.c	Y	int isinf(double x)
isnan	indisnan.c	Y	int isnan(double x)
j0	indbessl.c	E	double j0(double x)
j1	indbessl.c	E	double j1(double x)
jn	indbessl.c	E	double jn(int n,double x)
log	indlog.c	E	double log(double x)
log10	indlog.c	E	double log10(double x)
log10f	indlogf.c	E	float log10f(float x)

Function	Source Module	Reentrant?	Arguments/Return Value
logf	indlogf.c	E	float logf(float x)
matherr	inderr.c	Y	int matherr(struct exception *ex)
memcpy	ccmemcpy.c	Y	void *memcpy(void *s1, const void *s2, size_t n)
memset	ccmemset.c	Y	void *memset(void *s, int c, size_t n)
pow	indpow.c	E	double pow(double x, double y)
powf	indpowf.c	E	float powf(float x, float y)
rmatherr	inderr.c	Y	int rmatherr(struct rexception *ex)
_rnerr	indrnerr.c	N	int _rnerr(int num, int linenum, char *str, void *ptr, void *al, void *as, void *a3, void *a4, void *a5, int len);
sin	indsin.c	Y	double sin(double f0)
sinf	indsinf.c	Y	float sinf(float f0)
sinh	indsinh.c	E	double sinh(double x)
sinhf	indsinhf.c	E	float sinhf(float arg)
sqrt	indsqrt.c	E	double sqrt(double f0)
sqrtf	indsqrtf.c	E	float sqrtf(float f0)
tan	indtan.c	E	double tan(double x)
tanf	indtanf.c	E	float tanf(float arg)
tanh	indtanh.c	E	double tanh(double x)
tanhf	indtanhf.c	E	float tanhf(float x)
y0	indbessl.c	E	double y0(double x)
y1	indbessl.c	E	double y1(double x)
yn	indbessl.c	E	double yn(int n, double x)

Less Buffered I/O

The Green Hills ANSI C library includes a *Standard I/O Package*, abbreviated **stdio**.

stdio includes formatted I/O using **printf** and **scanf**, character I/O using **getc** and **putc**, plus other features.

In traditional implementations, all I/O performed in **stdio** is buffered automatically.

In embedded programming there is a constant trade-off between space and performance. Buffering improves performance by increasing the average number of characters written per system call. However, buffering occupies space for the code to manage the buffers, as well as for buffers themselves.

If I/O is totally unbuffered, every character read or written requires a system call. A benefit of this mode is that I/O is never delayed until the buffer is full or lost in a buffer if the application exits abnormally.

If I/O is buffered in the traditional manner, several kilobytes of code are added to the application, because the **stdio** package invokes **malloc** and various other routines to manage the buffering.

The *Less Buffered I/O* mode in which the Green Hills ANSI C library is built provides a reasonable compromise between full buffering and unbuffered I/O.

Rather than allocate a permanent buffer for each file, which is used as long as the file is open, *Less Buffered I/O* performs buffering within each of the following routines:

fwrite

fputs

puts

printf

fprintf

sprintf

vprints

vfprintf

vsprintf

Thus, during a single call, any characters, which are written to one of these functions, will be buffered. In this way, one function call will often require only one output system call. Once the function completes, all characters are written to the file. No characters are ever left in a buffer after the function call, eliminating the risk that output will be lost, and eliminating the need to flush buffers when a file is closed or the program exits.

No input routines are buffered in any way by the *Less Buffered I/O* method. Files are not closed upon program termination, except as noted below.

The program may enable full buffering of either input or output by calling either **setbuf()** or **setvbuf()**. In this case, characters are only written to a buffered file when the buffer is full, or **fflush()** or **fclose()** is called. Upon normal termination of the program, if **setbuf()** or **setvbuf()** has been called at any time, then all open files will be flushed and then closed.

All files perform in exactly the same manner, whether they are opened by default (**stdin**, **stdout**, and **stderr**) or opened by using **fopen()**.

Index

Symbols

Index

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z