Green Hills C

User's Guide

Version 1.8.9

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.

PubID: L02B-I1099-89NG

CONTENTS

	PREFACE	P-1
	About this Manual	P-2
	Supporting Documentation	P-2
	Typographical Conventions	P-3
1	C LANGUAGE FEATURES	1
	C Language Modes	2
	Pragma Directives	5
	The C Preprocessor	15
	Type Qualifiers	24
	Structure and Union Assignment and Comparisons	25
	Bit Fields	25
	Old-Fashioned Syntax	28
	Functions Returning Structures or Unions	29
	Enumerated Types	30
	Functions with Variable Arguments	31
	asm Statement	33
	Japanese Automotive C	34
	C Run-time Library	36
	Compiler Limitations	36
2	MIXING LANGUAGES	39
	How the Driver Builds a Mixed Language Executable	40
	Initialization of Libraries	42
	Main() Program Examples	42
	Performing I/O on a Single File in Multiple Languages	44
	Native UNIX Libraries versus Green Hills Libraries	45
	Calling a C Routine from FORTRAN	45
	Calling a FORTRAN Routine from C	53
	Calling a C Routine from Ada	60

CONTENTS

	Interfacing Pascal and C	62
	C Routines and Header Files In C++	63
	Using C++ in C Programs	64
	Function Prototyping in C versus C++	65
3	WRITING PORTABLE CODE	67
	Compatibility Between Green Hills Compilers	68
	Word Size Differences	68
	Byte Order Problems	69
	Alignment Requirements	70
	Structures, Unions, and Bit Fields	71
	Assumptions about Function Calling Conventions	72
	Pointer Issues	73
	NULL Pointer	73
	Character Set Dependencies	74
	Floating Point Range and Accuracy	74
	Operating System Dependencies	75
	Assembly Language Interfaces	75
	Evaluation Order	75
	Machine Specific Arithmetic	76
	Illegal Assumptions about Compiler Optimizations	77
	Memory Optimization Restrictions	78
	Problems with Source Level Debuggers	79
	Problems with Compiler Memory Size	80
	PCC Mode Incompatibilities	81
	Detection of Portability Problems	82
4	OPTIMIZATION	83
	Default Optimizations	84
	General Optimizations Enabled with the -O Option	87

CONTENTS

Specialized Optimizations Set with the Suboptions -OLAMISD 93
Selecting Optimizations 104

A KANJI CHARACTER SUPPORT A-1
About Kanji A-2
Green Hills Support for Kanji A-2
Wide-Character vs. Multi-Byte Representation A-3

INDEX I-1

.

PREFACE

ABOUT THIS MANUAL

This manual describes the features of the C language supported by the Green Hills C compilers. Only those features of the language which are consistent across the entire family of Green Hills C compilers on all hosts and targets are described here.

This manual is not a complete reference on the C programming language, nor does it attempt to teach programming methods.

All of the examples given in this manual were done on a Sun workstation running a UNIX environment. Where differences may exist on other systems, an attempt has been made to mention them.

In all of the explanations and examples in this manual, it is assumed that the Green Hills products have been installed in the directory **/usr/green**. If this is not the case for your system, simply substitute the correct directory wherever you see **/usr/green**.

SUPPORTING DOCUMENTATION

A complete set of documentation for the Green Hills C compiler includes a *Development Guide* which is specific to that processor and operating system. For information on using the compiler and for details which are specific to your distribution, please refer to the *Development Guide* for that distribution.

In addition to the documentation provided with the Green Hills C compiler, access to one or more of the following references will be valuable:

- Kernighan and Ritchie, *The C Programming Language*, First Edition, 1978, Prentice Hall
- Kernighan and Ritchie, *The C Programming Language*, Second Edition, 1988, Prentice Hall
- Harrbison and Steele, *C: A Reference Manual*, Third Edition, 1991, Prentice Hall
- ISO C Standard, ISO/IEC 9899:1990, (previously known as ANSI X3.159-1989)

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

ccppc [*-processor*] *filename*

the command **ccppc** should be entered as given, the word "processor" may optionally be substituted with the appropriate option, and the word "filename" must be substituted with the appropriate file name.

Chapter

1

C LANGUAGE FEATURES

This chapter describes aspects of the C language which are particular to Green Hills C.

C LANGUAGE MODES

Green Hills C supports four dialects of the C programming language:

- K+R
- Transition
- Permissive ANSI
- Strict ANSI

These terms are used throughout this document whenever a particular construct is supported only in the mode for a specific dialect. If no specific mode is mentioned, the construct is supported in all four modes.

The C Language Mode can be selected by the command line options shown in the table below. Redundant options are provided for compatibility with various UNIX C compilers.

Mode	Options
K+R	-Xs, -k+r
Transition	-Xt
Permissive ANSI	-Xa, -ansi
Strict ANSI	-Xc, -ANSI

Table 1 **Language Mode Options**

All files in a single program must be compiled with the same C language mode. Failure to do so may result in obscure failures at link-time or at run-time.

K+R MODE

This is the classic dialect of C in which UNIX was originally written. It is described in the first edition of *The C Programming Language* by Kernighan and Ritchie. The most important implementation of this dialect was the Portable C Compiler (PCC). There are many versions of this compiler, including the C compilers distributed with System V.3, BSD 4.3 and SunOS 4.x. Together these compilers implement hundreds of extensions. Without these extensions it is

impossible to compile the UNIX operating system or many existing C application programs.

Green Hills has chosen to follow the particular version of pcc distributed with the Berkeley 4.3 BSD version of UNIX. To this, many documented and undocumented extensions from other compilers have been added to make porting existing C programs to the Green Hills compiler as straightforward as possible.

The following are a few of the useful features of C supported in this mode:

- the **void** keyword
- enum types
- the predefined symbol _ _LINE_ _, _ _FILE_ _, _ _TIME_ _, and _ _DATE_ _
- **#error**, **#ident**, **#elif** and **#defined()** preprocessor directives
- functions that return structures
- **asm()** statements
- initialized extern variables
- initialized unions
- initialized automatic structs and arrays

TRANSITION MODE

The Transition mode is named after a similar mode of the C compiler provided with UNIX System V Release 4. This mode is intended to allow existing programs written in K & R C to make use of certain features of ANSI C. In particular, function prototypes and the keywords **const**, **signed**, and **volatile** are supported, along with all of the features of K+R mode. The **f**, **u**, and **l** suffixes for numeric constants are recognized, and spaces are allowed before the leading # in preprocessor directives.

In this mode the header **stdarg.h** must be used rather than **varargs.h** because function declarations follow the ANSI C rules rather than the K + R rules.

PERMISSIVE ANSI MODE

In this mode all of the features of the ANSI standard are provided, without some of the restrictions. In particular the ANSI standard prohibits certain

programming practices which may be useful or even necessary in certain cases. Some cases which are prohibited by the ANSI standard generate a warning in Permissive ANSI mode. Others, such as the following, generate no warning at all:

- casts in constant expressions
- bit fields with type char, short, or long, rather than int as required by ANSI
- lone semicolons outside the body of a procedure
- empty structure or union declarations
- empty source files
- **asm()** statements
- floating point values in constant expressions
- uncommented characters after **#endif**
- functions not declared void which do not return a value

Permissive ANSI mode is recommended for all existing C code which comes close to ANSI C and for all new development.

STRICT ANSI MODE

In this mode, all of the features of the ANSI standard are provided and all of the restrictions are enforced. In most cases, the code generated in this mode will be the same as in Permissive ANSI mode if the code compiles and if there are no uses of predefined symbols which differ between the two modes.

PRAGMA DIRECTIVES

#pragma directives control compiler behavior by allowing individual compiler implementations to add special features to C without changing the C language. Programs that use **#pragma** stay relatively portable, although they make use of features unavailable in all ANSI C implementations.

The ANSI standard states in 6.8.6 “Any pragma that is not recognized by the implementation is ignored.”

This requirement may help when porting code between compilers, because one compiler will silently recognize and process a certain pragma, while a different compiler which does not need that pragma or recognize it will silently ignore it. On the other hand, if a pragma is misspelled the compiler will probably not recognize it and therefore will silently ignore it also. For this reason, the option **-unknownpragmawarn** is provided to cause the compiler to give a warning for any pragma which is not recognized. Unrecognized pragma are always ignored.

The majority of Green Hills proprietary #pragma begin with the keyword ghs to differentiate them from other implementations. The compiler considers any pragma beginning with the ghs keyword to be recognized.

If the compiler recognizes a pragma, some amount of syntax checking will be performed. Incorrect pragma may generate warnings but not errors and are always ignored. The option **-nopragmawarn** disables warnings for pragma which the compiler recognizes but which are incorrect.

#pragma asm
#pragma endasm

#pragma asm marks the start of a sequence of lines which are to be copied literally to the assembly language output file. All lines after this pragma are copied until a **#pragma endasm** is encountered. No processing of comments, macros, or preprocessing directives will be performed. **#pragma endasm** may not contain any comments or \ escapes, although it may contain spaces or tabs before and after the # character, as long as the total length of the line does not exceed 255 characters. Both pragma are enabled with the **-pragma_asm_inline** and **-japanese_automotive_c** command line options.

#pragma ghs check=(all,none,assignbound,bounds,nilderef, revert, switch,case,uninitvariable,usevariable,zerodivide)

The following pragma items turn on or off various compile-time or run-time checks. They must be used outside of any function and they control all functions which are defined after the pragma. They all take the form:

```
#pragma ghs check=(pragma items)
```

The parentheses are only needed around *pragma-items* if more than one is used, in which case each item should be separated by a comma. To turn off any of these options, precede the *pragma-items* with the word **no**. This is useful to turn on everything except the indicated flag. For example, use **#pragma ghs check=(all,nozero)** to turn on all checks except **zero**.

All of these pragma items may also be given on the command line with the **-check=***pragma-items* option. For example, to turn on all checking except bounds checking, you could either include the following code in your program:

```
#pragma ghs check=(all,nobounds)
```

or you could use the following command line option:

```
-check=all,nobounds
```

Only the long form of the options are recognized on the command line. Therefore, **-check=assignbounds** is allowed, but **-check=assign** is not.

#pragma ghs check=all
: Turns on all checks.

#pragma ghs check=none
: Turns off all checks.

#pragma ghs check=assign or
#pragma ghs check=assignbound
: Checks that integral assignments to variables smaller than type int are in bounds.

#pragma ghs check=bound or
#pragma ghs check=bounds
: Checks at runtime that array index expressions do not exceed the declared array dimension, if it is known.

#pragma ghs check=nil or
#pragma ghs check=nilderef
Check for NULL pointer dereferences.

#pragma ghs check=revert
Reverts to the settings on the command line.

#pragma ghs check=switch or
#pragma ghs check=case
Generates a runtime warning if the **switch** expression does not match any of the **case** labels. This check is not done when a default case label is used or when the **switch** statement is enclosed in an **if** construct.

#pragma ghs check=uninit or
#pragma ghs check=uninitvariable
Check at runtime for simple variables which are read before they are initialized. Composite variables such as arrays, structs and unions are not checked. This option generates much larger and slower programs. It may be used to detect bugs which are otherwise very hard to find.

#pragma ghs check=use or
#pragma ghs check=usevariable
Generate a compile-time warning for three conditions:

1. Local variables or parameters which are not used.
2. Local variables or parameters which the compiler can detect are read before they are initialized. This checking uses a linear algorithm rather than data flow analysis, therefore it cannot detect all cases of uninitialized variables.
3. Non-void functions which do not return a value.

#pragma ghs check=zero or
#pragma ghs check=zerodivide
Generates an error message indicating that a zero divide occurred and terminates program execution.

If one of the above run-time checking pragma are used in a program, the Green Hills library, **libind**, must be linked in. In many environments, **libind** is linked in by default. In some environments, such as native UNIX, **libind** must be linked in manually.

#pragma ghs ifdebug *pragma*

This pragma and the three which follow it are used as a kind of prefix to any pragma listed in this chapter which begins with the keyword 'ghs'. For example,

```
#pragma ghs check=bounds
```

may be made to take effect only if debugging is enabled like this:

```
#pragma ghs ifdebug check=bounds
```

This invokes pragma *pragma* if the current file is compiled with debugging enabled.

#pragma ghs ifnodebug *pragma*

This invokes the pragma *pragma* if the current file is not compiled with debugging enabled.

#pragma ghs ifoptimize *pragma*

This invokes pragma *pragma* if the current file is being optimized *pragma*.

#pragma ghs ifnooptimize *pragma*

This invokes pragma *pragma* if the current file is not being optimized.

#pragma ghs includeonce

Same as the **-includeonce** option. Header files are only included once. This is illegal in strict ANSI C.

#pragma ghs inline

The next function to be defined becomes an inline function. See also **#pragma inline**.

#pragma ghs interrupt

The next function to be defined becomes an interrupt function.

#pragma ghs nofloat interrupt

The next function to be defined becomes an interrupt function which does not save and restore floating point registers.

#pragma ghs revertoptions

Undoes the effect of all options set via pragma and returns to the default specified on the command line.

#pragma ghs sda=*all*
#pragma ghs sda=*size*

Specifies the threshold size for variables placed in the Small Data Area (SDA).

#pragma ghs section sect="*name*"

The term section refers to an area of memory allocated at link time. The compiler categorizes all functions, variables and data into sections. Each section has a default name. This pragma allows you to rename the section, giving you more control over the location of objects in memory.

The following list displays sections which may be renamed (Note: not all sections are used on all processors). Normally, the following keywords are the same as the section's default name that the compiler uses:

text
rodata
data
rozdata
sdata
bss
tdata
sbss
zdata
zbss
rosdata

The keywords may be understood as follows**:**

text	Any function or other executable code.
data	Initialized data.

bss	Uninitialized or zero initialized data.
ro	Read-only
s	Small Data Area (SDA)
t	Tiny Data Area (TDA)
z	Zero Data Area (ZDA)

(Thus, **rozdata** refers to Read-only Zero Data Area.)

No existing section names may be used. For example, it is not allowed to rename text=".data". If the output is COFF, section names must be no more than 8 characters in length. Other than this, there are no restrictions on the names that may be used for sections. Most default section names begin with a period, but this not a requirement.

The syntax of #pragma ghs section sect="name" allows more than one section to be renamed on a single line, as follows:

```
#pragma ghs section text="text1" data="data1" bss="bss1"
```

To revert entirely to the default section names (except text), use the abbreviated form with no arguments:

```
#pragma ghs section
```

To revert an individual section to its default name, use the keyword default, without quotation marks:

```
#pragma ghs section data=default
```

In most environments, it is not allowed to have more than one text section. This means that #pragma ghs section text="text1" must appear in the file before any functions. It also means that #pragma ghs section text=default is not allowed after the text section has been renamed. In fact, once any functions have been seen, the abbreviated directive #pragma ghs section will NOT revert the name of the text section to the default, although it will revert all other sections to their defaults.

This pragma should be used outside of any function.

#pragma ghs startdata
#pragma ghs enddata

Forces any variables declared between the two pragmas to be allocated and referenced in normal data areas, outside any Small Data Area (SDA) or Zero Data Area (ZDA).

#pragma ghs startsda
#pragma ghs endsda

Forces any variables declared between the two pragmas to be allocated and referenced in the Small Data Area (SDA).

#pragma ghs starttda (CERTAIN PROCESSORS ONLY)
#pragma ghs endtda

Forces any variables declared between the two pragmas to be allocated and referenced in the Tiny Data Area (TDA).

#pragma ghs startzda
#pragma ghs endzda

Forces any variables declared between the two pragmas to be allocated and referenced in the Zero Data Area (SDA).

#pragma ghs zda=*all*
#pragma ghs zda=*size*

Specifies the threshold size for variables placed in the Zero Data Area (SDA).

#pragma ident "*string*"

This causes *string* to be placed in the output file as a comment. The exact behavior varies between systems.

#pragma inline *function-list* (C ONLY)

This pragma takes a list of comma separated function names. It will cause all the listed functions to behave as if the keyword __inline was used to declare the function. This pragma is enabled with the **-pragma_asm_inline** and

-japanese_automotive_c command line options. (See also **#pragma ghs inline**.)

#pragma intvect *intfunc integer_constant* (CERTAIN PROCESSORS ONLY)

This causes a pointer to the function *intfunc* to be output at the address specified by *integer_constant*. This pragma is intended to support interrupt vectors. For it to be useful, you must implement an interrupt handler which loads the function pointer from the specified address and transfers control to that address. *intfunc* should be the name of a function declared as an interrupt function somewhere in the application. No verification is made that function is the name of a valid interrupt function. To enable this pragma, use the **-japanese_automotive_c** command line option.

#pragma pack(*n*)
#pragma pack()

Controls packing of an individual structure. Causes all structs, unions, and classes declared after the pragma to be laid-out in memory such that no data member has alignment greater than *n* bytes. *n* must be 1, 2, 4, 8, or 16.

#pragma weak

1. **#pragma weak** *foo*

#pragma weak *foo* has 2 different uses:

a) It can be used with external references:

```
#pragma weak foo
extern int foo;
```

In this usage, *foo* must not be defined in the file containing the pragma, although it may be referenced. If *foo* is defined in another file, the pragma has no effect. If *foo* is not defined in any file, the pragma prevents an `undefined symbol` error in the linker and causes the address of *foo* to be zero. (Exception: In **-pic** or **-pid** mode, *foo* will have the address `0` plus an adjustment for the text or data segment offset.)

b) It can be used with initialized global definitions:

```
#pragma weak foo
int foo=1;
```

If *foo* never appears in any other file, or if *foo* only appears in external references such as

```
extern int foo;
```

in other files, then the #**pragma** has no effect. If a definition of *foo*, either initialized or common, appears in any other file, then the effect is as if *foo* was declared as

```
extern int foo;
```

2. #pragma weak *foo=bar*

This directive is used to create a globally defined symbol, *foo*, which has the same address as the globally defined symbol bar. The symbol *foo* should not be declared in the C file at all, and the symbol bar must be declared.

No reference to *foo* should appear in the file, since *foo* is not declared.

At link time, *foo* is identical to bar unless there is another global definition of *foo* in another file. In that case, the weak definition of *foo* disappears and is replaced by the global definition of *foo* from another file.

This is normally used to provide backwards compatible aliases to symbols which have been renamed in a new version of software:

```
#pragma weak oldname=name
int name()
{
    return 5;
}
```

If any old software depends on the existence of *oldname*, it will continue to work as if the function is called *oldname*. However, if all references to *oldname* which expect the old meaning are removed, the symbol effectively disappears and can be reused for another purpose.

C++ ONLY PRAGMA

#pragma hdrstop

It enables the user to specify where the set of header files are subject to precompilation ends.

#pragma instantiate

Causes a specified entity to be instantiated.

#pragma can_instantiate

Indicates that a specified entity can be instantiated in the current compilation, but need not be. This is used in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity turns out to be required.

#pragma do_not_instantiate

Suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition will be supplied.

#pragma no_pch

May be used to suppress precompiled header processing for a given source file.

THE C PREPROCESSOR

Green Hills C includes a built-in preprocessor. Preprocessing and compilation are normally performed in a single pass, improving compiler performance. The C Language mode affects behavior of the preprocessor. Generally, in K+R and Transition mode preprocessing follows the traditional UNIX rules and in Permissive ANSI and Strict ANSI modes, preprocessing follows the ANSI standard.

PREPROCESSOR OUTPUT FILE

Preprocessed output may be sent to standard output using the **-E** compiler driver option. Or it may be written to the file *filename*.**i** using the **-P** compile-time option. Normally, preprocessing is performed concurrently with compilation and no intermediate output is generated.

PREDEFINED SYMBOLS IN C

SYMBOLS REQUIRED BY ANSI C

The ANSI standard requires the preprocessor to define 5 symbols. These symbols and their values are shown in the table below. All of them except **_ _STDC_ _** are defined in all modes of C. **_ _STDC_ _** is defined in Permissive ANSI and Strict ANSI modes only.

Symbol	Value
_ _STDC_ _	0 in Permissive ANSI mode 1in Strict ANSI mode undefined in K+R and Transition modes
_ _FILE_ _	The name of the current source file, enclosed in quotes, e.g. **“file.c”**
_ _LINE_ _	The current line-number as an integer.
_ _DATE_ _	The current date, enclosed in quotes; it must always be ten characters. For example, **“Apr 01 2001”**
_ _TIME_ _	The current local time, enclosed in quotes. For example, **“11:46:51”**

Table 2 **ANSI C Predefined Symbols**

ADDITIONAL SYMBOLS PROVIDED BY GREEN HILLS C

The preprocessor defines a number of symbols by default. Different symbols are defined depending on the particular compiler and the options specified. In this way you can write code which will compile differently depending on the use of the **#ifdef** preprocessor directive. All preprocessor symbols are listed below.

The ANSI standard for C requires that all symbols which are predefined by the compiler begin with one or two underscores (_). Prior to the ANSI standard, C preprocessors defined symbols with names like **ghs** or **unix** which could conflict with symbol names in the user's program. In the list below, all symbols have 2 leading underscores and are defined in all modes. In K+R, Transition, and Permissive ANSI modes, some symbols are also defined without the two leading underscores; those symbols have a ***** in the second column. Recent preprocessors use 2 leading underscores and 2 trailing underscores. Symbols listed below which are available in all 3 forms (**ghs**, **_ _ghs**, and **_ _ghs_ _**) are marked with ******.

All of the symbols have the value 1 except **_ _ghs_alignment**, explained below. The general symbols are:

Symbol		Description
__ghs__	**	Any Green Hills Software compiler
__LANGUAGE_C__	**	Language is C (as opposed to C++ or assembly language)
__PROTOTYPES__		All modes of C except K+R mode
__BigEndian	*	Big Endian byte order
__LittleEndian	*	Little Endian byte order
__IeeeFloat	*	IEEE-695 Floating Point format
__SoftwareFloat		All floating point is done with integer instructions
__SoftwareDouble		Double precision floating point uses integer instructions
__NoFloat	*	No Floating Point mode
__ghs_alignment		Alignment of the type double. Has a value of either 1, 2, 4, or 8

Table 3 **General Symbols**

Symbol		Description
__ghs_packing		Represents value specified with **-Zp***n* option on the command line or with the **Structure Packing** button. Has a value of either 1, 2, 4, or 8. This is NOT set at all unless packing is explicitly set by the user either with **-Zp1** or with the **Structure Packing** button. This value is not changed by the **#pragma pack** directive.
__ghs_pic	*	Position Independent Code for embedded
__ghs_pid	*	Position Independent Data for embedded
__ghs_sda	*	Small Data Area optimization supported
__ghs_tda	*	Tiny Data Area optimization supported
__ghs_zda	*	Zero Data Area optimization supported
__Ptr_Is_Signed		Pointers are Signed
__Ptr_Is_Unsigned		Pointers are Unsigned
__Char_Is_Signed		Type char is signed char
__Char_Is_Unsigned		Type char is unsigned char
__Wchar_Is_Signed		Type wchar_t is signed short, int, or long
__Wchar_Is_Unsigned		Type wchar_t is unsigned short, int, or long
__Wchar_Is_Short__		Type wchar_t are short or unsigned short
__Wchar_Is_Int__		Type wchar_t are int or unsigned int
__Wchar_Is_Long__		Type wchar_t are long or unsigned long
__Int_Is_64		Type int is 8 bytes
__Int_Is_32		Type int is 4 bytes
__Long_Is_64		Type long is 8 bytes
__Long_Is_32		Type long is 4 bytes
__LL_Is_64		Type long long is 8 bytes
__Ptr_Is_64		Pointers are 8 bytes
__Ptr_Is_32		Pointers are 4 bytes
__Reg_Is_64		CPU has 64-bit registers
__Reg_Is_32		CPU has 32-bit registers

Table 3 **General Symbols**

The general target operating system symbols are:

Symbol		Description
__bsd	*	Bsd 4.x and SunOS 4.x
__DGUX__	**	Data General DG/UX
__sco	*	SCO UNIX
__sun	*	Sun Microsystems SunOS 4.x or Solaris 2.x
__sysV	*	UNIX System V.3
__sysV4	*	UNIX System V.4 or Solaris 2.x or DG/UX
__sysV4pic	*	Position Independent code on any of above
__windows	*	Microsoft Windows 3.1
__msw	*	Microsoft Windows 3.1
__VXWORKS	*	Wind River VxWorks
__unix__	**	Any version of UNIX

***Table 4* Target Operating System Symbols**

TARGET PROCESSOR PREDEFINED SYMBOLS

Target predefined symbols for a variety of architectures are tabulated below.:

Symbol	Description
__i386	Intel 80386, i486, or Pentium
__i486	Intel i486 or Pentium
__Japanese_Automotive_C	Japanese Automotive C
__ghs_thread_safe	True for velOSity and INTEGRITY
__ARM6	ARM 6 CPU variant
__ARM7	ARM 7 CPU variant
__ARM7m	ARM 7m CPU variant
__ARM7tm	ARM 7tm CPU variant
__ARM8	ARM 8 CPU variant
__ARM9	ARM 9 CPU variant

***Table 5* Target Processor Symbols**

Symbol	Description
__ARM7500fe	ARM 7500fe CPU variant
__StrongARM	StrongARM mode
__THUMB	When either THUMB code or libraries are selected
__GlobalRegisters=*n*	Corresponds to settings of -globalreg=options. *n* is a positive integer or 0.
__GlobalFloatingPointRegisters=*n*	Corresponds to setting of -globalfpreg=options. *n* is a positive integer or 0.
__MicroRAD	ARM's MicroRAD mode
__NDR	NDR CPU
__nCPU	nCPU
__MCore__	MCore CPU
__M200	MCore 200 CPU variant
__M300	MCore 300 CPU variant
__ColdFire	Motorola ColdFire
__MCF5100	NOTE: Motorola, but does NOT imply ColdFire!
__MCF5202	Motorola ColdFire 5202 CPU variant
__MCF5203	Motorola ColdFire 5203 CPU variant
__MCF5204	Motorola ColdFire 5204 CPU variant
__MCF5206	Motorola ColdFire 5206 CPU variant
__MCF5206E	Motorola ColdFire 5206E CPU variant
__MCF5207	Motorola ColdFire 5207 CPU variant
__m68k__	Motorola 68xxx, all CPU variants
__m88k__	Motorola m88000, all CPU variants
__m88110	Motorola m88110 CPU variant
__mc68000, __MC68000 *	Motorola 68xxx, all CPU variants
__mot68k *	Motorola 68xxx in Oasys assembler mode
__motcoff *	Motorola 68xxx in COFF mode

Table 5 **Target Processor Symbols**

Symbol		Description
__motelf	*	Motorola 68xxx in ELF mode
__m68000	*	Motorola 68000 CPU variant
__m68010	*	Motorola 68010 or 6830x CPU variants
__m68020	*	Motorola 68020 CPU variant
__m68030	*	Motorola 68030 or 68340 or 68360 CPU variants
__m68332	*	Motorola 6833x CPU variants
__m68360	*	Motorola 68360 CPU variant
__m68lc040	*	Motorola 68LC040 CPU variant
__m68ec040	*	Motorola 68EC040 CPU variant
__m68040	*	Motorola 68040 CPU variant
__m68lc060	*	Motorola 68LC060 CPU variant
__m68ec060	*	Motorola 68EC060 CPU variant
__m68060	*	Motorola 68060 CPU variant
__m68881	*	Motorola 68881 CPU variant
__m68882	*	Motorola 68882 CPU variant
__mips	*	SGI MIPS, all CPU variants
__mips2	*	MIPS R3000 or R3900
__mips64	*	MIPS with 64-bit integer registers
__MIPSEL		MIPS Little Endian
__MIPSEB	*	MIPS Big Endian
__mipsf64	*	MIPS with 64-bit floating point registers
__ns32000	*	National Semiconductor 32000
__R300__		MIPS R300
__R3000__		MIPS R3000
__R3700__		MIPS R3700
__R3750__		MIPS R3750
__R3900__		MIPS R3900

Table 5 **Target Processor Symbols**

Symbol	Description
__R3900e__	MIPS R3900e
__R4000__	MIPS R4000
__R4100__	MIPS R4100
__R4200__	MIPS R4200
__R4100__	MIPS R4100
__R4200__	MIPS R4200
__R4300__	MIPS R4300
__R4400__	MIPS R4400
__R4600__	MIPS R4600
__R4650__	MIPS R4650
__R4700__	MIPS R4700
__RH32__	TRW RH32
__PowerPC *	PowerPC
__ppc *	PowerPC
__ppc401	PowerPC 401 CPU variant
__ppc403	PowerPC 403 CPU variant
__ppc500	PowerPC 500 CPU variant
__ppc555	PowerPC 555 CPU variant
__ppc601	PowerPC 601 CPU variant
__ppc602	PowerPC 602 CPU variant
__ppc603	PowerPC 603 CPU variant
__ppc603e	PowerPC 603e CPU variant
__ppc604	PowerPC 604 CPU variant
__ppc604e	PowerPC 604e CPU variant
__ppc740	PowerPC 740 CPU variant
__ppc750	PowerPC 750 CPU variant
__ppc821	PowerPC 821 CPU variant

***Table 5* Target Processor Symbols**

Symbol		Description
__ppc823		PowerPC 823 CPU variant
__ppc860		PowerPC 860 CPU variant
__ppc8240		PowerPC 8240 CPU variant
__ppc8260		PowerPC 8260 CPU variant
__ppcec603e		PowerPC ec603e CPU variant
__rs6000c		PowerPC rs6000c CPU variant
__SH7000	*	For SH1, SH2, SH3, SH-DSP
__SH7400	*	For SH-DSP
__SH7600	*	For SH2
__SH7700	*	For SH3
__SH_1	*	For SH1
__SH_1_PLUS	*	For SH1+
__SH_2	*	For SH2
__SH_3	*	For SH3
__SH_4	*	For SH4
__SH_DSP	*	For SH-DSP
__sparc	*	Sun SPARC
__sparclite	*	Sun SPARClite
__danlite	*	DANlite chip
__ST100		For ST100
__GP16		For ST100
__StarCore__		For StarCore
__Tricore	*	For TriCore (NOTE: "c" is not capitalized here)
__Tricore_Rider_A	*	For TriCore (NOTE: "c" is not capitalized here)
__Tricore_Rider_B	*	For TriCore (NOTE: "c" is not capitalized here)
__V800	*	NEC V800 Series, all CPU variants
__V810	*	NEC V810 CPU variant

Table 5 **Target Processor Symbols**

Symbol		Description
__V810U	*	NEC V810U CPO variant
__V830	*	NEC V830 CPU variant
__V830R	*	NEC V830R CPU variant
__V850	*	NEC V850 CPU variant
__V850E	*	NEC V850E CPU variant

***Table 5* Target Processor Symbols**

For the i960, there are three levels of predefined symbols. The top level contains four symbols that are defined for all 960 processors: **__i960**, **i960**, **__i80960**, and **i80960**. The next two levels include families of processors and specific processors:

Symbols *		Description
__i960KA,	__i960_KA,	960KA processor
__i960KB,	__i960_KB,	960KB processor
__i960KX,	__i960_KX,	KX family (960KA, 960 KB)
__i960SA,	__i960_SA,	960SA processor
__i960SB,	__i960_SB,	960SB processor
__i960SX,	__i960_SX,	SX family (960SA, 960SB)
__i960CA,	__i960_CA,	960CA processor
__i960CF,	__i960_CF,	960CF processor
__i960CX,	__i960_CX,	CX family (960CA, 960CF)
__i960JA,	__i960_JA,	960JA processor
__i960JF,	__i960_JF,	960JF processor
__i960JD,	__i960_JD,	960JD processor
__i960RD	__i960RD	960 RD processor
__i960RP,	__i960_RP,	960RP processor
__i960JX,	__i960_JX,	960JX processor
__i960HA,	__i960_HA,	960HA processor

***Table 6* Target Processor Symbols for 960**

Symbols *	Description
__i960HD, __i960_HD,	960HD processor
__i960HT, __i960_HT,	960HT processor
__i960HX __i960HX	HX family (960HA, 960HD, 960 HT)

***Table 6* Target Processor Symbols for 960**

***** In K+R, Transition, and Permissive ANSI modes, this symbol is also defined without the 2 leading underscores.

****** This symbol is always defined with and without the 2 trailing underscores. In K+R, Transition, and Permissive ANSI modes, this symbol is also defined with neither the leading nor trailing underscores.

TYPE QUALIFIERS

There are two new type qualifiers in all modes except K+R mode: **volatile** and **const**.

VOLATILE

When optimizations are turned on with Standard Optimizations (**-O**) in Permissive ANSI or Strict ANSI modes, Memory Optimization (**-OM**) is turned on automatically (see section Memory Optimization with -OM). Memory Optimization means that the compiler may assume that memory locations only change under the control of the compiler. This assumption is usually, but not always, true. For example, it is not true for memory which is updated by interrupt routines, or for memory-mapped I/O. When the compiler is allowed to make this assumption, it may avoid and/or delay reads or writes to memory locations by maintaining a copy of the memory location in a register. This is generally much faster since reads and writes to registers are faster than read and writes to memory. The **volatile** qualifier specifically turns off **Memory Optimization** for the indicated variables, indicating that these variables may change and the compiler can not assume otherwise. This allows you to still use **Memory Optimization** when only some of your variables may change.

CONST

The compiler will give a compile-time error for any attempt to modify an object declared **const**. This qualifier also provides the compiler with additional information for use in optimizations. Wherever the value of a **const** variable is visible, the optimizer makes full use of the fact that this variable is simply a named constant value, combining it with other constants at compile time, and performing other simplifications. Even when the value of a **const** is not visible, the optimizer can make use of the fact that the variable is invariant to resequence statements and instructions or to move them outside of loops.

When **const** is used with **volatile**, even if the value of the object is visible, the compiler will never replace a use of the object with its value.

STRUCTURE AND UNION ASSIGNMENT AND COMPARISONS

Two structures or unions with the same type may be assigned or compared for equality or inequality. Assignment of one structure or union to another is done with a memory copy of the data. Comparison is done, on a bit by bit basis, on the entire structure or union, including padding bytes.

If there are padding bytes between fields or members of a structure or union due to memory alignment requirements, those holes cannot be accessed by means of the structure or union. Note also that global variables will always be initialized to zero so the holes will always be zero, but local variables may have random data in the holes. Therefore, two structures or unions with the same values for every field may not be equal when compared. For structures or unions that will be compared, it is important to either have no holes in the memory representation or to explicitly initialize each such variable with a structure assignment from a global variable known to have zeros in the holes.

A structure or a union may be passed as an argument to a function without restriction. The structure or union is copied when it is passed, however, so passing a very large structure or union is much less efficient than passing a pointer. For this reason we recommend that pointers be passed where possible.

BIT FIELDS

Bit fields in C have a base type which determines whether the field is signed or unsigned, its alignment, and its maximum size.

ANSI C LIMITATIONS

ANSI C requires that the base type of a bit field be either **int**, **signed int**, or **unsigned int**. This restriction is enforced in Strict ANSI mode and is ignored without warning in all other modes.

SIGNED VERSUS UNSIGNED BIT FIELDS

If the base type is either **char**, **short**, **int**, or **long**, without either the signed or unsigned qualifier, compile-time options **-signedfield** and **-unsignedfield** can be used to select whether the bit field is signed or unsigned. The default on each system is chosen for efficiency and compatibility with other important implementations on that system. It is not safe to assume that bit fields will be signed or unsigned when using different C compilers, even between Green Hills C compilers for different systems.

The choice of signed versus unsigned bit fields can significantly affect code efficiency. Unless the processor has special instructions for this purpose, it may require 2 or even 3 instructions to extract the bits of a signed field whenever that field is evaluated. This is because the bits must be sign-extended to the size of an int, which may require 2 arithmetic shifts.

Another difficulty with signed bit fields is seen in the following example:

```
int i;
struct {int x:2; } y;
y.x = 2;
i = y.x;
if (y.x > 0) func();
```

In this example, if **x** is an unsigned field, i will have the value 2 and **func()** will be called. If **x** is a signed field, i will have the value -2 and **func()** will not be called. This is because the range represented by a signed field with 2 bits is from -2 to 1. The value 2 is out of range which is not detected. Instead it is interpreted as -2 when **y.x** is later evaluated.

SIZE AND ALIGNMENT OF BIT FIELDS

There is a close relationship between the base type of a field and the space it occupies. First, an individual bit field can never have more bits than its base type.

Second, a bit field must always fit entirely within a memory location that could hold its base type. It cannot cross a boundary that a simple variable of that type cannot cross. Thus, a field of type char can be placed in any memory location, but cannot cross a byte boundary. For example, variable **w** below requires 3 bytes because fields **a** and **b** do not fit in a single byte. Therefore 4 bits of padding are inserted before field **b**. Similarly **b** and **c** do not fit in a byte and 2 bits of padding are inserted before **c**.

```
struct {
    char a:4;
    char b:6;
    char c:5;
} w;
```

Variable **x** below requires only 2 bytes because the total size of the 3 fields is less than the size of a short.

```
struct {
    short a:4;
    short b:6;
    short c:6;
} x;
```

Third, the alignment of a bit field is determined by its base type. Padding may be inserted so that the field and the structure containing it is properly aligned. In the examples above, **w** has the alignment of a char which is always 1 byte, and **x** has the alignment of a short which is either 1 byte or 2 bytes. (Refer to your *Development Guide* for information on the size and alignment of C and C++ Data Types.) Neither **w** nor **x** would be any larger due to alignment, but **y** and **z** below will have an extra byte of padding added if short is 2 byte aligned.

```
struct {
    char byte;
    short a:4;    {padding inserted before ‘a’ for
                   alignment}
    short b:6;
    short c:6;
} y;

struct {
    short a:4;
```

```
    short b:6;
    short c:6;
    char byte;    {padding added after ‘byte’ for
                   alignment}
} z;
```

OLD-FASHIONED SYNTAX

In K+R and Transition modes there are two forms of old-fashioned syntax which are still supported. The first is initialization of variables without an = (such as **int x 5** in place of **int x = 5**). The second is reflexive operators with the = first (such as **x =+ 5** instead of **x += 5**). The compiler always gives one of the following warnings when it recognizes these old forms:

```
warning: Old-fashioned initialization
```

or

```
warning: Old-fashioned assignment operator
```

Alternatively, the option **-Xnooldfashioned** causes the compiler to completely reject the old-fashioned syntax forms, and instead to behave exactly as it would in Permissive ANSI and Strict ANSI modes. In those modes **int x 5** is illegal and **x =+ 5** is interpreted as **x=5**, **x =& y** is interpreted as **x = (&y)**, and **x =* p** is interpreted as **x = (*p)**. The following table shows examples of the old and the new syntax:

Type	Old syntax	New syntax
Initializers	int x 5,y 6;	int x=5,y=6;
Operators	x =+ 5	x += 5
	x =- 6	x -= 6
	x =* 9	x *= 9
	x =/ 4	x /= 4
	x =& y	x &= y
	x =| 7	x |= 7
	x =^ 1	x ^= 1

Table 7 **Old vs. New Syntax**

Type	Old syntax	New syntax
	x =% z	x %= z
	x =<< 3	x <<= 3
	x =>> 5	x >>= 5

Table 7 **Old vs. New Syntax**

FUNCTIONS RETURNING STRUCTURES OR UNIONS

With most Green Hills C compilers, other than C-68000 and the C-to-C translator, functions that return structures or unions are handled in a reentrant manner. Consider the following example:

```
struct s_t {
    int i;
    float x;
} s;

struct s_t init_s(int i, float x)
{
    struct s_t ret;
    ret.i = i;
    ret.x = x;
    return ret;
}

main(void)
{
    s = init_s(33, 4.2);
}
```

In **main**, the compiler creates a temporary variable of type **struct s_t** and passes its address to **init_s**. When **init_s** executes the return statement, the contents of **ret** are copied to this temporary structure. In **main()**, after the call, the contents of the temporary are copied into **s**.

This method is completely safe and reentrant. Often more efficient code would result if the C program were written like this instead:

```
struct s_t {
    int i;
    float x;
} s;

void init_s(struct s_t *p, int i, float x) {
p->i = i;
p->x = x;
}

main(void) {
    init_s(&s, 33, 4.2);
}
```

In C-68000 and the C-to-C translator, structure return is non-reentrant. Instead of having the caller allocate a temporary variable, the function **init_s** itself allocates the variable, using permanent memory. The function **init_s** returns a pointer to this variable and the caller, main, copies it to **s**. The code is slightly more efficient than the reentrant form, but it has the problem that the return value can be corrupted if **init_s** is invoked again after it has returned, but before the caller has had time to copy the value. This could happen if **init_s** is called by an interrupt routine, for example. The second call to **init_s** would modify the return value of **init_s** and return. Then the first caller to **init_s** would use the value returned by the second call, not the first. This problem can be avoided by rewriting **init_s** as shown above.

ENUMERATED TYPES

Variables of type **enum** actually have the type **int**. Similarly, the members of an enumerated type are essentially named constants of type int. The compile-time option **-shortenum** allows the compiler to select the smallest predefined type (including unsigned types) that allows representation of all listed values.

The ANSI standard does not call for strict type checking with respect to enumerated types. In fact, a variable of enumerated type is considered equivalent to a normal int and most operations which are allowed on **int** are allowed on enumerated types, including assignment of a variable or member of one enumerated type to a variable or parameter of another type.

FUNCTIONS WITH VARIABLE ARGUMENTS

Green Hills C supports functions with variable parameters. In K+R mode, this is done using the **varargs** facility. In Transition, Permissive ANSI, and Strict ANSI, this is done using the **stdarg** facility. The two implementations are different and incompatible. You must take care to use the correct facility for the mode of C in use.

An important limitation of variable parameters is that the types **char**, **short**, and **float** are not supported. When a function with variable parameters is called, the caller promotes expressions of these types to **int** and **double**. Therefore, inside a function with variable parameters, only variables of type **int**, **long**, **double**, and **pointer** should be expected as parameters.

THE VARARGS FACILITY

To use the **varargs** facility you must do the following:

1. The line **#include <varargs.h>** must appear before the first function definition.
2. The last parameter of a variable argument list function must be named **va_alist**.
3. The last parameter declaration of a variable argument list function must be **va_dcl**.
4. There must not be a semicolon between **va_dcl** and the initial left brace (**{**) of the function.
5. There must be a variable declared in the function of type **va_list**.
6. The varargs facility must be initialized at the top of the function by passing the variable of type **va_list** to a call of the macro **va_start**.
7. To obtain the variable arguments to the function, in left to right order, the macro **va_arg** is invoked once for each argument. The first argument to the macro **va_arg** is the variable of type **va_list**. The second argument is the type of the current argument of the function. The **va_arg** macro returns the value of the current argument of the function.
8. The **varargs** facility must be terminated by passing the variable of type **va_list** to a call of the macro **va_end** at the end of the function.

EXAMPLE:

```
#include <varargs.h>
/* Return the sum of a variable number of "int"
arguments */
Sum(n, va_alist)
int n;
va_dcl/* steps 3 and 4 */
{
    va_list params;  /* step 5 */
    int ret = 0;
    va_start(params);  /* step 6 */
    while (n-- > 0)
    ret += va_arg(params,int);  /* step 7 */
    va_end(params);  /* step 8 */
    return(ret);
}
```

THE STDARG FACILITY

The **stdarg** facility has some additional limitations. First, every function with variable parameters is required to have at least one fixed parameter. Second, a prototype for the function must appear before its first invocation.

To use the **stdarg** facility you must do the following:

1. The line **#include <stdarg.h>** must appear before the first function definition
2. In the function definition, at least one fixed parameter must appear in the parameter list before the **...**
3. The last parameter in the function's parameter list must be **...**
4. There must be a variable declared in the function of type **va_list**.
5. The **stdarg** facility must be initialized by invoking **va_start** at the beginning of the function with 2 parameters: the variable of type **va_list** and the last fixed parameter in the function parameter list.
6. To obtain the variable parameters to the function in left to right order, invoke the macro **va_arg** once for each parameter. The first parameter to **va_arg** is the variable of type **va_list**. The second parameter is the type of the current parameter of the function. The **va_arg** macro returns the value of the current parameter of the function.

7. The **stdarg** facility must be terminated by invoking the macro **va_end** with the variable of type **va_list** as its parameter.

EXAMPLE:

```
#include <stdarg.h>
/* Return the sum of the parameters, using the first
parameter as a key to determine the types of the
parameters. */

double sum(const char *key, ...)
{
    double ret = 0;
    va_list ap;
    va_start(ap, key);

    while (*key) {
        switch (*key++) {
        case 'i': ret += va_arg(ap, int); break;
        case 'l': ret += va_arg(ap, long); break;
        case 'd': ret += va_arg(ap, double); break;
        case 'p': ret += *(va_arg(ap, int*)); break;
        }
    }
    va_end(ap);
    return ret;
}
```

ASM STATEMENT

The asm statement generates in-line assembly code and can be used anywhere a statement can be used within a function and anywhere a declaration can be used outside of a function. There are two spellings, **_ _asm** and **asm**, but only **_ _asm** is recognized in Strict ANSI mode. The option **-noasm** prevents the compiler from recognizing **asm** as a keyword in other modes, allowing a variable or function named **asm** to be declared. The keyword **_ _asm** is always recognized, even with the **-noasm** option.

_ _asm (" *any_line_of_assembly_language*");

or

asm (“ *any_line_of_assembly_language*”);

The entire contents of the string will be passed through to the assembly language output file, in the same position as it appears in the C source file. If the underlying assembler requires a space or tab before the opcode, this tab or space must appear in the string.

For example:

```
asm (“ sethi %hi(L16),%o0“);
```

This statement drops the **sethi** instruction into the assembly code generated by the compiler, corresponding exactly to where the compiler found it in the source code. The compiler will, by default, issue the following message for every asm statement found in the file:

```
warning: asm statement not portable
```

This warning may be suppressed with the **-noasmwarn** option.

It is important to note that the compiler uses assembly language instructions and directives throughout the file without concern for possible interactions with user **asm** statements. Furthermore the allocation of variables to registers and memory may vary from one compilation to another. Therefore the use of the **asm** statement is considered extremely non-portable and may be difficult to maintain.

JAPANESE AUTOMOTIVE C

Green Hills Software supports Japanese Automotive C for all processors. 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.** Please refer to the "General Pragma" section for a full description.

-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 always have type **int** or **unsigned int**

-noasmwarn
: Prevents a warning from being printed for each **asm** statement.

Selecting Japanese Automotive C also enables

#pragma intvect *function integer_constant*

which supports interrupt vectors, by causing a pointer to *function* to be output at the address specified by *integer_constant*. Please refer to the "General Pragma" section for a full description.

Selecting Japanese Automotive C also enforces the following behavior: 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 integer is smaller than the pointer, then the data is reduced from the upper bit.

In addition, selecting Japanese Automotive C enables several built-in functions to control interrupts, for the targets which currently support it:

void _ _DI(*void*);
: Disables all interrupts.

void _ _EI(*void*);
Enables all interrupts.
void _ _set_il(*int n*);
Sets interrupt level to *n*.

C RUN-TIME LIBRARY

On UNIX systems, Green Hills C can use the standard C library. The Green Hills ANSI C Library, **libansi**, is provided for users needing an ANSI C library and for those on non-UNIX systems which do not already have a C library. **libansi** is supplied as either object code or C source code, depending on the environment. See the Appendix, C Runtime Libraries in the Development Guide.

To use the Green Hills ANSI C Library you need a standard Green Hills C compiler license. Under this license, unlimited distribution of programs which are linked with Green Hills ANSI C Library object code is permitted without charge. However, distribution of the Green Hills ANSI C Library source code or object code is not permitted.

COMPILER LIMITATIONS

(U > x)	Theoretically unlimited, and actually tested to limit "**x**"
(U)	Theoretically unlimited

*Theoretically unlimited actually means limited by memory limits

- 15 nesting levels of compound statements, iteration control structures and selection control structures (U > 500)
- 8 nesting levels of conditional inclusion (U > 500)
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration (U > 64)
- 31 nesting levels of parenthesized declarators within a full declarator (U)
- 32 nesting levels of parenthesized expressions within a full expression (U > 200)
- 31 significant characters in an external identifier (U)
- 511 external identifiers in one translation unit (U)
- 127 identifiers with block scope declared in one block (U)

- 1024 macro identifiers simultaneously defined in 1 translation unit (U)
- 31 parameters in one function declaration (U 100)
- 31 arguments in one function call (U > 100)
- 31 parameters in one macro definition (U > 100)
- 31 arguments in one macro call (U > 100)
- 509 characters in a logical source line (U > 3000)
- 509 characters in a character string literal or wide string literal (after concatenation) (U > 3000)
- 8 nesting levels for #included files (64)
- 257 case labels for a switch statement, excluding those for any nested switch statements (U)
- 127 members in a single structure or union (U)
- 127 enumeration constants in a single enumeration (U)
- 15 levels of nested structure or union definitions in a single struct-declaration-list (U)

Chapter

2

MIXING LANGUAGES

HOW THE DRIVER BUILDS A MIXED LANGUAGE EXECUTABLE

With Green Hills compilers, you can mix and match C, C++, FORTRAN, Pascal, and Ada routines in the same executable files, subject to certain constraints.

The Green Hills compiler drivers are compatible. This permits a C driver to compile a FORTRAN module, and a Pascal driver to compile a C++ module. The driver uses the input filename extension to determine the correct language, rather than assuming that the name of the driver determines the source code language.

While compatible during compilation, the various drivers differ during the link phase. To link an application the driver must determine all of the languages in use, in order to know which libraries to include. The driver assumes that every application has modules written in C and assembly language, and further, that there is at least one module written in the driver's default language. If source files written in other languages are on the command line, as indicated by the file extension, then the driver recognizes that those languages exist in the application as well.

Therefore, mixing any one language with C is easy, as the driver always assumes C is in use. In this case, the driver for the language other than C should be used for linking the application, to assure the correct linkage.

To link two languages other than C into a single application, all of the source files are placed on the command line so the driver can compile and link in a single step. This assures giving the driver full information during the link phase.

The most difficult case is where each module must be compiled separately and the link phase is done strictly from object files which come from several different languages. In this case, it is best to use the driver for the language with the most complicated linkage requirements. Specifically, to link C, FORTRAN, and Pascal, use the FORTRAN driver and add the Pascal library at the end of the driver command line. To link C, C++ and either FORTRAN or Pascal, use the C++ driver and place the FORTRAN or Pascal libraries at the end of the driver command line.

Some command line options have different meanings for different languages. The drivers recognize these differences and apply all relevant meanings. For example, the **-C** option governs the handling of comments by the preprocessor in C and C++, but in Pascal and FORTRAN the same option enables array bounds checking. Therefore, specifying **-C** on the driver command line along with files written in C and FORTRAN will have different effects on the different language modules.

THE -LANGUAGE OPTION

The **-language** option facilitates mixing languages in the same program. It is written as:

-language=*language*

where *language* is either **cxx**, **fortran**, or **pascal**. It is not necessary to specify C.

The **-language** option tells the driver that files written in *language* are being mixed with the default language. This option is specified once for each language being mixed. It is not necessary to specify the driver's default language.

EXAMPLE:

Three precompiled object files, **main.o**, **pigeon.o**, and **falcon.o**, are written in C, Pascal, and FORTRAN, respectively. The following command line tells the driver about all three languages when linking:

```
gfc -language=pascal main.o pigeon.o falcon.o
```

Here, the driver knows about FORTRAN because the FORTRAN driver is being used (**gfc**). All drivers assume C, and the **-language=pascal** option informs the driver about Pascal.

To link the same three modules with the C driver:

```
gcc -language=pascal -language=fortran main.o
pigeon.o falcon.o
```

INITIALIZATION OF LIBRARIES

A multiple language application may need to perform input and output in more than one language. With a little care to avoid conflicts between languages, this is fully supported. If input and output will always be performed on different files by each language, then the initialization and deinitialization of each language's runtime routines is handled automatically by the main program in a single language application. Therefore, if the application will only perform I/O in one language other than C, then it is easy to write the main program for the application in that language. For more complex requirements, a main program may be written in C which performs the initialization and deinitialization of the library runtime routines.

NOTE

If libraries are static when you have **main()** in C, then the module in the FORTRAN library which declares **main()** is not brought in so there is no reference to **MAIN_**.

However if the libraries are dynamic, then the following appear: **main()** in C and **main()** in the FORTRAN library (which will never be called). Since there is no FORTRAN PROGRAM, **MAIN_** becomes an undefined symbol, even though it will never be referenced.

MAIN() PROGRAM EXAMPLES

If you implement your own **main()** function, **__gh_initrec()** must be called unless FORTRAN IO is not used in the program. This function initializes the logical UNIT table within the FORTRAN library. It sets Units 5, 6, and 0 to **stdin**, **stdout**, and **stderr** respectively and sets all Units (0-99) to a proper initial state.

A C MAIN() PROGRAM FOR C++

```
main() {
    _main();/* must be first executable line */

    /* rest of main goes here */

    exit(0);/* must be last executable line */
```

```
}
```

A C MAIN() PROGRAM FOR FORTRAN

```
int __gh_argc;
char **__gh_argv;

extern void (*__gh_initrec)();
extern void (*__gh_uninitrec)();

int main(argc, argv)
int argc;
char **argv;
{
    __gh_argc = argc;
    __gh_argv = argv;

    if (__gh_initrec)
                __gh_initrec();

/* rest of main goes here */

    if (__gh_uninitrec)
                __gh_uninitrec();
    return(0);
}
```

A C MAIN() PROGRAM FOR PASCAL

```
main(argc, argv)
int argc;
char **argv;
{
    extern int __argc;
    extern char **__argv;
__argc = argc;
   __argv = argv;/* the 4 lines above must be first */
    /* rest of main goes here */

    __GHSexit(0);/* must be last executable line */
```

```
}
```

A C main() Program for Ada

When using Ada, two procedures facilitate a non-Ada main program, **adainit** and **adafinal**. The procedure **adainit** performs Ada elaborations, and **adafinal** performs Ada finalizations. These two procedures can be called at the beginning and end, respectively, of a non-Ada main program.

Performing I/O on a Single File in Multiple Languages

Some applications benefit from performing input and output on a single file or device from more than one language; an example is pre-opened files. In C, these are **stdin**, **stdout**, and **stderr**. In C++, they are **cin**, **cout**, and **cerr**. In FORTRAN, they are **Units 5**, **6**, and **0** respectively. In Pascal, the first two files are **input** and **output**, and the equivalent of C's **stderr** cannot be used directly.

All languages have full access to these pre-opened files, and input and output can easily be mixed between the languages on these files. However, for the best results, a complete input or output operation is done in a single language. In FORTRAN, a single **READ**, **WRITE**, or **PRINT** statement is a complete operation. In Pascal, a single **read**, **readln**, **write**, or **writeln** call is a complete operation. In C, any call to a library function which performs input or output is a complete operation. If this rule is followed, all data will be output correctly and in the intended sequence. The C library routine **fflush()** flushes the buffer of the pre-opened files in all languages, except in C++. To flush one of these files in C++, use the notation *file*<<**flush**. For example, **cout**<<**flush**.

Performing input and output on a single file which is not preopened is more difficult. It is possible to open the file once in each language and perform input and output independently in each language. In many cases this would be unacceptable, particularly when working with a device rather than a simple file.

It is possible to open a file in FORTRAN and subsequently perform input and output on that file by using the FORTRAN library routines **GETCHAN** and **GETFD**. The FORTRAN function **GETCHAN** takes a single argument which is the **Unit** number of a FORTRAN file and returns a **FILE*** which can then be used with C library routines such as **fprintf()**, **fread()**, **fwrite()**, **fflush()**, **fseek()**, **fstat()** and **fputc()**. Operations on such a file are compatible to the same extent as the three pre-opened files.

The FORTRAN function **GETFD** takes a single argument which is the **Unit** number of a FORTRAN file and returns an integer which can then be used with lower level routines such as **read()**, **write()**, **lseek()**, and **stat()**. Because these low level routines are not compatible with **fprintf()**, **fread()**, **fwrite()**, etc., their use may conflict with the FORTRAN runtime routines.

There is currently no mechanism for performing input and output in C on a file opened in Pascal.

NATIVE UNIX LIBRARIES VERSUS GREEN HILLS LIBRARIES

This section refers only to native UNIX users.

Although the combination of multiple languages in a single application is fully supported, certain differences cannot be avoided between programs written entirely in one language and those written in multiple languages, due primarily to library selection.

The C and C++ languages use the native UNIX math and C libraries by default. The FORTRAN and Pascal languages use the Green Hills math library. ANSI C uses the Green Hills math and C libraries. This means that the combination of FORTRAN and C will cause the entire application to use the Green Hills math library, and the combination of ANSI C with C++ will cause the entire application to use the Green Hills math and C libraries. Therefore, programs written entirely in C or C++ may behave differently than otherwise identical programs written partially in C or C++ and partially in FORTRAN or ANSI C.

CALLING A C ROUTINE FROM FORTRAN

This section shows how to call C subroutines from FORTRAN.

ARGUMENT PASSING

By default, all FORTRAN arguments are passed by reference. Therefore, each parameter in the called C routine must be a pointer of the appropriate type. The following table shows how arguments passed by FORTRAN are received by C:

FORTRAN Passes	C Receives
REAL or REAL*4	float *
DOUBLE PRECISION or REAL*8	double *
INTEGER or INTEGER*4	int *
INTEGER*2	short *
INTEGER*1	signed char *
LOGICAL or LOGICAL*4	long *
LOGICAL*2	short *
LOGICAL*1	signed char *
COMPLEX or COMPLEX*8	struct complex {float realpart, imagpart} *
DOUBLE COMPLEX or COMPLEX*16	struct dcomplex {double realpart, imagpart} *
CHARACTER	signed char * and int (for length)

Table 8 **Passing Arguments from FORTRAN to C**

FORTRAN **CHARACTER** types are a special case. When a C function receives a **CHARACTER** argument by a FORTRAN routine, it receives not only a pointer to the char variable, but also its length, as an **int** (not as an **int ***). This **int** will appear at the end of the argument list. If more than one **CHARACTER** parameter is passed, then an extra **int** for every **CHARACTER** parameter will be passed at the end of the argument list, in the order that the **CHARACTER** parameters are passed. The called C routine must declare one extra variable of type **int** for every FORTRAN **CHARACTER** argument passed in order to receive the information.

For example, a FORTRAN routine calls a C function with two **CHARACTER** parameters and two **INTEGER** parameters:

```
CHARACTER A,B
INTEGER X,Y
CALL NAME(A,X,B,Y)
END
```

The C routine, then, is:

```
name_(char *a, int *x, char *b, int *y, int alen, int
blen)
{}
```

In this routine, the **char *a** points to **CHARACTER A**, **int *x** points to **INTEGER X**, **char *b** points to **CHARACTER B**, **int *y** points to **INTEGER Y**, **int alen** is the length of **CHARACTER A**, and **int blen** is the length of **CHARACTER B**. The extra arguments, **int alen** and **int blen**, appear at the end of the argument list in the order that their corresponding **CHARACTER** parameters were passed (**A** is passed before **B**, so **alen** appears before **blen**).

Although FORTRAN **CHARACTER** string constants are null terminated, **CHARACTER** variables are not. Thus, the character strings **A** and **B** in the above example do not end with an extra 0. However, if the FORTRAN code were changed to the following, the C code could remain the same:

```
CHARACTER A
INTEGER X,Y
CALL NAME(A,X,”this is a string”,Y)
END
```

The string “**this is a string**” will end in an extra zero. However, this 0 will not be counted as part of the string length being passed. So, in the above example, **blen** is 16, not 17.

RETURN TYPES

Called C functions may return values to FORTRAN routines.

SIMPLE RETURN TYPES

An **int** C function must be declared either as **INTEGER (**or **INTEGER*4**) or **LOGICAL** (or **LOGICAL*4**) in the calling FORTRAN routine.

An ANSI C function which returns a **float** must be declared as **REAL** or **REAL*4** in the calling FORTRAN routine.

An ANSI C function which returns a **double** must be declared as **DOUBLE PRECISION** or **REAL*8** in the calling FORTRAN routine.

A non-ANSI C function which returns a **float** or **double** must be declared as **DOUBLE PRECISION** or **REAL*8** in the calling FORTRAN routine.

CHARACTER

Some implementations do not allow functions which return **CHARACTER** types to be written in C. The following description applies only to those implementations, such as Green Hills, which allow this.

A **CHARACTER** type may not be returned directly with a C **return** statement. Instead, when a C function wants to return a FORTRAN CHARACTER result, then two extra arguments are passed to the C function. These arguments appear at the beginning of the argument list. The first argument in the C function must be a **char ***. The character string to be returned should be placed where this argument points. The second argument must be the maximum permitted length of the character string. For example:

```
CHARACTER*9 NAME
CHARACTER*9 A
A=NAME()
PRINT*,A
END
```

The C function is:

```
name_(c, b)
char *c;
int b;
{
    char d[]=”pigeon”;
    int i, len;
    len=strlen(d);
    if (len > b)
    len = b;
    for (i=0; i < len; i++)
```

```
            c[i] = d[i];
    for (i=len; i < b; i++)
            c[i] = ‘ ‘;
}
```

In the FORTRAN routine, the function **name** is not called with any arguments. Since the function is declared as a **CHARACTER** return type, two arguments will be automatically passed. The C function receives these as a pointer to the return location (**char *c**) and the length (**int b**) of the character string. The C function does not use the **return** statement.

COMPLEX AND DOUBLE COMPLEX

Some implementations do not allow functions which return **COMPLEX** or **DOUBLE COMPLEX** types to be written in C. The following description applies only to those implementations, such as Green Hills, which do allow this.

COMPLEX (or **COMPLEX*8**) or **DOUBLE COMPLEX** (or **COMPLEX*16**) types may not be returned with a C **return** statement. When a function is declared to be of one of these types, then one extra argument is passed to the C function. This argument will appear at the beginning of the argument list. The C function must declare a special **struct** in which to put the return information. The first argument in the C function must be a pointer to the previously defined **struct**. Table 8 on page 2-46 lists the necessary struct declarations for these two return types. For example, a FORTRAN routine calling a C function with a COMPLEX return type:

```
COMPLEX A
COMPLEX COMP
A=COMP()
PRINT*,A
END
```

can have the C function:

```
struct complex {float realpart, imagpart;};

comp_ (c)
struct complex *c;
{
    c->realpart=1.9;
```

```
    c->imagpart=4.5;
}
```

In the FORTRAN routine, the function **comp** is not called with any arguments. Since the function is declared as having a **COMPLEX** return type, one argument will automatically be passed at the beginning of the argument list. The C function receives this argument as a pointer to a **struct** to store the return information in (**struct complex *c**). The C function does not use the **return** statement.

ALTERNATE RETURNS

A FORTRAN routine may call a C function using the alternate return conventions. The C routine would use the return statement in the same way a FORTRAN using alternate returns would, except that instead of the FORTRAN **RETURN**, the C program would use **return 0**. For example:

```
X = 9
Y = 3
CALL COMPARE(X,Y,*100,*200,*300)
PRINT*,'Illegal input'
GOTO 99
100 PRINT*,'X < Y'
GOTO 99
200 PRINT*,'X == Y'
GOTO 99
300 PRINT*,'X > Y'
GOTO 99
99 END
```

The C function could be:

```
compare_(a, b)
float *a, *b;
{
 if (*a < 0.0 || *b < 0.0)
  return 0;
 if (*a < *b)
  return 1;
 if (*a == *b)
  return 2;
```

```
  return 3;
 }
```

If **compare** returns a 0 in the above example, the next line after the function call will be executed. If 1 is returned, then line 100 will be the next line executed.

SYMBOL NAMING CONVENTIONS

FORTRAN is not case-sensitive and converts all characters (outside of quotation marks) to lower case. In a FORTRAN program, the symbol names **FALCON**, **Falcon** and **falcon** are all the same item. C is case-sensitive. In a C program, the symbols **FALCON**, **Falcon** and **falcon** are three distinct identifiers. So, only C functions whose names are all lower case are called, unless FORTRAN routines are compile with a **-U** option, making FORTRAN case-sensitive.

FORTRAN also appends an underscore (_) to each function name. To call a C function from a FORTRAN routine, the name of the C routine must end in an underscore. For example, instead of naming a C routine **falcon()**, it is named **falcon_()**. This feature allows calling C routines from FORTRAN via an interface routine. The next section explains this in detail.

CALLING C ROUTINES FROM FORTRAN

Because FORTRAN passes function arguments as pointers, it is not possible to directly call pre-compiled C routines that haven't been explicitly written for FORTRAN. FORTRAN appends an underscore to the end of function names to allow an interface routine of the same name. An interface routine could be called from FORTRAN, and would then call the actual C routine with the correct arguments.

For example, for the following pre-compiled C routine is not possible to call this routine from FORTRAN because **i** and **j** are not pointers:

```
int add(int i, int j)
{
    return i + j;
}
```

However, with the following interface routine in C this routine can now be called from FORTRAN, which in turn calls the real **add** routine:

```
int add_(int *i, int *j)
{
    return add(*i, *j);
}
```

For example:

```
INTEGER ADD
I = ADD(4, 5)
END
```

COMMON BLOCKS

FORTRAN modifies the names of COMMON blocks. All capital letters are converted to lowercase, but the character or characters appended to the name of the common block differ, depending on the compilation mode.

In f77 compatibility mode, a single underscore is appended to COMMON block names. Since this can cause name conflicts between subprogram names and COMMON block names, in VMS compatibility mode, a dollar sign (**$**) is appended instead.

The **-X608** option causes COMMON blocks to be named in the VMS style, with a dollar sign appended. This option can be selected independently of VMS compatibility mode. With this, f77 compatibility mode can be used, and **-X608** can be specified on the command line to name the COMMON block with a dollar sign suffix instead of an underscore.

The **-X608** is usually enabled in VMS compatibility mode. However, f77 style names can be specified while in VMS compatibility mode by specifying **-Z608** on the command line.

An alternate form of VMS style names for environments do not allow dollar signs in names. This is enabled with the **-X402** option and causes two underscores to be appended to COMMON block names instead of one dollar sign. The **-X402** option is ignored unless VMS style COMMON block names are being generated. (**-X402** in f77 mode can be used if **-X608** is specified.)

Mode	Switch	Effect
f77	(default)	block_
	-X608	block$
	-X608 -X402	block_ _
VMS	(default)	block$
	-Z608	block_
	-X402	block_ _

Table 9 **COMMON Block Naming Conventions**

CALLING A FORTRAN ROUTINE FROM C

This section shows how to call FORTRAN subroutines from C.

ARGUMENT PASSING

All FORTRAN parameters are passed by reference, so the corresponding argument in the C call must be a pointer of the appropriate type. The table below shows the argument type that C must pass to correspond to the FORTRAN parameter.

C Passes	FORTRAN Receives
float *	REAL or REAL*4
double *	DOUBLE PRECISION or REAL*8
int *	INTEGER or INTEGER*4
short *	INTEGER*2
signed char *	INTEGER*1
int *	LOGICAL or LOGICAL*4
short *	LOGICAL*2
char *	LOGICAL*1

Table 10 **Passing Arguments from C to FORTRAN**

C Passes	FORTRAN Receives
struct complex {float realpart, imagpart;} *	COMPLEX or COMPLEX*8
struct dcomplex {double realpart, imagpart;} *	DOUBLE COMPLEX or COMPLEX*16
char * and int	CHARACTER

Table 10 **Passing Arguments from C to FORTRAN**

For example, to pass an integer variable **a** from C to FORTRAN, pass **&a**.

Passing a **char** argument to a FORTRAN function is a special case. The C routine must pass not only a pointer to the **char** variable, but also its length, as an **int** (not as an **int ***). This **int** must be passed as the last argument. If more than one **char** is being passed by the C routine, then each one will have a separate **int** associated with it. The **int**s must all appear at the end of the argument list, in the same order that their corresponding **char**s appear. For example:

```
extern int falcon_();
main()
{
    char *c1=”pigeon”;
    char *c2=”sofa sofa”;
    int extra=5;
    int len=falcon_(c1, c2, &extra, strlen(c1),
      strlen(c2));
    printf(“%d\n”,len);
}
```

This C routine passes two **CHARACTER** parameters and one **INTEGER** parameter to a FORTRAN function. It accomplishes this by passing five arguments. The first two are pointers to **char**s being passed (**c1** and **c2**), the third is the **int** being passed (**extra**), and the last two are the lengths of **c1** and **c2**. The corresponding FORTRAN function is:

```
integer function falcon(a,b,x)
character*(*)a
character*(*)b
integer x
```

```
falcon=len(a)+len(b)+x
end
```

RETURN TYPES

FORTRAN functions may return values to C routines.

SIMPLE RETURN TYPES

An **INTEGER** or (**INTEGER*4**) or **LOGICAL** (or **LOGICAL*4**) FORTRAN function must be declared as **int** in the calling C routine.

A **DOUBLE PRECISION** or **REAL*8** FORTRAN function must be declared as **double** in the calling C routine. Since C usually promotes **float** return values to **double**, a **REAL** return value may not be accessible in C. This is not true for ANSI C, however.

CHARACTER

Some implementations do not allow functions which return **CHARACTER** types to be called from C. The following description applies only to those implementations which do allow this.

FORTRAN functions that have a **CHARACTER** return type are special cases. A value is not actually returned to the calling C routine; instead, the C routine must pass two extra arguments in which to store the return values. The first argument passed must be a **char *** to point to the beginning of the return string. The second argument must be an **int** that is the length of the **char ***. All other normal arguments must follow these two. For example:

```
extern void falcon();
main()
{
    char buff[20];
    char xbuff[]=”pigeon”;

    falcon_(buff, sizeof(buff), xbuff,
      sizeof(xbuff)-1);
    printf(“%s\n”, buff);
}
```

Here, two extra arguments are passed, both for a character string being passed. The size of **xbuff** is passed as one short to remove the null character that C will put at the end of the string. The return string will be stored in **buff**. The FORTRAN is then:

```
character*20 function falcon(x)
character*(*) x
falcon=x // ’ sofa sofa’
end
```

This function appends a string to the input string (x), then passes back the new string as the return value.

COMPLEX, COMPLEX*8, DOUBLE COMPLEX, COMPLEX*16

Some implementations do not allow functions which return **COMPLEX**, **COMPLEX*8**, **DOUBLE COMPLEX**, or **COMPLEX*16** types to be called from C. The following description applies only to those implementations which do allow this.

FORTRAN functions that have a **COMPLEX** (or **COMPLEX*8**) or **DOUBLE COMPLEX** (or **COMPLEX*16**) return type are special cases. A value is not actually returned to the calling C routine; instead, the C routine must pass an extra argument in which to store the return value. The first argument passed must be a pointer to a predefined **struct** of the correct type. The return value will be stored in this **struct**. All other arguments must follow this one. For example:

```
struct complex {float realpart, imagpart;};
extern void falcon();
main()
{
    struct complex comp;
    int x=5;

    falcon_(&comp, &x);
    printf(“%f + %fi\n”, comp.realpart,
      comp.imagpart);
}
```

NOTE: For the PowerPC: In order for printf to work correctly, #include needs to be added to the beginning of the function:

```
#include <stdio.h>
struct complex {float realpart, imagpart;};
extern void falcon();
main()
{
    struct complex comp;
    int x=5;

    falcon_(&comp, &x);
    printf("%f + %fi\n", comp.realpart,
      comp.imagpart);
}
```

Here, the returned complex number is stored in **comp**, and **x** is an argument being passed. The FORTRAN function is:

```
complex function falcon(x)
integer x
complex y
y=(0.0 , 2.3)
falcon=y+x
end
```

ALTERNATE RETURNS

The FORTRAN alternate return statements return the corresponding integer to the calling C routine (the simple **RETURN** statement returns a 0 to C). The calling C routine makes appropriate use of these return values. Use of a **switch** statement is recommended. There should be a **case** label corresponding to each valid alternate return, and a **default** case to handle all return values outside the expected range. For example:

```
main()
{
 float x, y;
 int ret;

 x = 9;
```

```
 y = 3;
 ret = compare_(&x, &y);
 switch (ret)
 {
 default: printf("Illegal input\n"()
  break;
 case 1: printf(“x < y\n”);
  break;
 case 2: printf(“x == y\n”);
 break;
 case 3: printf(“x > y\n”);
  break;
 }
}
```

The FORTRAN function would be:

```
SUBROUTINE COMPARE(A,B,*,*,*)
IF (A .LT.  0.0 .OR.  B .LT.  0.0) RETURN
IF (A .LT.  B) RETURN 1
IF (A .EQ.  B) RETURN 2
RETURN 3
END
```

SYMBOL NAMING CONVENTIONS

FORTRAN is not case-sensitive and will convert all characters to lower case. In a FORTRAN program the symbol names **FALCON**, **Falcon** and **falcon** refer to the same item. C is case-sensitive. In a C program, the symbols **FALCON**, **Falcon** and **falcon** are three distinct identifiers.Compiling FORTRAN routines with a **-U** option makes FORTRAN case-sensitive; otherwise, only C functions with lower case names can be called.

FORTRAN also appends an underscore (_) to function names. To call a C subroutine from a FORTRAN routine, the name of the C routine must end in an underscore. For example, a C routine has to be named **falcon_()** instead of **falcon()**. This feature allows calling pre-compiled C routines from FORTRAN via an interface routine, explained in detail in the next section.

COMMON BLOCKS

FORTRAN modifies the names of COMMON blocks. All capital letters are converted to lowercase, but when using a Green Hills FORTRAN compiler, the character or characters which are appended to the name of the common block differ depending upon the compilation mode.

In f77 compatibility mode, a single underscore is appended to COMMON block names. Since this can cause name conflicts between subprograms and COMMON blocks with the same name, in VMS compatibility mode, a dollar sign (**$**) is appended.

The **-X608** option causes COMMON blocks to be named in the VMS style, with a dollar sign appended. This option can be selected independently of VMS compatibility mode. Thus, using f77 compatibility mode and specifying **-X608** on the command line gives the COMMON block a dollar sign suffix instead of an underscore.

Normally, **-X608** is enabled in VMS compatibility mode. However, f77 style names can be specified while in VMS compatibility mode, by specifying **-Z608** on the command line.

An alternate form of VMS style names for environments does not allow dollar signs in names. This is enabled with the **-X402** option and causes two underscores to be appended to COMMON block names instead of one dollar sign. The **-X402** option is ignored unless VMS style COMMON block names are being generated. (**-X402** can be used in f77 mode by specifying **-X608**.)

Mode	Switch	Effect
f77	(default)	block_
	-X608	block$
	-X608 -X402	block_ _
VMS	(default)	block$
	-Z608	block_
	-X402	block_ _

Table 11 **COMMON Block Naming Conventions**

CALLING A C ROUTINE FROM ADA

This section shows how to call C subroutines from Ada.

PRAGMA INTERFACE C

Pragma interface specifies that a subprogram is written in some other language, and the definition of that subprogram resides in a separate object module. Pragma interface is allowed at the place of a declarative item in a package specification. The subprogram specification for which pragma interface is given must appear in the same compilation unit, with the optional *link-name* limited to 62 characters.

For example, to create a link to call C routine “name” in Ada, a package specification has to first be created, containing the Ada declaration of the C routine. The package specification **C_LINK** is:

```
PACKAGE C_LINK IS
   PROCEDURE Name ;
PRIVATE
   pragma interface(C, Name, ”name”) ;
END C_LINK:
```

The corresponding C routine is:

```
void name()
{
   printf(“This routine is called from Ada”);
}
```

ARGUMENT PASSING

Each parameter in the called C routine must be the appropriate type. The following table shows how the arguments passed by Ada are received by C:

Ada Passes	C Receives
INTEGER	int
INTEGER	long
SHORT_INTEGER	short
CHARACTER	char
BYTE_INTEGER	char
FLOAT	float
LONG_FLOAT	double

Table 12 **Passing Arguments from Ada to C**

The previous example can pass an integer and float to the C routine; it is modified to:

```
PACKAGE C_LINK IS
        PROCEDURE Name(A_Integer: INTEGER; A_Float:
FLOAT) ;
     PRIVATE
        pragma interface(C, Name, “name”) ;
     END C_LINK:
```

The corresponding C routine is:

```
void name(int a_integer; float a_float)
{
  printf(“This routine is called from Ada”);
  printf(“This is an integer passed from Ada %d\n”,
  a_integer);
  printf(“This is a float passed from Ada %f\n”,
  a_float);
}
```

Function calls operate in the same manner as procedures. The Function types must be compatible in C and Ada.

ARRAY AND STRING TYPES

For Static Ada Array Types, individual components must be structurally compatible to the corresponding C variable. Dynamic Arrays, however, can be passed from Ada to C using the address of the first element:

```
Dynamic_Array(Dynamic_Array'First)'Address
```

In Ada, information is kept in the record regarding bounds of the array.

C strings are terminated by an ASCII null character, ASCII 16#00#. Passing a string to C is much like passing a Dynamic Array, with the exception of appending an ASCII null character to the end of the string.

For example, for an Ada string declared:

```
My_String: STRING(1 . . 8);
```

To pass this to a C string:

```
My_String(My_String'First)'Address
```

POINTERS AND ADDRESS TYPES

The address convention is identical for Green Hills Ada and C compilers.

INTERFACING PASCAL AND C

This section shows how to interface Pascal and C:

NAMING CONVENTIONS

By default, the names of Pascal external variables, procedures, and functions are accessible from C functions linked with the Pascal program. External Pascal names are accessed by using the same name in C. Green Hills Pascal is case-sensitive by default; however, using the **-s** or **-X59** compile time options make the Pascal compiler case-insensitive, causing it to convert all uppercase character to lowercase. C is always case-sensitive.

When compiling with the **-s** or **-X174** option (Strict ISO mode), the names of Pascal external procedures and functions are changed by appending an additional underscore (_). If this option is used, then to call the Pascal function **Falcon** from C means calling the function **falcon_**. This is the only function of the **-X174** option, while **-s** has many effects.

This option causes all of the C library functions provided with Pascal to become inaccessible.

REDEFINING WRITE OR READ

If a Pascal program redefines the built-in procedure **WRITE** or **READ**, it must be compiled with the **-s** option. The Green Hills C Run-time Library and the UNIX C library use the names **write** and **read** (to which **WRITE** and **READ** are translated by Pascal) for the basic I/O primitives. If the program redefines these names, then very strange results (often infinite loops) occur. The **-s** compile-time option translates these names to **write_** and **read_** instead, so no redefinition will occur. However, under these options communication between Pascal and C or the C Library becomes much more complicated.

C ROUTINES AND HEADER FILES IN C++

The C++ language allows much use of existing C code. Therefore, it is fairly straightforward to call functions written in ANSI C from C++. The syntax of the two languages is very similar and the use of header files has been continued in C++.

By default, the names of functions are encoded or mangled in C++, whereas in C, the names of functions are unchanged. C++ provides the **extern** specifier to identify non-C++ functions so their names will not be mangled. Therefore, this specifier, allows including ANSI declarations for any C functions and then linking with the compiled C object code.

To specify a C declaration:

- Specify or declare functions individually. For example:

```
extern “C” {
    int fclose(FILE *);
    FILE *fopen(const char *, const char *);
```

```
}
```

- Specifies that two functions with external C linkage are to be declared.
- It is easy to include ANSI C in a C++ source. C++ requires prototyped declarations, as does ANSI C. It is not advisable to include non-prototyped declarations since they mean something different in C++. If they are used, any error messages may or may not point to the non-prototype declaration.
- Declare entire header files with **extern**. For example:

```
extern “C” {
    #include <stdio.h>
    #include <string.h>
}
```

- Has the same effect on all the function declarations that appear in **stdio.h** and **string.h** as the previous example has on the two specific functions (**fclose** and **fopen**).
- If code contains both C and C++, then the **extern** statements can be placed within **#ifdef _ _cplusplus** statements. This practice is common within header files. For example:

```
#ifdef _ _cplusplus
  extern "C" {
#endif
  void assert(int);
       void _assert(const char *,const int ,const
        char *);
#ifdef _ _cplusplus
  }
#endif
```

USING C++ IN C PROGRAMS

Many features in the C++ language simplify complicated tasks, for most languages. It makes little sense to attempt to call C++ from C in most cases, since doing so would force the C programmer to reproduce work performed by the C++ compiler. The various implementations and different mechanisms of C++ make porting of C programs which call C++ more difficult.

Inclusion of C++ modules in C programs is not a trivial. C has no support for any of the C++ extensions to the language. The C programmer must manually perform some of the tasks automatically done by a C++ compiler. Some knowledge of the internal mechanics and details of a C++ implementation is necessary, as follows:

- Encoding of C++ names can be a problem. The C++ compiler encodes or mangles function and class member names. Any C++ function or class members called from a C program must be referenced by the encoded or mangled names.
- The way member functions are handled by a C++ compiler must be known. All member functions (except static member functions) have the special object member pointer **this** inserted automatically as the first argument in the parameter list. A C programmer must add the argument **this** manually when calling any member functions from C.
- Special processing is needed to handle constructors and destructors for static objects. On most systems the **main** module has special function calls inserted to insure that all static constructor/destructor calls are made properly. If **main** is not in a C++ module, then the C programmer must manually include calls to **_main** in the C main module. The **_main** code is contained in the C++ library and therefore must be linked into the final executable.
- Finally, after the executable is produced, the postlink program must be run, as is the case with any C++ executable. If no static objects are used in the program, this step is not necessary. There are four global objects in the **iostream** library: **cout**, **cin**, **cerr**, and **clog**. If any of these objects are used in a program, the postlink program must be run on the executable.
- Virtual functions are also handled automatically by a C++ compiler, but involve additional coding to access or use them from a C environment.

FUNCTION PROTOTYPING IN C VERSUS C++

In ANSI C and C++, header files provide prototypes for library functions which enforce a standard interface between the calling program and the called function.

Function prototyping requires that the function declaration include the function return type and the number and type of the arguments. When a prototype is available for a function, the compiler is able to perform argument checking and

coercion on calls to that function. If a prototype is not available for a function when it is called, ANSI C will behave like K&R C. The return type of the function is assumed to be **int**, and actual arguments will be promoted to **int**s, **long**s, **double**s, or pointers as appropriate. In C++, however, it is an error to call a function which has not been declared with a prototype.

Another important difference between ANSI C and C++ is that a non-prototype declaration of a function, such as:

```
char *function_name();
```

has no effect on the number and type of the arguments in ANSI C, but in C++ it is understood as:

```
char *function_name(void)
```

which means that the function has no arguments at all. If the function declaration occurs within the scope of an **extern "C"** declaration, the function has non-C++ linkage and therefore cannot be overloaded. This means that if a traditional K&R style declaration of a function appears in a header file and the **#include** directive which accesses that header file is enclosed in **extern "C" { }**, then it will be impossible to redeclare that function with arguments.

Chapter

3

Writing Portable Code

The C language has been implemented by many vendors on a large collection of machines and systems. One important reason for using C is to simplify the task of building and maintaining software on multiple platforms. But not all features of the C language cater to this goal. C intentionally provides features which behave differently on different systems. For C programs to be truly portable, the programmer must be careful to avoid these non-portable features of the language.

Certain differences between C compilers are vendor specific differences. If all of the C compilers you ever use come from a single vendor you can avoid these differences. Many more of the differences however are particular to the processor and the operating system in use. When porting between different platforms it may be impossible to avoid these differences, except by careful coding.

COMPATIBILITY BETWEEN GREEN HILLS COMPILERS

All Green Hills C compilers follow the same interpretation of the C language, as described in this manual. There are a few features and options which are not available in all Green Hills C compilers. These features and options are described in the *Development Guide* for each compiler. To ensure a C program is portable between all Green Hills C compilers, we recommend using only those features and options described in this manual and the texts mentioned in the preface. Command line options may also be needed to adjust for differences between the default behavior of each Green Hills C compiler.

For information on compatibility with Green Hills compilers for other languages, see Chapter 2, “Mixing Languages”.

WORD SIZE DIFFERENCES

Green Hills compilers are available on machines with 32-bit and 64-bit word size. Other C compilers have been written for machines with other word sizes. Porting C programs between machines of different word sizes requires particular care because most primary data types can be effected by word size.

RANGE OF REPRESENTABLE VALUES

The size of each basic numeric type controls the range of values which may be represented by that type. The header files **limits.h** and **float.h** provide defined symbols which represent the minimum and maximum values for all numeric

data types in C. A portable program should use these symbols and never depend on the use of values outside the allowed range.

If arithmetic operations cause overflow, underflow, or loss of precision, the program may not detect the error or may behave differently on different systems.

RELATIVE SIZES OF DATA TYPES

C places very weak requirements on the relative size of the basic types, but it is not unusual for C programs to assume otherwise. For example, C only requires that short be no larger than **int** and that int **be** no larger than long. It would be legal for **short**, **int**, and **long** to all be the same size or for them all to be different. With all 32-bit Green Hills C compilers **short** is 16 bits and **int** and **long** are 32 bits. To assume **int** is twice as large as **short**, but the same size as **long** is non-portable. With 64-bit Green Hills C compilers, **short** is 16 bits, and **int** and **long** are either 32 or 64 bits. You can only be assured that **long** is not smaller than **int**.

Another common non-portable assumption is that pointers are the same size as **int** or **long**. Neither is guaranteed. With all 32-bit Green Hills C compilers, pointers are 32 bits. But with 64-bit Green Hills C compilers, pointers may be either 32 or 64 bits, independent of the size of **int** or **long**.

In C, all integer constants have type **int** unless marked with a type suffix. In certain cases the use of a plain integer constant instead of a long integer constant can be non-portable.

BYTE ORDER PROBLEMS

Since the success of the IBM/360, byte machines have been more popular than word machines. The advantage of byte machines is their efficient processing of character data. The general acceptance of byte machines has led to easier program portability between machines.

There is, however, one major portability problem between byte machines. The first successful byte machine, the IBM/360, placed the most significant byte of a multiple byte integer value at the lowest address. Many byte machines such as 68000, RS/6000, SH, and SPARC have followed the IBM convention. The second successful byte machine, the PDP-11, placed the least significant byte of

a multiple byte integer value at the lowest address. Intellectual descendants of the PDP-11, such as the VAX, and i386/i486/Pentium and some RISC processors, such as the V800, have followed the DEC convention. These two groups seem to be so well entrenched that no agreement on byte ordering is possible. A further complication arises because some processors, such as the i960, M88000, PowerPC, and MIPS R3000/R4000, support both byte orders, although a given system is normally built to use only one byte order.

Between machines with different byte ordering, programs which overlay characters and integers in memory or which use character pointers to integer variables and vice versa are often not portable.

Programs that declare a single variable with different integer types in different modules may fail when ported to a machine with a different byte order.

ALIGNMENT REQUIREMENTS

Some systems will not load or store a 2 byte object unless that object is on an even address. Other systems have a similar requirement for 4 or 8 byte objects. Others may allow certain accesses, but require more time to perform them. Therefore, alignment of data is both a matter of correctness and time efficiency. Although increased alignment may improve performance, it also consumes space, due to padding inserted to achieve alignment.

The alignment requirements on each system are chosen both to satisfy the restrictions of the hardware and to achieve a reasonable balance between performance and space. The alignment rules for each system differ and often are not configurable. Therefore programs that make assumptions about the relative position of data objects in memory or elements within structures or arrays are not portable, even among the Green Hills C compilers.

The C language imposes these restrictions on size and alignment:

- The alignment of a struct, union, or array is equal to the maximum alignment requirement of any of its members.
- The size of a struct, union, or array is always a multiple of the maximum alignment requirement of any of its members.
- The offset of any member of a struct, union, or array is always a multiple of its alignment requirement.

- All dynamic memory allocation routines provided with the compiler will return a pointer aligned to the maximum alignment for any object on that machine.

All Green Hills C compilers also satisfy these principles:

- The stack is maintained on an alignment suitable for any object.
- Parameters and local variables are allocated on the stack according to their alignment requirement.
- Local variables are arranged on the stack to avoid unnecessary padding due to alignment.

If a program does not use integer arithmetic for pointer computations and ensures that all general purpose memory allocation routines return maximally aligned pointers, then all references to dynamically allocated memory will be properly aligned.

STRUCTURES, UNIONS, AND BIT FIELDS

The preceding issues of size, byte order, and alignment all effect the allocation of data in memory. In particular, compound data structures such as structures, unions, bit fields and arrays are very much effected by them.

UNIONS

A union in C allows the same memory location to be accessed as more than one type. This is inherently non-portable. Suppose a union consists of an integer and an array of four characters. Whether the first element of the array is the most significant part of the integer or the least depends on byte order. It is not even certain that the integer and the array of character have the same size.

These problems increase when integer, floating point and pointer fields are combined and are even more severe when structures or bit fields are members of unions.

STRUCTURES

The C language guarantees that fields in a structure are allocated in the order declared.

The exact offset of each field from the base of the structure depends on the size and alignment of the field itself and of those which precede it. The offset of the first field is always 0, but padding is inserted as necessary to satisfy the alignment requirement of each subsequent field, and may also be added at the end of the structure to make its overall size a multiple of its alignment.

Any program which assumes the offset of a field within a structure or which assumes that certain fields in two different structures always have the same offset are non-portable.

BIT FIELDS

The allocation of bit fields in a structure is very dependent on alignment rules. In addition, the exact layout of bits within a bit field varies between systems and cannot be assumed by a portable program.

ASSUMPTIONS ABOUT FUNCTION CALLING CONVENTIONS

Early implementations of C used a very straightforward approach to function calls. All parameters were pushed on the stack from right to left. All integral types smaller than int were promoted to int and float was promoted to double. Given this implementation, it was possible to write functions in C which handled variable parameters, even before the varargs and stdarg facilities. But such functions are non-portable, depending on an intimate knowledge of the calling conventions.

Many modern C compilers pass some parameters in registers and may not evaluate parameters from right to left. Integer and floating point variables, not to mention structures, may have different rules. One non-portable assumption is that a double may be passed to a function which expects two integers. Not only does this assume a relationship between the size of the two types and a certain ordering of bytes and words, but it assumes doubles and integers follow all of the same rules.

A much more common assumption is that pointer and integers may be interchanged when passing parameters. C does not guarantee that a pointer may be assigned to an integer and back without loss of information, even if the two are the same size. The only safe way to write a function which can correctly accept either pointer or integer parameters is to use the **varargs** or **stdarg** facility.

Even among integral types, a program may assume that int and long are interchangeable. C programs written in such an environment may invoke **printf** using the **%d** operator to refer to a long parameter. When this program is ported to a system where long is larger than int it will fail. The correct way is to use **%ld** for long parameters.

The same portability problems exist with respect to function return values. A function which returns an int should never be used to return a pointer or long or floating point value, even though it may work reliably on a particular system. A common mistake here is to omit a declaration of a function that returns a pointer, and then place a cast around the invocation of that function. The cast cannot fix the error, it only prevents the compiler from reporting it.

POINTER ISSUES

Nearly all machines supported by Green Hills compilers are byte addressable, but this is not a requirement of C. On some machines, a pointer to an int and a pointer to a char are not interchangeable. ANSI C requires that void pointers handle all pointer types, but the void pointer must be cast or assigned to its original type before being used. Similarly C does not require that function pointers and data pointers be interchangeable, but some C programs incorrectly make this assumption.

C supports portable pointer arithmetic, provided it is used correctly. In ANSI C, the difference of 2 pointers is only defined for cases where both pointers refer to 2 elements within the same array object. Even so, in unusual cases the difference may be outside the range of **ptrdiff_t** (which is either **int** or **long**). Subtraction or comparison of pointers to two separate objects may give non-portable results due to differences in memory layout or because pointers signed on one system and unsigned on another.

Pointer arithmetic should always be done directly on the pointers, not by casting or assigning the pointers to integer types.

NULL POINTER

In all Green Hills C compilers, and most C compilers in general, the NULL pointer has the value 0. But there are still two portability issues. Some older programs depended on the contents of memory location 0 being 0. This is now a

well recognized programming error and some modern machine purposely give a memory fault for any attempt to read or write to location 0.

A much more subtle problem is the size of NULL. On a machine where pointers are larger than int, it is incorrect to use the constant 0 as a NULL pointer, because 0 is of type **int**, which is smaller than a pointer. This matters when passing NULL to a function which takes variable parameters or which is not declared with a prototype.

CHARACTER SET DEPENDENCIES

Not all computer systems use the same characters. All computer systems recognize letters, digits, and the standard punctuation characters. But there is considerable variation among the less commonly used characters. Therefore, programs which use the less common characters may not be portable.

Your Green Hills compiler uses the ASCII character set and the ASCII collating sequence. Some language implementations use a different collating sequence, such as EBCDIC.

Programs which manipulate character data, especially string sorting algorithms, may be dependent on a particular character collating sequence. The collating sequence is the order in which characters are defined by the implementation. If one character appears before a second character in the collating sequence, then the first character will be ‘‘less than” the second character when they are compared. In the ASCII collating sequence, the lowercase letters “a” to “z” appear as the contiguous integer values 97 to 122 (decimal). In other collating sequences, such as EBCDIC, the lowercase letters are not contiguous.

To make character and string sorting programs portable, care must be taken to avoid dependencies on the character collating sequence. If a program is designed to operate with a collating sequence other than ASCII, it may be necessary to modify string and character comparison code to operate with ASCII.

FLOATING POINT RANGE AND ACCURACY

One of the most variable aspects of different machines is floating point arithmetic, where the range, precision, accuracy and base can vary widely. This can lead to many portability problems which can only be addressed

numerically. Your Green Hills compiler uses IEEE floating point representation.

OPERATING SYSTEM DEPENDENCIES

Programs which access operating system resources, such as files, by their system names are often not portable. The file and I/O device naming conventions vary greatly among computer systems. In order to write portable programs it is necessary to minimize the use of explicit file names in the program. It is best if these names can be input to the program when the program is run.

If a program contains explicit file names it may be necessary to change them to names acceptable to the target system. Refer to your target operating system documentation for a description of legal file names for your environment.

ASSEMBLY LANGUAGE INTERFACES

Programs which use embedded assembly code or interface to external assembly will require all of the assembly code to be redone when the program is transported to a new machine.

EVALUATION ORDER

None of the language specifications fully specify the order in which the various components of an expression or statement must be evaluated, and they disallow computations whose results depend on which permitted evaluation order is used. Many illegal programs have gone undetected because they have only been compiled with one compiler. Since your Green Hills compiler's evaluation order may not be identical to the evaluation order of other compilers, some of these illegal programs which operate as expected with another compiler may not operate the same way when compiled with your Green Hills compiler.

Some language implementations may evaluate the arguments to a function from right to left, others from left to right.

Expressions with side effects, such as subroutine, procedure, or function calls, may be executed in a different order by your Green Hills compiler and other compilers. When a variable is modified as a side effect of an expression and its value is also used at another point in the expression, it is not defined whether

the value used at either point in the expression is the value before or after modification. Different values for the same variable could potentially be used at different places in the expression depending on the order the compiler chose for evaluation.

The operators ++, --, +=, etc., may be executed in a different order by your Green Hills compiler and other compilers.

Your Green Hills compiler may allocate some pointer variables not declared **register** to registers. This may allow the compiler to generate more efficient sequences for post increment operators than other compilers. These sequences may involve incrementing at a different position in the statement than with other compilers. In particular, statements of the form:

*p++ = *expression involving p*

often evaluate differently under PCC than they do with a Green Hills compiler.

A particular case of evaluation order dependency is the use of the **?:** operator in an expression which is an argument to a function call. Your Green Hills compiler evaluates all question mark operators before any other arguments, and keeps the result in temporary storage. PCC evaluates the **?:** operator at its position in the argument list. The call:

```
foo(b?i:i+i, i++)
```

will usually evaluate differently under PCC than under your Green Hills compiler.

MACHINE SPECIFIC ARITHMETIC

Certain arithmetic operators in C are intended to generate the most efficient corresponding operation on the target machine. If all input values are within the expected range, the results are portable, but out of range values may give different results on different systems.

SHIFT

The shift operators in C have this characteristic. If the right-hand operand is negative or exceeds the number of bits in the left-hard operand the behavior is undefined. In Green Hills C compilers, the operands will be given to the

hardware as if the operands were legal and the result depends entirely on the hardware. Some systems accept a negative shift and reverse the direction of the shift, but many do not. Shifting by more than the number of bits is the same as shifting by 1 less than the number of bits on some systems, but on others it has very different results.

If the left-hand operand of a right shift is signed, C does not require the compiler to propagate the sign bit. That means a correct C compiler is allowed to yield a positive number when right shifting a negative number by one.

DIVISION

The division operator may round up or down when applied to signed integers if one or both of them is negative. Division by 0 produces different results on different machines.

The remainder operator always satisfies the rule

```
( a / b ) * b + a % b == a
```

as long as **b** is not 0. Therefore if **a** or **b** is negative, the sign of the remainder may or may not match the sign of the dividend, depending on the machine.

ILLEGAL ASSUMPTIONS ABOUT COMPILER OPTIMIZATIONS

Some programs illegally depend on the exact code that some particular compiler generates. Such programs are particularly difficult to port to an advanced optimizing compiler, such as your Green Hills compiler, because the optimizer makes major changes in the code in order to make the program smaller and/or faster. Described below are some of the most common illegal assumptions made about code generation. Please familiarize yourself with the optimizations described in Chapter 4, “Optimization”, before reading further.

IMPLIED REGISTER USAGE

Some programs rely on the exact register allocation scheme used by the compiler. Such programs are completely illegal, and will never transport without modification.

For example, C programs that rely on register variables being allocated sequentially to pass hidden parameters will not work. Hidden returns (i.e. using

return and expecting to return the value of the last evaluated expression) will not work either.

MEMORY ALLOCATION ASSUMPTIONS

Memory is allocated by your Green Hills compiler in a different way than by the industry's standard compilers and other companies' compilers. This can cause problems in porting programs which illegally depend on the memory allocation peculiarities of other compilers:

- Some programs depend on the compiler allocating variables in memory in the order that they are declared. Your Green Hills compiler will not necessarily allocate variables in the order of declaration.
- Some programs depend on knowing that the compiler will allocate all variables even if they are not used. Your Green Hills compiler may not allocate unused variables.
- Some programs depend on knowing that certain variables will be allocated in memory. Your Green Hills compiler will allocate certain variables to registers that the standard compilers other compilers would always allocate to memory.

Programs compiled with your Green Hills compiler must not make assumptions regarding the order or allocation of variables in memory (except where the language standard specifies it).

MEMORY OPTIMIZATION RESTRICTIONS

READ THIS SECTION CAREFULLY IF YOU ARE PORTING SYSTEM CODE OR APPLICATIONS THAT USE SHARED MEMORY OR SIGNALS.

Using the command line option **-OM** will enable the compiler to assume that memory locations do not change asynchronously with respect to the running program. In particular, when the compiler reads or writes some memory location, it will assume that the same value is still there several instructions later. To avoid the (potentially high) speed penalties involved in re-reading memory, the compiler will attempt to find a copy of the value which is itself still in a register, and use that instead.

This can easily cause problems for many parts of operating systems, device drivers, memory mapped I/O locations, shared memory environments, multiple process environments, interrupt driven routines, and when UNIX style signals are enabled. In C, general optimizations may be used as described in the next section.

MEMORY OPTIMIZATION IN C

An example of potential problems with memory optimizations is that many UNIX device drivers need to use memory locations which are really I/O registers that can change at any time. A typical example of a loop waiting for a device register to change is:

```
while (!(*TSRADDR & (1 << TXSBIT)));
```

If memory optimizations are enabled while compiling this loop, the compiler may generate code that reads the value pointed to by **TSRADDR** only once. With **-OLM**, it is almost certain that this will be the case. When this happens, the loop will execute either once or forever, depending on the value of the bit when it is first tested, and the loop will be rendered either ineffective or fatal.

Depending on the situation, the compiler may be able to detect loops like the above, and generate code that operates correctly even with **-OM** set. However, if the loop body were to test more than one bit at the same address, the compiler will contort the loop in an attempt to read memory as few times as possible.

The compiler assumes that you will use the **volatile** type qualifier when it is available. This means that **-O** implies **-OM** whenever the ANSI modes of compilation are used in C. If, for some reason, you are unable to use **volatile**, and this is a real problem, you can add the option **-Onomemory** to your command line to force memory optimization off. Note that **-Onomemory** also implies **-O**.

PROBLEMS WITH SOURCE LEVEL DEBUGGERS

This section describes various problems relating to source level debuggers.

VARIABLE ALLOCATION

Once a variable is allocated to a register it will always reside in that register. However, since other variables may share the register, it may not always

contain the current value of the variable. This may cause a source level debugger to give incorrect results. If you ask for the value of a variable at a point outside the range of its use, the compiler may have temporarily allocated that register for some other purpose. Always check results just after they are assigned, or when the current value is going to be used later. Near the end of a function most of the local variables will no longer be in use, so it is more likely that the register has been reallocated.

ADVANCED OPTIMIZATIONS

In general, Green Hills recommends that all optimizations be turned off if source level debugging is to be performed. The following are examples of specific problems that can be caused when optimizations are used in conjunction with source level debuggers.

- The common subexpression elimination optimization causes the compiler to try to precalculate expressions which are used more than once and save the result in a register. During debugging, the programmer will not find the expression itself, since it was evaluated and saved at an earlier time.
- Various loop and branch optimizations rearrange entire statements or blocks of statements causing difficulties with source level debugging since there will no longer be a direct correlation between source lines and executable instructions.

PROBLEMS WITH COMPILER MEMORY SIZE

Your Green Hills compiler is an advanced optimizing compiler. It is much better than the current generation of “optimizing” microprocessor compilers. In accordance with its greater capability, it requires more memory. The compiler requires 1 megabyte of memory just for the program. It is designed to work best when 2 megabytes or more of memory are available. It will run in less memory but with some degradation of performance or capability.

The compiler’s primary use of memory is for the program, static data structures, global declarations, parse trees, and generated machine code. Global declarations consist of the global constant, type, variable, and function declarations. Memory usage increases when large numbers of declarations are included in a compilation. Even unused global declarations must be stored throughout the compilation. If memory size problems exist, try to reduce the size of the include files by including just the declarations that are needed.

Memory is also needed for basic blocks. Every possible branch creates a new block. Machine generated programs with very large **switch** statements or a very large number of small **if** statements may increase memory usage.

Your Green Hills compiler is a one pass compiler. That is, it reads the source program only once. Each function is converted into a parse tree as it is read. When the end of the function is reached, the optimizer is called with the parse tree as input. The optimizer modifies the parse tree and then passes it on to the code generator. The code generator produces an internal representation of the machine code to be output for the function. Another optimization phase is then called to modify this machine code. Finally the optimized machine code for the function is output. After the machine code is output, the memory being used for the parse tree and machine code is released for use in compiling the next function.

The maximum memory usage for parse trees and machine code is determined by the size of the largest function in the program. If memory size problems exist, turn off the optimizer and reduce the size of the largest function. A simple function of less than 100 lines should not cause memory size problems. However, procedures which are more than 1000 lines, or contain very complex statements, can require several megabytes of memory to compile.

PCC MODE INCOMPATIBILITIES

The C preprocessor that is provided with PCC has many undocumented features. Most of these undocumented features are implemented in your Green Hills C compiler in PCC mode.

One little known feature of the C Preprocessor allows the results of two macro expansions to be concatenated into a single token. For example:

```
#define x /
#define y *
x/**/y A comment */
int va1;
```

This program is preprocessed by PCC into the following legal program before being compiled:

```
/* A comment */
int va1;
```

Due to the one pass nature of your Green Hills compiler it is not possible for its built-in preprocessor to manufacture a token such as **/***. In order to compile a program with such constructs, it is necessary to run the compiler in two passes. First, compile the program with the **-E** compile time option to produce the preprocessed source, then compile the preprocessed source as you normally would.

However, as a special case, the compiler can construct an identifier as:

```
#define O 2
int va2;
main() {
va/**/O = 1;
}
```

which becomes:

```
int va2;
main() {
va2 = 1;
}
```

DETECTION OF PORTABILITY PROBLEMS

Many of the problems associated with porting programs to your Green Hills C compiler from other compilers can be detected with the UNIX utility program **lint**(1). You should look for variables used before definition, routines using **return** and **return(*x*)**, nonportable character operations, evaluation order undefined, and routines whose value is used but not set. **lint** is not able to detect code that relies on the allocation order of memory variables, or that rely upon the arithmetic characteristics of short data types. Furthermore, since **lint** does not do actual data flow analysis, the absence of a message does not imply the absence of a problem.

Chapter

4

OPTIMIZATION

Along with providing standard optimizations available with other compilers, the Green Hills compiler supports an advanced set of optimizations. Among these optimizations are specialized suboptions which allow you to target specific types and areas of code for improved performance.

This chapter describes the Green Hills compiler optimizations under three categories:

- Optimizations performed by default
- General optimizations enabled with the **-O** option
- Specialized optimizations enabled with the suboptions **-OLAMISD**

DEFAULT OPTIMIZATIONS

This section describes the optimizations that the compiler performs by default, when no options are set:

- Constant Folding
- Register Allocation by Coloring
- Register Coalescing
- Loop Rotation

CONSTANT FOLDING

Constant folding optimization is performed when the compiler can determine at compile-time that an expression is a constant. The compiler substitutes the constant for any reference to the constant expression.

In these examples, the constant expression INT_MAX/2 has a value of 16383.

Initial C source code:

```
#define INT_MAX 32767
short subr(){
    int x;
    x=INT_MAX/2;
    return(x); }
```

Optimized C source code:

```
short subr(){
    int x;
```

```
    x = 16383;
    return(x); }
```

REGISTER ALLOCATION BY COLORING

Register allocation by coloring is used to permanently maintain a selected set of local scalar variables in registers based on their frequency of reference and their lifetimes. During program compilation, the optimizer uses data flow analysis to determine the lifetime of each variable. The register allocator also uses this information to assign different variables within a function to the same register if the lifetimes of the variables do not overlap. This increases the opportunity for allocating variables to registers.

With the local variables preallocated to registers, the compiler can optimize the code significantly, since additional memory load and store instructions are not required to reference the variables.

In these examples, the variables **a** and **b** are both assigned to the same register since their lifetimes do not overlap (note that the code could be optimized still further, but is left as is to simplify the examples).

EXAMPLES:

Initial C source code:

```
int subr(x)
int x;
{
    int a,b;
    a=x;
    b=x*2;
    return b;
}
```

Optimized C source code:

```
int subr(x)
int x;
{
    int a;
    a=x;
```

```
    a=x*2;
    return a;
}
```

For small functions, the compiler maintains all local variables in registers. Scalars generally are considered for register allocation unless their values are accessed with the address operator (**&**). This optimization is disabled with the **-nooverload** option.

REGISTER COALESCING

With register coalescing optimization, the optimizer uses the destination register as a work register when evaluating the associated expression and organizes the instruction sequence so the result ends up in the destination register. This optimization eliminates the additional register-to-register copies required when using a temporary register.

Initial C source code:

```
int fun(a,b,c)
int a,b,c;
{
    int ret = a+b+c;
    return ret;
}
```

Optimized C source code:

```
int fun(a,b,c)
int a,b,c;
{
    return a+b+c;
}
```

LOOP ROTATION

Loop rotation optimization refers to locating the termination test and a conditional branch at the bottom of the loop. Therefore, the loop only processes one branch instruction on each iteration. Most compilers place the termination test and an unconditional branch at the top of the loop and an additional unconditional branch at the bottom.

EXAMPLES:

Initial C source code:

```
int subr(i)
int i;
{
    while (i < 10)
            i *= i;
    return(i);
}
```

Optimized C source code:

```
int subr(i)
int i;
{
    goto L7;
    do {
            i *= i;
    L7:
    } while (i < 10);
    return(i);
}
```

In addition, if the compiler can determine that the loop is executed at least one time, the loop is entered at the top. If not, the compiler generates an unconditional branch at the top of the loop to the termination test.

GENERAL OPTIMIZATIONS ENABLED WITH THE -O OPTION

General optimizations are enabled with the **-O** option. When **-O** is selected, all of the following optimizations are performed:

- Common Subexpression Elimination
- Peephole Optimization
- Pipeline Instruction Scheduling
- Static Address Elimination
- Dead Code Elimination
- Constant Propagation

- Tail Recursion

Certain **-O** optimizations can be controlled with **-Ono** options, each of which disables a specific **-O** optimization but enables all others. For example, the **-Onocse** option enables all **-O** optimizations except for common subexpression elimination. These options are described in the appropriate optimization sections.

STATIC ADDRESS ELIMINATION

With static address elimination optimization, the optimizer assigns frequently used static variables to registers within the scope of the function. This optimization eliminates the loads and stores required with memory allocation. It is enabled with the **-OM** option.

In these examples, the address of the static variable **x** is maintained in register.

EXAMPLES

Initial C source code:

```
int subr(q)
int q;
{
    static int x=0;
    x++;
    q+=x;
    return(q);
}
```

Optimized C source code:

```
int subr(q)
int q;
{
    static int x=0;
    register int x_ = x;
    x_++;
    q+=x_;
    x=x_;
    return(q);
```

```
}
```

Note that this optimization is performed not only for locally defined static variables, but also for global variables, as shown in the following example:

Initial C source code:

```
int x = 0;

int subr(q)
int q;
{
    x++;
    q+=x;
    return q;
}
```

Optimized C source code:

```
int x=0;

int subr(q)
int q;
{
    register int x_ = x;
    x_++;
    q+=x_;
    x=x_;
    return(q);
}
```

PEEPHOLE OPTIMIZATION

Peephole optimization identifies common code patterns and replaces this code with more efficient code patterns. This includes optimizations such as removal of unreachable code, flow of control and algebraic simplifications. The compiler only performs this optimization when local code analysis insures that the results will be correct without further analysis of the surrounding code. This optimization is disabled with the **-Onopeep** option.

Initial C source code:

```
int subr(x,y,z)
int x,y,z;
{
    y = x;
    z = y;
    return z;
}
```

Optimized C source code:

```
int subr(x,y,z)
int x,y,z;
{
    return x;
}
```

COMMON SUBEXPRESSION ELIMINATION

Common subexpression elimination is performed when a previously calculated expression is part of a later expression and none of the variable values in the subexpression have changed. The optimizer retains the value of the subexpression in a register for reuse. This optimization is disabled with the **-Onocse** option.

Initial C source code:

```
int subr(x,y)
int x,y;
{
    int a, b;
    x += a+b;
    y += a+b;
    if (y < 0)
    return(y);
    return(x);
}
```

Optimized C source code:

```
int subr(x,y)
```

```
{
    int a, b, _v6;
    x+=(_v6=a+b);
    y+=_v6;
    if (y<0)
    return y;
    return x;
}
```

TAIL RECURSION

A procedure is considered tail recursive if the last statement executed is a procedure call to itself followed by a return statement. This is sometimes simply called a recursive procedure. Tail recursion optimization replaces the procedure call with a branch instruction and eliminates the return statement.

Initial C source code:

```
int sum(n)
int n;
{
    if (n <= 1)
             return(1);
    else
    return(n+ sum(n-1));
}
```

Optimized C source code:

```
int sum(n)
int n;
{
    int _v3=0;
L1:
    if (n <= 1)
             return _v3+1;
    _v3 += n;
    _n--;
    goto L1;
}
```

DEAD CODE ELIMINATION

With dead code elimination, the optimizer does not generate assembly code for statements computing values that are never used and therefore have no effect on the program results.

In this example, the optimizer eliminates all code for processing the variable **a** since it knows at compile-time that the variable **a** is zero and therefore any code referencing it is not used.

Initial C source code:

```
#define F0 0
#define F2 2
int subr(x)
int x;
{
    int a,b,c;
    a=F0*x;
    b=F2*x;
    return ((a)? a : b);
}
```

Optimized C source code:

```
int subr(x)
int x;
{
    int b;
    b=2*x;
    return(b);
}
```

CONSTANT PROPAGATION

Constant propagation is the replacement of one or more variables with constants over the course of a variable's lifetime if the variable's value is known and does not change during that lifetime. The following examples show code optimized with constant propagation:

C source code:

```
main()
{
    int i,a,b;
    a = 3;

    for (i=0;i<1000;i++)
    b += a;        /* a is constant over the lifetime
                   ; of the loop */
    printf("%d\n", b);
}
```

Optimized C source code:

```
main()
{
    int i,b;
    for (i=0; i<1000; i++)
              b+=3;
           printf(“%d\n”, b);
}
```

SPECIALIZED OPTIMIZATIONS SET WITH THE SUBOPTIONS -OLAMISD

The specialized optimizations are enabled using the **-OL**, **-OA**, **-OM**, **-OI**, **-OS,** or **-OD** options. These optimizations enable the general optimization along with the indicated suboptions. The optimizations provided by each option are as follows:

-OL	Loop Optimization: Strength Reduction Loop Invariant Removal Loop Unrolling
-OA	Algorithmic Optimization
-OM	Memory Optimization
-OI	Inlining Optimization
-OS	Size Optimization
-OD	Delete Optimization

You can combine these suboptions (**L**, **A**, **M**, **I S,** and **D**) in any order by appending them to the **-O** option. For example, the **-OLAMISD** option turns on all optimizations.

LOOP OPTIMIZATION WITH -OL

Loop optimization is selected with the **-OL** option. This option informs the compiler that most computation is performed within the innermost loops. Therefore, the compiler focuses most of the available machine resources on optimizing that portion of code.

The following loop optimizations are performed:

- Strength Reduction
- Loop Invariant Removal
- Loop Unrolling.

You can also list specific functions for this optimization using the following syntax:

-OL=*func1*,*func2*,...,*funcn*

The **-Onounroll** and **-Ounroll8** options can be used with **-OL** to affect loop unrolling. See section Register Caching Over Loops on page 96 for more information.

STRENGTH REDUCTION

Strength reduction optimization is applied to arrays subscripted with the loop index. Most compilers access the array element by multiplying the size of the element by the loop index. The Green Hills compilers store the address of the array in a register and add the size of the array element to the register on each iteration of the loop.

Initial C source code:

```
subr()
{
    int i;
    int q[4];
    for (i=0;i<4;i++)
            q[i]=i;
```

```
}
```

Optimized C source code:

```
subr()
{
    int i;
    int q[4];
    int *_ptr;
    for (i=0, _ptr=q; i<4; i++)
                      *_ptr++ = i;
}
```

Strength reduction also applies to multiplying a loop invariant with the loop index. The optimizer replaces a multiply instruction or a call to the **mul()** library function with add and shift instructions.

LOOP INVARIANT REMOVAL

Loop invariant removal enhances loop performance. Each loop is examined for expressions or address calculations that do not change within the loop. These computations are located outside the loop and their values are stored in registers.

This optimization is particularly valuable for reducing the code generated to access an element of an array when the array index does not change within the loop.

Initial C source code:

```
subr()
{
    int i,j;
    int q[4],p[4];
    for (i=3;i>=0;i--)
    q[i]=i;
    for (j=0;j<4;j++)
              p[j]=q[i];
}
```

Optimized C source code:

```
subr()
{
    int i,j;
    int q[4],p[4];
    int *_ptr;
    for (i=3; i>=0; i--)
              q[i] = i;
    for (j=0, _ptr = &q[i]; j<4; j++)
              p[j] = *_ptr;
}
```

REGISTER CACHING OVER LOOPS

With register caching over loops optimization, the compiler duplicates the code in the innermost loop up to a maximum of four times by default. This optimization produces more straightline code, which removes much of the loop overhead in testing for stop condition and branching. This allows better use of the register allocator and more opportunity for instruction pipelining. It is most effective when the innermost loop is relatively short causing minimal increase in code size.

There are two options that can be used along with **-OL** to affect loop unrolling. **-Ounroll8** allows loops to be unrolled up to 8 times instead of the default maximum of 4 times.'**-Onounroll** disables loop unrolling but enables the other **-OL** options.

The following simple examples use a constant loop size of 100 and a maximum loop index of four to show the effect of this optimization.

Initial C source code:

```
subr(a)
int a[];
{
    int i;
    for (i=0;i<100;i++)
              a[i]=i;
}
```

Optimized C source code:

```
subr(a)
int a[];
{
    int i;
    for (i=0;i<100;i+=4) {
    a[i]=i;
    a[i+1]=i+1;
    a[i+2]=i+2;
    a[i+3]=i+3;
    }
}
```

Calling the size of the loop **n**, suppose that **n** is large (auxiliary loop execution time is negligible); then, the original loop takes **n***(4 cycles per iteration) == 4**n** cycles to complete. The unrolled loop takes **n**/4*(10 cycles per iteration) == 2.5**n** cycles to complete. With **n** large, the unrolling has the effect of making the loop execute in only 63% of the time required by the original loop.

ALGORITHMIC OPTIMIZATION WITH -OA

These optimizations assume the program implements a portable algorithm which is not affected by the limitations of finite hardware. For example, these optimizations may apply algebraic properties such as associativity without respect to the possibility of overflow, underflow, round-of, loss of precision, or division by zero.

Furthermore, these optimizations assume that the algorithm never makes use of the characteristics of two's complement integer arithmetic or IEEE floating point arithmetic beyond that implementation independent rules of ANSI C. For example, ANSI C states that the size of an **int** is implementation defined and in most environments supported by Green HIlls compilers, an **int** is a 32-bit two's complement number. For example any program that depends on an **int** having exactly 32-bit bits, rather than 35 bits, or which depends on two's complement arithmetic rather than signed magnitude or some other representation should NOT be compiled with **-OA**.

For example,

```
unsigned char c = -1;
```

```
if (c == 255)
    foo();  /* with -OA this might not be called */
signed char s = -127;
if (c - 5 > 0)  /* note that c-5 yields 4 because of overflow */
    bar();  /* with -OA this might not be called */
```

In ANSI C, the include file **limits.h** provides implementation defined bounds for all integral types. Any code which depends on the result of an arithmetic operation which exceeds these bounds should not be compiled with **-OA**.

Some programs achieve portability by intentionally forcing overflow in order to determine the limitations of the hardware. The results of these tests are then used to avoid overflow in the rest of the program. These overflow tests should NOT be compiled with **-OA**.

ALGEBRAIC ALGORITHMIC OPTIMIZATION

With some systems there is an additional type of algorithmic optimization that can be enabled with the **-X915** option (note that **-OA** must also be specified for this to work). With this optimization, whenever the compiler finds a multiply across an add, such as **X*(Y+Z)**, where **X** is a constant, it will distribute the multiply across the add, so our previous example would become: **X*Y+X*Z**. Even though this actually increases the number of calculations performed (from two to three) it can actually increase the speed of the calculation due to better register usage on some systems.

MEMORY OPTIMIZATION WITH -OM

Memory optimization is enabled with the **-OM** option. This allows the compiler to optimize repeated memory reads by placing the value in a register. Subsequent read operations then refer to the register rather than the actual memory location. With this optimization the compiler assumes that memory locations only change with explicit store instructions and therefore are not affected by any external sources.

It is therefore not recommended for applications in which memory could by externally affected: device drivers, operating systems, and shared memory. This also applies in a non-virtual memory environment when interrupts are enabled.

The **-OM** option is automatically set with the **-O** option in full ANSI or 90% ANSI mode (the **-ANSI** or **-ansi** options), since the **volatile** keyword is defined

to explicitly identify objects that may change without the compiler's knowledge or control. If you wish to want to use **-O** without using **-OM** in one of these modes, you may use the **-Onomemory** option. This option turns on **-O**, but turns off memory optimization.

SPACE OPTIMIZATION WITH -OS

Space optimization is enabled with the **-OS** option. This tells the compiler to perform all default and general optimizations that would increase efficiency but not greatly increase code size. For instance, if you compiled your code with the optimization option **-OSL**, the compiler would omit the loop unrolling phase.

INLINING WITH -OI

The term "inlining" refers to the process of substituting the contents of a function or subroutine in place of the call to that function or subroutine. The resulting code is faster, since the overhead of a jump-to-subroutine call has been eliminated. Typically, a small function or subroutine that is frequently executed, but is called from only a few locations within the program, is the best candidate for inlining. This way, the maximum benefit can be obtained by increasing efficiency in high usage areas, while not significantly increasing program size. Note that this feature is not currently supported with C++. See your Release Notes for more information.

The following program illustrates the basic principles of inlining. The main program in this case contains a simple loop which calls the function **sub()**. The call itself occurs only once in the program code, but the function is executed for each iteration of the loop. The call is easily replaced by the routine code for **sub** itself, eliminating both the need for parameter passing and the overhead of a jump-to-subroutine. The reduced overhead per execution becomes a major savings in program speed.

Initial C source code:

```
_ _inline sub(x) {
    printf("x=%d\n",x);
    return;
}
main() {
    int i;
```

```
    for (i=1;i<10;i++)
            sub(i);
}
```

Optimized C source code:

```
sub(x) {
 printf("x=%d\n",x);
 return;
}
main() {
 int i;
 for (i=1;i<10;i++)
 printf("x=%d\n",i);
}
```

Note that the code for **sub** has not been eliminated, although the main program no longer contains a call to **sub**. The compiler generates code for each function, whether or not it is inlined, so that it will be available to be called from other modules and so that its address can be taken. While the size of the actual generated code was not changed significantly, the execution speed of the main program was improved by eliminating the jump-to-subroutine overhead.

USING THE INLINER

The Green Hills implementation of inlining is language independent within the Green Hills family of compilers. Routines of one language may be freely inlined into programs of another language. Also, inlining is performed across modules: if a function **falcon()** to be inlined is defined in one module but used in several, the compiler will be able to inline **falcon()** in all the modules in which it is used.

For the sake of brevity, the word “function” in the following sections on inlining is used to apply to FORTRAN subroutines as well.

SELECTING FUNCTIONS TO BE INLINED

There are three methods for selecting the functions to be inlined:

Manual Inlining

The **_ _inline** keyword may be inserted in the source code immediately before the declaration of each function to be inlined. This is referred to as manual inlining. Manual inlining is always active even if no other optimizations or inlining methods have been enabled.

Automatic Inlining

With automatic inlining, the compiler determines which functions will be inlined. Automatic inlining is selected with the command line option **-OI**.

Command Line Inlining

Command line inlining allows the user to specify the names of certain functions to be inlined on the command line. This resembles manual inlining in that the user determines whether or not each function will be inlined. Command line inlining is selected with the command line option **-OI=***name1*,*name2*.

SINGLE-PASS AND TWO-PASS INLINING

Whenever a function is used in only one file, and is defined in that file before it is used, and is manually marked for inlining with **_ _inline**, the function will be inlined during the normal course of compilation. This is referred to as single-pass inlining.

In order to inline a function which is not declared before it is used, or which is called from a file other than the file in which it is declared, two-pass inlining is required.

The command line options **-OI** and **-OI=** always enable two-pass inlining in addition to determining the criteria for selecting the functions to be inlined. Therefore, it may be necessary to specify the **-OI** or **-OI=** option to enable two-pass inlining, even if every function is manually marked for inlining.

USING THE COMMAND LINE OPTIONS

-OI The **-OI** option indicates that automatic inlining should be performed and that manual inlining should be performed in two passes. The compiler will automatically select functions to be inlined. In addition, each function which is manually marked with **_ _inline** will be inlined, including those which are used before

they are declared in a file and those which are used in files in which they are not declared. For example,

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

will cause the compiler to be invoked twice for each of the three source modules. First, each of the source files will be processed to produce an inline file with a **.inf** extension. Then each source file will be compiled again to produce an object file. On the second pass, both the original source file and the three **.inf** inline files will be used as input.

-OI=*names* The **-OI=***name* option also indicates that command line inlining should be performed and that manual inlining should be performed in two passes. A list of names of functions to be inlined may be specified after the **-OI=** option, separated by commas. In addition, each function which is manually marked for inlining with **_ _inline** will be inlined, including those which are used before they are declared in a file and those which are used in files in which they are not declared. If automatic inlining is also to be performed the **-OI** option must be used as well.
The command line

```
% gcc -OI=sub,func main.c prog1.c prog2.c
```

will cause the functions **sub()** and **func()** to be inlined wherever they are encountered, along with each function which is manually marked for inlining with **_ _inline**.

-OI= The **-OI=** option without any arguments indicates that only manual inlining should be performed in two passes.

TWO-PASS INLINING IMPLEMENTATION

When two-pass inlining is enabled, the compiler driver invokes the compiler inliner once for each source module, creating an inline file for each module. All of the functions in a single source file which are candidates for inlining are stored in the corresponding inline file. The name of the inline file is formed by taking the source filename and replacing the suffix with a **.inf** suffix.

Next, the compiler is invoked a second time for each source file. The original source file along with all previously created **.inf** inline files will be used as input. An object file will be generated for each source file, exactly as it would if no inlining had been performed. The difference will be that certain calls to functions will have been replaced with inline copies of those routines. (Functions which are inlined will also have code generated for them, ensuring full compatibility with conventional programming techniques.) Finally, all of the object files will be linked normally.

INLINING OPTIMIZATION ENHANCEMENTS

Inlining is traditionally considered an optimization which increases program size for the sake of improving program speed. Program size is increased because a single function is generated in each place where it is called. Program speed is improved because the branch-to-subroutine call is eliminated. In fact, there are many ways in which inlining serves to reduce program size as well as improve program speed. When a call is replaced by inlined code, the compiler can usually avoid saving and restoring several registers before and after the call. Parameters which normally must be passed on the stack to a called routine can be accessed directly by the inlined routine in their original location.

Furthermore, because Green Hills compilers perform inlining before most global optimizations, the process of inlining can significantly enhance the opportunities for additional optimizations resulting in very efficient code.

For example, if one or more parameter values are constant, large portions of the inlined routine may be reduced or eliminated at compile-time and loops which normally execute a variable number of times may become constant.

Register allocation may improve because the overhead associated with a call is eliminated. On most architectures, when a call to a routine exists within a routine, the number of registers available for local variables and temporaries is reduced. If all routine calls can be eliminated by inlining, the number of registers available for variables and temporaries will be increased.

Pessimistic assumptions made by the compiler when compiling the caller may not be necessary if no call is made. Normally the compiler must assume that global variables may be changed when a call is performed. This prevents the compiler from optimizing the values of expressions which contain global

variables across a call to a function. When the function is inlined, the call is eliminated and the global variables may be optimized freely.

INLINING LIMITATIONS

The inlining optimization is subject to the following limitations:

- Source line number information related to inlined routines is deleted. When executing a program under control of a source debugger, no source code will be available for the inlined routine. Single stepping by source line will cause the entire inlined call to be executed as a single statement. However, you can debug the inlined call by stepping through the sequence of inlined machine instructions at the point of the source-level call.
- Functions containing **asm** statements cannot be inlined.
- Routines written in assembly language cannot be inlined because they are simply assembled to produce an object file. They cannot be processed by the compiler inliner.

SELECTING OPTIMIZATIONS

This section provides a demonstration on using the UNIX system profiling utility to take full advantage of the specialized optimizations available with the Green Hills Compiler to improve the performance of your application.

The information that is generated by the profiler is commonly used to identify time-critical or inefficient code. This data is also very useful to select the appropriate optimizations for your particular application and specifically to identify functions for inlining and loop optimizations.

The system profiler produces a profile of your application which contains statistics relative to each function. Using the **-p** compiler option results in an executable containing calls to the system routine “monitor”. When your executable is run, these calls keep track of each function's performance. This raw data is written to a file called **mon.out**. The profile utility, **prof**, interprets the data in **mon.out** and generates a formatted report. The following list shows the categories of information in the report and what each category means.

%time	percentage of total run-time spent within a function
cumsecs	cumulative seconds spent for processing a function
#call	number of times a function is called

ms/call	time in milliseconds per function call
name	function name

When your code is linked, the compiler driver uses special profiled libraries to generate your executable.

Chapter

A

KANJI CHARACTER SUPPORT

.

ABOUT KANJI

Kanji is the Japanese name for one of the written forms of Japanese. Almost all Kanji characters are taken from Chinese, although a few are specific to Japanese.

Both Chinese and Kanji use one or more characters to represent a word. In Chinese, each character is monosyllabic; in Japanese a Kanji character can be monosyllabic, multisyllabic, or both, depending on the character. (Almost all Kanji characters have multiple pronunciations.). There are over 30,000 characters in Chinese, but Kanji routinely uses only a portion of them. For example, one can read a Japanese newspaper knowing less than 2,000 Kanji characters. The computer representations for Kanji provide between 5,000 and 10,000 Kanji characters.

In Japanese, there is also a phonetic alphabet with less than 100 characters. Just like English, this alphabet has cursive and block forms. They are called Hiragana (cursive) and Katakana (block). Today Katakana characters are used to spell foreign words and proper names phonetically.

GREEN HILLS SUPPORT FOR KANJI

Green Hills C, C++, and FORTRAN compilers (and MULTI) provide support for Kanji characters in comments, character strings, and character constants. Their use in variable names or other identifiers is not supported because of the conflicts it would cause with syntactic rules requiring identifiers to begin with a letter or underscore. Some Japanese vendors provide a Kanji preprocessor which allows Kanji characters to be used in identifiers in C and FORTRAN.

Kanji fonts are available with X, allowing MULTI to support Kanji, and many companies manufacture PC's and Workstations with support for Kanji in both the keyboard and display.

Green Hills run-time libraries also correctly process character data containing Kanji characters. There are several different ways to represent Kanji characters. The Kanji representation directly supported by Green Hills compilers uses a pair of characters to represent a single Kanji character. The first character is always in the range of 0xa0 to 0xfe and the second character is always in the range of 0x80 to 0xfe.

The exact representation of Kanji is usually irrelevant to the compiler and libraries, as long as neither byte conflicts with special values such as **\0, \n, \r, ", \',** etc.

The C compiler allows a 2-byte Kanji character with single quotes. This is easily done, as C allows two ASCII characters between single quotes.

WIDE-CHARACTER VS. MULTI-BYTE REPRESENTATION

In ANSI C, a new data type called 'wide characters' was invented to handle Kanji and other languages more systematically. A wide character is a character which is represented in more bits than a typical character. There are wide character strings also.

The difference between an old fashioned character string containing Kanji and a wide character string containing Kanji is that the former is an array of bytes and the latter is an array of shorts or longs. For this reason, the old fashioned representation is called multi-byte. The old way used 2 units for Kanji, while the new way uses bigger units.

The third character in a wide character string is always the third element, but in a multi-byte character string it could be one element or two. What makes the old way even more difficult is that the character could begin after the 2nd, 3rd, or 4th byte, depending upon whether there were Kanji or ASCII characters before it in the string.

There are library routines to convert between wide character and multi-byte character forms of characters and character strings. The Green Hills library recognizes multi-byte Kanji character data and correctly converts it to and from the wide character representation. This is done in the C locale because there is not yet an accepted convention for naming Kanji-specific locales.

.

Index

A

Ada
- address types 62
- array types 62
- C main() program for Ada 44
- pointers types 62
- string types 62

address types 62
algebraic algorithmic optimization 93, 97, 98
alignment
- requirements 70
- type double 16

alternate returns 50, 57
-ANSI 2
-ansi 2
ANSI C
- library 36
- limitations 26
- mode 24
- Permissive mode 3
- predefined symbols 15
- required symbols 15
- standard P-2
- strict 4

argument passing
- C and Ada 61
- C and FORTRAN 46
- FORTRAN and C 53

arithmetic, machine specific 76
array
- loop invariant removal 95
- types 62

asm
- C language support 3
- Permissive ANSI support 4
- statements 33, 104

assembly language interfaces 75
automatic inlining 101

B

bit fields
- alignment and size 26
- C language 25
- Permissive ANSI support 4
- signed 26
- structure allocation 72
- unsigned 26

branch instruction 91
Bsd 4.x 18
built-in functions
- void__DI 35
- void__EI 36
- void_set_il 36

byte order 16, 69

C

C
- memory optimization 79
- restrictions on size and alignment 70

C++
- header files 63
- in C programs 64

#call 104
calling conventions 72
calling languages
- C from Ada 60
- C from FORTRAN 45
- FORTRAN from C 53

cerr global object 65
character set dependencies 74
CHARACTER type 48, 55
cin global object 65
C language
- dialects 2
- library 36
- modes 2

clog global object 65
C main() program for Pascal 43
code efficiency 26
command line inlining 101
COMMON blocks
- information 52
- naming conventions 53, 59

common subexpression elimination 87, 90
compiler
- license 36
- memory size problems 80
- optimization 77

compile-time checking 6
COMPLEX*16 type 56
COMPLEX*8 type 56
COMPLEX type 49, 56

conditional branch 86
const 3, 24, 25
constant
    expressions 4
    folding 84
    propagation 87, 92
cout global object 65
C preprocessor output file 15
C routines 63
cumsecs 104

D

data general DG/UX 18
_ _DATE_ _ 15
dead code elimination 87, 92
debugger problems
    advanced optimizations 80
    variable allocation 79
#defined() 3
delete optimization 93
destination register 86
directives 3
division operator 77
%d operator 73
DOUBLE COMPLEX type 49, 56

E

-E 15, 82
#elif 3
embedded 17
empty structure 4
#endif 4
enum 3
enumerated types 30
#error 3
error 4, 25
evaluation order 75
examples
    C main() program for C++ 42
    C main() program for FORTRAN 43

F

_ _FILE_ _ 15
floating point format 16
floating point range 74
FORTRAN
    CHARACTER type 48, 55
    COMMON 52, 59
    COMPLEX*16 type 56
    COMPLEX*8 type 56
    COMPLEX type 49, 56
    DOUBLE COMPLEX type 49, 56
    VMS compatibility mode 52, 59
function
    calling conventions 72
    prototype in C and C++ 65
    prototypes 3

G

general optimizations 87
general symbols 16
    _ _BigEndian 16
    _ _Char_Is_Signed 17
    _ _Char_Is_Unsigned 17
    _ _ghs_ _ 16
    _ _ghs_alignment 16
    _ _ghs_packing 17
    _ _ghs_pic 17
    _ _ghs_pid 17
    _ _ghs_sda 17
    _ _ghs_tda 17
    _ _ghs_zda 17
    _ _IeeeFloat 16
    _ _Int_Is_32 17
    _ _Int_Is_64 17
    _ _LANGUAGE_C_ _ 16
    _ _LittleEndian 16
    _ _LL_Is_64 17
    _ _Long_Is_32 17
    _ _Long_Is_64 17
    _ _NoFloat 16
    _ _PROTOTYPES_ _ 16
    _ _Ptr_Is_32 17
    _ _Ptr_Is_64 17
    _ _Ptr_Is_Signed 17
    _ _Ptr_Is_Unsigned 17
    _ _Reg_Is_32 17
    _ _Reg_Is_64 17
    _ _SoftwareDouble 16
    _ _SoftwareFloat 16
    _ _Wchar_Is_Int_ _ 17
    _ _Wchar_Is_Long_ _ 17

_ _Wchar_Is_Short_ _ 17
_ _Wchar_Is_Signed 17
_ _Wchar_Is_Unsigned 17
_ _gh_initrec() 42
Green Hills library 45

H

Harrbison and Steele P-2
header files in C++ 63

I

I/O on single file in multiple languages 44
#ident 3
inefficient code,identifying with profiler 104
_ _inline 101
inlining
identifying functions 104
information 99
using the inliner 100
inlining methods
automatic 101
command line 101
manual 101
single-pass 101
two-pass 101, 102
inlining optimization 93, 103
instruction pipelining 96
interfacing Pascal and C 62
interrupt
routines 24
iostream library
cerr 65
cin 65
clog 65
cout 65
ISO C Standard P-2

J

Japanese Automotive C 34
-japanese_automotive_c 35

K

K+R
C features 3
information 15, 24
mode 2
-k+r 2
Kanji A-2
Green Hills support A-2
multi-byte representation A-3
wide-character representation A-3
Kernighan and Ritchie P-2

L

-language 41
language mode options 2
%ld operator 73
libansi 36
libind 7
library 36
Green Hills 45
initialization 42
native UNIX 45
_ _LINE_ _ 15
local scalar variables 85
loop
invariant analysis 93, 95
optimization 93, 104
rotation 86
unrolling 93, 94, 96

M

machine specific arithmetic 76
manual inlining 101
memory
alignment 25
allocation 78
size problems 80
memory-mapped I/O 24
memory optimization
in C 79
information 93, 98
restrictions 78
volatie type qualifier 24
Microsoft Windows 3.1 18
mixed language executable 40
monitor 104
mon.out 104

ms/call 105
multi-byte representation A-3

N

name 105
named constant value 25
naming conventions, Pascal 62
native UNIX library 45
-noasm 33
-no_asm_warn 35
-noasmwarn 34
-nooverload 86
-no_short_enum 35
NULL pointer
    dereferences 7
    information 73
numeric constants 3

O

-O 87, 98
-OA 93, 97
-OD 93
-OI 93, 99, 101
-OI= 102
-OL 93, 94
old-fashioned
    assignment 28
    initialization 28
    syntax 28
-OM 24, 78, 88, 93, 98
-Ono 88
-Onocse 88, 90
-Onomemory 79, 99
-Onopeep 89
-Onounroll 94, 96
operating system dependencies 75
operating system symbols
    _ _bsd 18
    _ _DGUX_ _ 18
    _ _msw 18
    _ _sco 18
    _ _sun 18
    _ _sysV 18
    _ _sysV4 18
    _ _sysV4pic 18
    _ _unix_ _ 18
    _ _VXWORKS 18
    _ _windows 18
operators
    division 77
    shift 76
optimization
    advanced 80
    algorithmic 98
    default 84
    loop invariant analysis 95
    loop unrolling 96
    strength reduction 94
order evaluation 75
-OS 93, 99
-OSL 99
-Ounroll8 94, 96

P

-P 15
-p 104
padding bytes 25
PCC (Portable C Compiler) 2
    mode incompatibilities 81
peephole optimization 87, 89
performance improvement 104
Permissive ANSI
    information 15, 24
    mode 3
pipeline instruction scheduling 87
pointer
    checking 7
    information 17
    issues 73
    NULL 73
    types 62
portability problems 82
position independent code 17, 18
position independent data 17
pragma
    asm 5
    can_instantiate 14
    do_not_instantiate 14
    hdrstop 14
    ident 11
    inline 11
    instantiate 14
    intvect 35

no_pch 14
pack 12
weak 12
-pragma_asm_inline 35
pragma endasm 5
pragma ghs
check 6
enddata 6
ifnodebug 8
ifnooptimize 8
includeonce 8
interrupt 8
intvect 12
nofloat interrupt 9
revertoptions 9
sda 9
section sect 9
startdata 6
zda 9
pragma ghs ifdebug 8
pragma ghs ifoptimize 8
pragma interface C 60
predefined symbols 3, 15
predefined symbols, target specific 18
i960 23
Intel x86 18
MIPS 20
Motorola 19
SH 22
SPARC 22
V800 22
preprocessor 3
prof 104
profiler 104
prototypes
for library functions 65
function in C and C++ 65
information 16
transition mode 3

R

READ 63
recursive 91
references
Harrbison and Steel, C, A Reference Manual P-2
ISO C Standard P-2
Kernighan and Ritchie, The C Programming Language P-2
registers
allocation by coloring 85
allocator 96
caching 96
coalescing 86
usage 77
requirements for alignment 70
restrictions on size and alignment 70
return types
simple 47, 55
routine
calling C from Ada 60
calling C from FORTRAN 45
calling FORTRAN from C 53
C language 63
run-time
checking 6
library 36

S

-s 62, 63
scalar variables 85, 86
SCO UNIX 18
semicolons 4
shift operator 76
-shortenum 30
signed 3
signed bit fields 26
-signedfield 26
single-pass inlining 101
size optimization 93
small data area optimization 17
Solaris 2.x 18
space optimization 99
standard optimizations 24
static address elimination 87, 88
static variables 88
stdarg 31, 32
stdarg.h 3
_ _STDC_ _ 15
straightline code 96
strength reduction 93, 94
Strict ANSI 15

mode 4
string types 62
structure
assignment 25
parameters 25
structures 25, 29, 71
SunOS 4.x 18
symbol naming 51, 58
syntax, old-fashioned 28

T

tail recursion 88, 91
termination test 86
_ _TIME_ _ 15
%time 104
tiny data area optimization 17
Toyota Motor Corporation 34
Transition mode 3, 15, 24
two-pass inlining 101
type
char 17, 31
enum 30
enumerated 30
float 31
int 17, 30
long 17
short 31
wchar_t 17
type qualifiers
const 25
volatile 24

U

unions 25, 29, 71
UNIX
information 18
library 45
System V.3 18
System V.4 18
unsigned bit fields 26
-unsigned_char 35
-unsigned_field 35
-unsignedfield 26

V

varargs 31
varargs.h 3
variable
allocation 79
arguments 31
limitations 31
void 3
void _ _DI 35
void _ _EI 36
void_set_il 36
volatile 3, 24

W

warning 4, 34
wide-character representation A-3
Wind River VxWorks 18
word size 68
WRITE 63

X

-X174 63
-X402 52, 59
-X59 62
-X608 52, 59
-X915 98
-Xa 2
-Xc 2
-Xnooldfashioned 28
-Xs 2
-Xt 2

Z

-Z608 52, 59
zero data area optimization 17