Data General

Customer Documentation

Green Hills Software C-88000™ User's Guide

Green Hills Software C-88000™ User's Guide

069-100230-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Green Hills Software, Inc.

510 Castillo St.	230 Second Ave.
Santa Barbara, Ca. 93101	Waltham, Ma. 02154
(805) 965-6044	(617) 890-7889
Fax: 965-6343	Fax: 890-4644

Table of Contents

CHAPTER 1

Overview

1.1. The Green Hills C Documentation Set

The Green Hills C standard compiler documentation set includes a User's Guide and Language Reference Manual. Additional documentation on product installation and execution is provided separately. You may need to refer to separate documentation describing the assembler, librarian and linker for your target system, and also the operating system and hardware architecture.

1.2. User's Guide Structure

The Green Hills C User's Guide is system specific, and describes compile time options, porting and optimization, and considerations for the target operating environment.

Overview

> The Overview describes the structure of the documentation for the compiler.

Language Features

> This section describes the main features of Green Hills C Version 1.8.5, language enhancements/extensions and compatibility.

Target

> The Target chapter describes the target processor and operating system environment in which your program will operate. It describes calling conventions, register allocation and memory allocation strategies. It describes restrictions imposed on the compiler by the target system. It also tells how to modify the output of the compiler to be compatible with different target environments.

Optimization

> The Optimization chapter gives detailed information about the optimizations used by Green Hills C to improve program performance. It also explains how to get the best performance out of your program.

Porting Programs to C

> This chapter tells you about difficulties that you may encounter in moving a program developed with another compiler to Green Hills C. It gives specific examples of difficulties that may be encountered and how to resolve them.

CHAPTER 2

The C Language

2.1. Introduction

C-88000 is a complete implementation of the C programming language, and supports three separate modes; ANSI, FullANSI and PCC. These terms are used throughout this document whenever a particular construct is supported only in a specific mode. If no specific mode is mentioned, the construct is supported in all three modes.

FullANSI mode (-ANSI or -X153) provides 100 percent compliance with the ANSI C X3J11 standard and disallows any non-compliant constructs.

ANSI mode (-ansi or -X316) provides 90 percent compliance with the ANSI C X3J11 standard, allowing certain useful, but non-compliant, constructs to be supported while providing an ANSI C framework.

PCC mode (the default) is provided for compatibility with PCC, the Portable C Compiler. PCC is the most widely used implementation of C. It is the compiler that is used to implement and maintain UNIX, the largest and most important body of C code. Therefore, Green Hills has chosen to use PCC, and in particular the Berkeley 4.2BSD VAX implementation of PCC, as the default definition of the C language.

C-88000 in PCC mode contains everything in the basic C language, as well as all of the documented Western Electric extensions, and all of the undocumented features of the Berkeley compiler used in implementing UNIX. There are hundreds of extensions to the basic C language which are implemented in all versions of PCC. Without these extensions it is impossible to compile UNIX and many existing C applications programs. Several of the most important of these extensions are listed in the section on PCC compatibility, but this is by no means a complete list.

Currently, the user documentation for C-88000 consists of this document, along with Kernighan and Ritchie. When ANSI or FullANSI modes are used, the ANSI publication X3J11/88-083 should be used as additional documentation. Unix implementations will also need the documentation provided with UNIX (the Western Electric extensions).

A complete and concise C Reference Manual for the Green Hills family of C compilers is currently under development and will be available in mid-1990. This document will replace the supplementary documentation currently required in addition to the C-88000 User's Guide.

unions that will be compared, it is important to either have no holes in the memory representation or for each such variable to be explicitly initialized 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, so passing a very large structure or union is not recommended.

2.2. PCC Compatibility

2.2.1. Old Fashioned Constructs

The default behavior of the compiler in PCC mode is to allow old fashioned initialization (such as int x 5) and old style reflexive operators (such as x =+ 5), but generate the messages:

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

The -X84 option causes the compiler to disallow these constructs in PCC mode (in ANSI and FulIANSI modes these are always illegal).

The following table shows examples of the old and new style syntax which is affected by -X84.

	Old			**New**		
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	=|	0	x	|=	0
	x	=^	1	x	^=	1
	x	=%	z	x	%=	z
	x	=<<	3	x	<<=	3
	x	=>>	5	x	>>=	5

2.2.2. Enumerated Types

Enum identifiers are signed integers by default for compatibility with PCC. The compile time option -X6 allows them to be allocated to the smallest predefined type which allows representation of all listed values, including unsigned integer types.

2.2.3. The VARARGS Facility

C-88000 supports the UNIX VARARGS facility. The VARARGS facility allows a function to access its parameters in left to right order even if the number and/or types of the parameters are not known until run time. To use the VARARGS facility:

2.2.5. Extern and Common

In PCC, the default storage class for a variable declared in the outer scope is ''common''. That is, the variable will be allocated separately from this module. It will be allocated with the same initial address as all other variables of storage class ''common'' with the same name declared in the outer scope of other modules. The size of the variable allocated will be the size of the largest of the ''common'' variables of that name. In PCC, the storage class ''extern'' defines a variable to be a reference to the ''common'' variable of that name. If there is an ''extern'' declaration for a name there must be at least one ''common'' declaration of that name in the program. There may be many ''extern'' and ''common'' declarations of the same name. The PCC model for ''extern'' and ''common'' is supported by all UNIX versions of C-88000.

In some target environments ''common'' is not implemented, or it is implemented very poorly. In those cases a different interpretation is made for the default storage class. If a variable is declared ''extern'' in one module there must be exactly one declaration of a variable of the same name and type with the default storage class in exactly one module in the same program. There may be many ''extern'' declarations for the variable. This interpretation for the default storage class seems to fit the definition in Kernighan and Richie better than the PCC definition.

If the second method is followed, a program can be ported to any implementation of C. The first method is more convenient when using include files. It is the only method used in UNIX. Most UNIX programs cannot be ported unchanged to target environments that do not support ''common''.

2.2.6. asm Statement

The asm statement (for inline assembly code) in C-88000 is the same as in PCC. In C-88000 the asm statement can be used anywhere a statement can appear. If -X436 is specified, the **asm** statement can be used anywhere a declaration can appear, even between functions.

Since the code generated by C-88000 is substantially different than the code generated by other compilers it is usually necessary to modify most asm statements.

The predefined identifier _ _asm is available in all modes, the predefined identifier asm is not available in FullANSI mode.

The asm statement is not supported in compilers which generate object code directly.

_ _STDC_ _ is a predefined preprocessor symbol available only in FullANSI mode.
It is a decimal constant with the value of 1 indicating full conformance
to the ANSI XJ311 standard.

2.3.5. Trigraph Sequences

A set of nine alternate representations for graphic characters not supported on all terminals is provided as part of the ANSI standard support in 1.8.5. These alternate representations all begin with the two character prefix "??" and are called trigraph sequences. They are recognized and replaced by their ASCII counterparts during the initial translation phase of the compiler. Trigraph sequences are recognized and converted only in ANSI and FullANSI modes.

2.3.6. Type Qualifiers

There are two new type qualifiers in ANSI and FullANSI modes, **const** and **volatile**, which may be specified no more than once in a specifier or qualifier list.

2.3.6.1. volatile

When optimizations are turned on with -O in ANSI or FullANSI modes, -OM is turned on automatically. -OM means that the compiler may assume that memory locations only change under the control of the compiler, (ie. not true for memory which is updated by interrupt routines, or for memory-mapped io, for example). Since the compiler is allowed to make this assumption (which almost always true), it may avoid and/or delay reads or writes to memory locations by maintaining a copy of the memory location in register(s). The volatile qualifier specifically turns off the -OM optimization for the indicated variables, allowing all non-volatile variables to benefit from the -OM improvements.

2.3.6.2. const

This qualifier provides the compiler with additional information for use in optimizations. Wherever the value of a const variable is visible, the optimizer make 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.

CHAPTER 3

C Library

On UNIX systems, C-88000 can use the standard C library. For users wishing to use the new ANSI C libraries, and for non-UNIX systems which do not already have a C library, Green Hills supplies the Green Hills C library. This library is supplied as either object code or C source code, depending on the environment.

To use the Green Hills C Library you need a standard C-88000 compiler license. Under this license, unlimited distribution of programs which are linked with Green Hills C Library object code is permitted without charge. However distribution of the Green Hills C Library source code or object code is not permitted.

CHAPTER 4

Motorola 88000 Target

4.1. Introduction

This chapter describes the Motorola 88000 target environment for C-88000.

4.2. Motorola 88000 Characteristics

The Motorola 88000 memory is byte addressed with 32 bit addresses. Bits are numbered with bit zero as the most significant bit.

By default, bytes are ordered with the most significant byte of a multiple byte value stored at the lowest address, the Big-Endian byte order, as on the IBM/370 and MC68000 (opposite of the VAX and 8086). The reverse byte-order, Little-Endian, can be achieved by using the -Z78 compile time option. For the purposes of this document, Big-Endian byte ordering is assumed.

Floating point is IEEE format (32 and 64 bits), most significant byte at the lowest address.

Character encoding is ASCII.

The stack is always eight byte aligned.

Bit fields are allocated starting at the most significant bit. Every bit field is fully contained in four or fewer bytes. Each struct, union, and array is aligned to the maximum alignment requirement of any of its components.

Data Type	**Size**	**Alignment**
int	32	32
long	32	32
*	32	32
short	16	16
char	8	8
float	32	32
double	64	64
unsigned	32	32
unsigned char	8	8
unsigned short	16	16
enum (default)	32	32

Although arguments are evaluated from right to left, they are assigned stack offsets from left to right. The first argument is always at offset 0. The size of the first argument is rounded up to a multiple of 4 bytes and added to its offset to determine the offset of the second argument. If the second argument requires 8 byte alignment and its offset would not otherwise be a multiple of 8 bytes, its offset is increased by 4 bytes. This is repeated until offsets have been assigned to all arguments. The size of the entire argument area is then increased by 4 bytes if necessary so that it will also be a multiple of 8 bytes. A space of at least 32 bytes and large enough to hold this argument area is present on the stack immediately before the call.

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

Arguments with offset 0 through 28 may be placed in registers r2 through r9, respectively, if they have both 4 byte size and 4 byte alignment or if they have both 8 byte size and 8 byte alignment. Eight byte arguments are placed in the two consecutive registers which correspond to their offset.

A call to a function uses a bsr or jsr instruction which saves the return address in r1. The return from a function uses a ''jmp r1'' instruction.

Return values that are scalar, pointer, 32-bit floating point or 4 byte aligned 4 bytes sized structures or unions are returned in r2, sign or zero extended to 32 bits for types smaller than 32 bits. 64-bit floating point values are returned in the register pair r2/r3.

To call a function which returns a structure or union (unless it is 4 byte aligned and 4 byte sized), the address of a temporary of the return type is passed in r12. The function returns the structure value by copying the return value to the address pointed to by r12 before returning to the caller.

A function call is assumed to destroy r1 to r13. No other registers are destroyed by a function.

Accesses to parameters or local stack storage are always made relative to the stack pointer. A frame pointer is only set up if source level debugging is required.

CHAPTER 5

Optimization

5.1. Introduction

C-88000 does many optimizations which are not available in other C compilers. These optimizations can reduce the size of a program by up to 30% and increase its speed by up to four times. C-88000 performs all of the optimizations performed by most other C compilers. It folds constant expressions, converts multiplications into shifts and divides into multiplications when it is advantageous, and eliminates redundant jumps and unreachable code.

5.2. General Optimizations

General optimizations always make programs smaller and faster.

5.2.1. Register Allocation by Coloring

Register allocation by coloring is used to keep the most commonly used values in registers at all times. The entire function is examined to determine which local variables and parameters are used most frequently. The most commonly used variables and parameters are allocated to machine registers. No memory is allocated for them. This optimization has a significant savings in execution speed and it saves a great deal of space. Referencing a variable in a register usually takes one-third of the space and one-third of the time of referencing a variable in memory.

The register allocator uses data flow analysis to find the lifetime of each variable. Using this information, it increases the number of variables which are stored in registers by using the same register for several variables in the same function. Two variables may be allocated to the same register if there is no place in the program in which both variables hold a value that will be used later on. Most of the time, all local variables are kept in registers and none in memory.

By default, any integer, pointer, enum, float, or double automatic (or register) variable is a candidate for allocation to a register, unless its address is taken with the ''&'' operator.

By default, all register candidates will be allocated to the available registers so as to give either the fastest or densest code possible (as controlled by the -OL compile time option). Most C compilers will allocate one register variable to each available register and then allocate all other register variables and all automatic variables in the stack frame. C-88000 will allocate as many of the register variables to registers as it can. Then it will allocate any automatic variables to registers if it can. C-88000 is much better than most C compilers in its register allocation.

In the following example, C-88000 allocates i and j to the same register because their lifetimes do not overlap.

5.2.3. Entry and Exit Code Optimization

Most compilers use a frame pointer register in each function. The frame pointer is used to access local variables, to point up the call stack to allow stack traces to be printed during debugging, and to unwind the stack for an exception mechanism. The frame pointer is valuable but it is usually not necessary. By default, C-88000 does not set up a frame pointer in each function. C-88000 will generate a frame pointer if the code is the same size or smaller with a frame pointer, but otherwise it will not create a frame pointer and it will access all local variables by using the stack pointer instead.

If it is necessary to have a frame pointer in every function the ''-ga'' compile time option can be specified on the command line. This compile time option will guarantee that there will always be a frame pointer, but it will increase the size of the program.

If a function is very short (a common occurrence in structured programming), the entry and exit code may take a large fraction of the space and execution time of the function. If all of the parameters and local variables of a function are allocated in registers (usually for a function of 20 lines or less), the compiler can often eliminate the subroutine entry and exit code entirely. This optimization generates code much like the best assembly language implementation.

See the example under Register Allocation by Coloring for improvements to the entry and exit code.

5.2.4. Static Address Elimination

A valuable optimization performed by C-88000 is to maintain frequently used static addresses in registers. Since static addresses are 4 bytes long, if a static address is used just twice in a function, it is faster and smaller to load the address into a register just once at the beginning of the function and always use ''register indirect'' addressing to access it. In this way, most static references are reduced to one-third of the space and less execution time.

```
    p()
    {
        f(1);
        f(2);
        f(3);
        f(4);
    }
```

```
_p:
        subu    r31,r31,8
        st      r1,r31,4
        bsr.n   _f
        or      r2,r0,1
        bsr.n   _f
        or      r2,r0,2
        bsr.n   _f
        or      r2,r0,3
        bsr.n   _f
        or      r2,r0,4
        ld      r1,r31,4
```

5.2.7. Loop Rotation

In C, the ''for'' and ''while'' statements specify the loop termination conditions at the top of the loop. Therefore, many C compilers generate a termination test at the top of the loop and an unconditional branch from the bottom of the loop to the top of the loop. The loop will execute two branch instructions on each iteration of the loop.

A better way to generate code for loops is to place the test at the bottom of the loop. This is called ''Loop Rotation''. If it can be determined at compile time that the loop will always execute at least once then the loop is entered from the top. If it cannot be determined that the loop will be executed at least once, then an unconditional branch to the termination test is placed before the loop entry. With the test at the bottom only one branch is executed on each iteration of the loop.

5.2.8. Peephole Optimizations

Peephole optimizations are local improvements to the code which are certain to be correct without further analysis of the surrounding code. An example would be two machine instructions where the first moves the contents of register A to register B, and the second instruction moves the contents of register B to register A. If the program code never branches to the second instruction (i.e. both instructions are always executed together), the second instruction can be safely eliminated.

All of the peephole optimizations which have been implemented are safe for device driver code. Should there be any reason to suppress these optimizations, it can be done with the -X9 compile time option.

5.3. Loop Optimizations

Programs which execute for long periods of time execute millions or billions of instructions. Since most programs consist of tens or hundreds of thousands of instructions, some instructions must be executed many times. To increase the speed of a program it is necessary to identify which instructions are executed the most often and concentrate the optimizations in these areas. Computer languages have two main constructs for repeating the execution of instructions: loops and subroutines. By making specific optimizations for each of these constructs it is possible to significantly improve the performance of most programs.

The loop optimizer is selected by the -OL compile time option. This compile time options informs C-88000 that most computation is performed in inner loops. When this compile time option is specified, C-88000 assigns most of the machines resources, registers in particular, to uses in the innermost loop. This can result in significant performance increases in programs which do most of their computation in loops.

The loop optimizer draws resources away from other useful optimizations. If -OL is specified for a program in which very little computation is done in inner loops, most of the machine's resources will be misdirected in attempting to optimize infrequently executed loops. This can result in decreasing the total performance of the program. The -OL compile time option should only be used on modules for which the programmer is certain most processing occurs in loops.

5.3.1. Loop Invariant Analysis

''Loop Invariant Analysis'' is used to speed up loops. Each loop is examined for expressions and address calculations which do not change in the loop. These computations are moved out of the loop and the value is stored into a register. This optimization is particularly valuable for removing array subscripts from a loop when the subscript is a variable or expression which is not modified in the loop. In a small loop, all invariant expressions will be accessed with ''register mode'' and all invariant addresses will be accessed with ''register indirect modes.'' This optimization usually eliminates all computations of invariant expressions and addresses in loops.

CHAPTER 6

Porting Programs to C-88000

Some programs which appear to compile and operate correctly when compiled with other C compilers, may not operate correctly when compiled with C-88000. The C Language specifications define programs in such a way that portable programs will always work with all C compilers, including C-88000. The problem is that many programmers make non-portable assumptions about the machine or compiler that they are using. This chapter discusses many non-portable assumptions which can cause programs to fail when compiled with C-88000.

6.1. Compatibility with other Green Hills Compilers

All Green Hills Languages use the same calling conventions for all subroutines, routines, procedures, and functions. Therefore, code from other Green Hills Languages can be freely used within your C-88000 program.

The implementation of each Green Hills C Compiler is the same for each Green Hills Target. Therefore, legal programs written in C-88000 can be moved to any other Green Hills C Compiler.

C-88000 can be obtained on any Green Hills Host. It is exactly the same on every Host. Therefore, program development can be done on more than one Host, and moving your development to a new Host system is easy.

6.2. Word Size Problems

Some machines are byte addressable. That is, they have addresses which refer to 8 bit bytes. They typically have operations which operate on 8, 16, 32, 64 and 128 bit quantities. Other machines are word addressable. That is, they have addresses which refer to words of a standard size varying from 16 to 64 bits. They typically have operations which operate on multiples of the word size.

If two different machines have different word sizes or if one is word addressable and the other is byte addressable, a program which operates on one machine may not operate on the other machine for several reasons. The word size affects the range of numbers implemented by the ''int'' data type. The word size also affects the precision and range of the float and double data types.

The most common word size problems are (often undetected) integer overflows and floating point underflows, overflows, and loss of precision. The layout of bit aligned data structures will vary with the word size, so overlaying structures in memory (with union types or pointers) makes programs difficult to port to another compile. Doing address arithmetic in integer variables is often not portable. C provides portable pointer arithmetic if it is used correctly.

overlays. It will also lead to problems with programs which make implicit assumptions about the size and offset of objects.

6.5. 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.

C-88000 uses the ASCII character set and the ASCII collating sequence. Some implementations of C 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 lower-case letters ''a'' to ''z'' ear as the contiguous values 97 to 122. In other collating sequences the lower-case letters are not contiguous.

To make character and string sorting programs portable, care must be taken to avoid dependence 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.

6.6. Floating Point Range and Accuracy

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

6.7. 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 the names to names acceptable to the target system in order to get them to operate with C-88000. Refer to your target operating system documentation for a description of legal file names for your environment.

6.8. 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.

6.9. Evaluation Order

The C Language specification does not fully specify the order in which the various components of an expression or statement must be evaluated, but it disallows computations whose results depend on which permitted evaluation order is used. Many illegal programs have gone undetected for years because they have only been compiled with one compiler. Since the C-88000 evaluation order is not

```
    main()
    {
      va/**/O = 1;
    }
```

Which becomes (in both PCC and C-88000)

```
    main()
    {
      va1 = 1;
    }
```

6.11. 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 C-88000, 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 about code generation that some programs depend on to work. Please familiarize yourself with the optimizations described in the ''Optimization'' chapter before reading this section.

6.11.1. Problems with Setjmp and Longjmp

Under the default configuration of C-88000, an occasional problem surrounds the undocumented subtleties of the ''setjmp'' and ''longjmp'' functions in some UNIX programs. Setjmp is a function which saves the contents of the registers, the stack context, and the program counter into a ''label'' variable. The longjmp function restores the registers, stack content and program counter from the contents of the ''label'' variable and continues executing after the call to setjmp. Under PCC only variables specified ''register'' will be allocated to registers and, therefore, saved in the ''label'' variable, the other variables will remain on the stack. If a ''register'' variable is modified after the call to setjmp, a longjmp will restore the ''register'' variable to the value saved in the ''label'' variable, so the modification will be lost. However if a non-''register'' variable is modified after the call to setjmp, a longjmp will not affect the value of the variable and the modification will be retained. Some versions of some UNIX programs, as well as the Plum Hall Validation Suite, depend on whether a variable's value will be restored by longjmp. Since the Green Hills compiler may allocate automatic variables to registers and may allocate ''register'' variables in memory, it is not predictable as to whether any modifications to a variable which take place after a setjmp will be retained or lost after a call to longjmp on the same ''label'' variable.

The -X18 switch causes all programmer defined variables which are not declared ''register'' to be allocated in memory as in the portable C compiler. The -X18 switch generates worse code than the default configuration, but in the few cases in which the (undocumented) subtleties of setjmp and longjmp are depended upon, it will operate consistently with the portable C compiler. The compile time option -X125 allows the compiler to detect the presence of a call to setjmp and only enables -X18 in those routines.

6.11.5. Problems with Source Level Debuggers

6.11.5.1. 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, the register may not always contain the value of that variable. This may cause a source level debugger to give incorrect results. If you ask for the value of a variable at a point at which that variable is about to be assigned into, 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 are no longer going to be used, so the chance that the register has been reallocated is much higher.

6.11.5.2. Advanced Optimizations

In general, we recommend that all optimizations be turned off if source level debugging is to be performed. Here are some examples of specific problems that can be caused when optimizations are used in conjunction with source level debuggers.

CSE, or Common Subexpression Evaluation, causes the compiler to precalculate expressions and store the result in a register. During debugging, the programmer will not find the expression itself, since it was evaluated and substituted 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.

On 88000 targets, the -X307 option should be used to turn off pipeline scheduling since this optimization effects virtually all code, changing the order in which operations are performed. There is no accurate way to map a source line to a sequence of instructions after this type of optimization is performed.

6.12. Problems with Compiler Memory Size

C-88000 is an advanced optimizing compiler. It is much better that the current generation of ''optimizing'' microprocessor C compilers. In accordance with its greater capability it requires more memory. C-88000 requires 800 Kbytes just for the program. It is designed to work best when it has 2 Mbytes or more of memory 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. This is a major use of memory 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. Another use of memory is 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 use excessive memory.

CHAPTER 7

Compile Time Options

7.1. Normal Compile Time Options

The following compile time options are available for general use and allow the user to select common compilation options, such as ANSI mode, or to specify library directories and/or redirect output.

Most options are case sensitive and are recognized only as shown in the documentation. The Green Hills driver passes on any invalid option specifications, therefore the -v and/or -X255 options are often useful to verify which options were accepted and are in use by the compiler for the current compilation. Note that your compiler has been configured so that it can be used in its intended environment without needing to specify any special compile time options. These special options are discussed in a later section. In some cases, the ''english'' version of an option, such as -ANSI, will be translated by the driver into its -Xnnn equivalent (in this case, -X153). In all cases where this is true, the equivalent -Xnnn option will be documented as part of the option description. Familiarity with the -Xnnn equivalent is necessary when using the -v or -X255 verification options since the compiler will echo the -Xnnn form rather than the ''english'' equivalent that was specified on the command line.

More than one option may be specified. Options may appear anywhere on the compilation command line, and except where noted below may be specified in any order.

-ansi This option places the compiler in ANSI mode. ANSI mode is 90% compliant with the ANSI X3J11 standard, allowing certain useful, but non-compliant, constucts to be supported while providing an ANSI C framework. -X316 is equivalent to -ansi.

-ANSI This option places the compiler in FullANSI mode. FullANSI mode is 100% compliant with the ANSI X3J11 standard and disallows any non-standard constructs, generating error and/or warning messages. -X153 is equivalent to -ANSI.

-c (UNIX Host only) Do not produce executable files, produce only object files. For each source language file specified, compile the source language file into object code output. Put the object code output into a file whose name ends in ''.o''.

-C If this option is given, comments are output in the preprocessor output. The default is to strip comments from the output.

-Dname Define ''name'' to the preprocessor with the value 1. This is equivalent to putting ''#define name 1'' at the top of the source file.

-Dname=string
Define ''name'' to the preprocessor with the value ''string''. This is equivalent to putting ''#define name string'' at the top of the source file.

-E Do not compile the program, instead place the output of the preprocessor on the standard output file. This is useful for debugging preprocessor macros. The integrated preprocessor

-Uname	Undefine the predefined preprocessor symbol ''name''. This is equivalent to putting ''#undef name'' at the top of the source file. This option can be used to remove any symbols which are predefined in the compiler.
-v	(UNIX Host only) Have the compiler driver print out the program name and command line arguments as it runs each subprocess.
-w	Suppress warning diagnostics.

-X21	Map all identifiers to upper case, for assemblers which require this.
-X31	(Non-UNIX Host only) Allow arbitrary file names to be specified to the compiler.
-X32	Display the names of files as they are opened. Useful for finding out why the compiler cannot find an include file.
-X37	Emit a warning when dead code is eliminated.
-X39	Do not move frequently used procedure and data addresses to registers.
-X55	Make bit fields of type int, short, and char be signed. The default is for all bit fields to be unsigned.
-X58	Do not put an underscore in front of the names of global variables and procedures.
-X74	The target system is UNIX System V.
-X80	Turn off the branch tail merging optimization. This can speed up compilation in some cases.
-X81	Allow extern variables to be initialized (by turning off extern). This is an error in cc, and by default in C-88000.
-X84	Do not recognize C anachronisms. This may cause various syntax errors if old style constructs have been used. By default the old assignment operators (=+ =- ...), initialization (int i 1), and references to members of other structures compile correctly but generate warning messages.
-X85	Generate ''.lcomm'' (BSD) or ''.bss'' (UNIX System V) for zero initialized statics, rather than placing uninitialized local data in the initialized data section with initial value 0. This can result in significant reduction in the size of binary files. The -X85 option is normally on by default in a UNIX environment.
-X105	Allow redefinition of #define symbols to the preprocessor.
-X114	Target is UNIX BSD 4.2. This controls debug and binary output formats among other things.
-X115	Target is UNIX BSD 4.1. This controls debug and binary output formats among other things.
-X153	Place the compiler in FullANSI mode. This is set automatically when -ANSI is specified. If this option is used in conjunction with -O, -OM is automatically selected too.
-X159	Do not allocate 2 named variables to the same register. This makes debugging easier but may result in slower code.
-X164	Do not stop in the event of a code generator abort or ''Internal Compiler Error'' error message. Place a message in the assembly language output indicating where the error occurred. This option is occasionally useful for determining the cause of a compiler failure. If this option is used, the compiler may crash or otherwise terminate abnormally.
-X167	Unsupported option. Evaluate expressions involving only float operands as float (not double). Do not expand float arguments to double. Do not expand float return values to double.
-X168	Do not move invariant floating point expressions out of loops.
-X171	Do not create a static base register.
-X187	Suppress output from #ident.
-X188	Use Fortran mixed mode expression evaluation rules. In particular, do float*float computation in single precision, do not convert to double precision before performing operation.
-X202	Don't output "." before assembler directives.
-X206	Suppress errors for illegal preprocessor directives that are skipped because they occur within #if...#endif pairs that evaluate to false.

-X352	Don't extend float arguments to double in order to pass them to functions.
-X353	Perform common subexpression analysis twice. Rarely useful.
-X370	Output line numbers in the assembly file.
-X380	Parentheses behave as they are said to in (some versions of) the proposed ANSI C standard, that is the compiler may not associate over them.
-X394	Do not recognize various tautologies like (x-x)==0 and (3+a)+(4+b)==7+a+b during common subexpression evaluation and do not eliminate simple divides, such as those used by Dhrystone during common subexpression elimination.
-X401	Enable CEXTERNAL declaration.
-X403	Accept noalias keyword in C.
-X405	Type 'char' is unsigned type.
-X407	(Weitek only) Turn on IEEE machine independent software floating point.
-X414	Use 96-bit objects for long doubles.
-X415	Allow C++ style references.
-X416	Allow functional casts (ie. double(3)) and automatically typedef struct names as in C++.
-X421	Union of integer types allocated and passed in registers.
-X422	All types larger than 128 bits are forced to that alignment.
-X424	Replace unsigned division by constant with multiply by the pseudo-reciprocal and shift right.
-X428	Turn on CSE register caching.
-X429	Don't try to propagate common subexpressions through loops.
-X434	Put in code to check for nil pointer dereferences.
-X442	Do not perform special optimizations for constant multiplies.
-X447	Produce code to generate a runtime error if a switch (case, computed goto) statement has no default (otherwise) case and is entered with a value that is not one of the listed cases.
-X465	Mark errors in cpp listing file.
-X468	Suppress constant propagation optimization. See also -X230.
-X469	Don't recognize sin, cos, etc. as pure functions.
-X470	Suppress tail recursion.
-X474	Suppress Common Subexpression Elimination (CSE). See also -X230.
-X482	Disable loop unrolling. This option is intended for use with -OL to allow various loop optimizations to be performed without turning on loop unrolling.
-X483	Flush assembly output at end of each routine.
-X484	Pass small structs in registers.
-X485	Extra args for return values in rtmps.

-X490*filename*

Use specified filename string for .file directive. Normally the compiler takes the name of the source file and uses it to generate a .file directive of the form ''.file "filename.c" ''. However, when an external preprocessor is being used, the filename that the compiler sees is not the name of the original file, but of an intermediate file. By using this option it is possible to tell the compiler the name of the original source file. This option may be passed either to the driver or directly to the compiler.

CHAPTER 8

Runtime Error Messages

The following table lists error messages generated when the compiler inserts debugging checks into the output program. The switches that can cause each of these messages to appear are listed to the left of the resulting message.

-X57	Array index/variable assignment out of bounds
-X434	Nil pointer dereference
-X447	Case/switch index out of bounds
-X588	Unititialized variable/field referenced

CHAPTER 9

Compile Time Error Messages

The C compile time error messages are listed below in alphabetical order.

# operator must be followed by a macro parameter
operator may not begin or terminate a macro
#defines nested too deeply
#defines recursive or too complex
A braced initializer must contain a value
A cast is illegal in the expression of a #if/#elif
A constant may not be assigned a new value
A file (translation unit) must contain at least one declaration/definition
A function may not return an incomplete type
A function must be declared by specifying parens
A non-prototype parameter list may not be specified here
A type or a storage class must be specified in a declaration
A volatile in a register makes no sense (to me)
Array size exceeds implementation limit
Array size must be constant
Bit field not legal as operand of sizeof
Cannot take the address of this object
Case expression not constant
Case not in switch statement
Character constant too long
Const ignored
Constant expected
Could not disambiguate overloaded procedure name:
Declaration not legal here
Duplicate case
Empty character constant illegal
End of file found in #if, #ifdef, or #ifndef
End of file found in comment
End of line found in character constant
End of line found in string
Fatal error in reading library file:
File name too long
Function illegal in structure or union
Globalvalue initializer must be constant
Globalvalue must be int or unsigned
Illegal Type
Illegal character
Illegal floating constant
Illegal function
Illegal initial value

This object has no defined size
This use of an incomplete type is illegal
This warning message reserved for the inliner.
Too few arguments passed to function
Too many -I options
Too many arguments passed to function
Too many parameters for a macro
Two storage classes specified
Type mismatch
Type size exceeds implementation limit
Unexpected end of file
Unknown size for parameter
Unmatched #endif
Variable expected
Void type argument illegal
Void type for
warning: #ident directive not allowed in ANSI C.
warning: Ambiguous lvalue usage:
warning: Arithmetic constant too large for type/array index
warning: Cannot take the address of this object
warning: Illegal combination of pointer and integer
warning: Inline routine has complex inits of non-local variables, expansion suprressed.
warning: Macro arguments extend beyond invoking macro
warning: Name converted to string constant
warning: Nameless parameter in function definition
warning: Negative or zero array size
warning: Old fashioned type declaration, double assumed
warning: Old-fashioned initialization
warning: Preprocessing directives found inside of macro argument
warning: Shift amount too large or too small
warning: This compiler does not have any inlining capability.
warning: Undefined static function used-
warning: Unrecognized lower case letter after backslash inside string
warning: Variable read before written:
warning: Wrong number of params in macro call
warning: asm statement is not portable
warning: illegal macro name
warning: old-fashioned assignment operator

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

   Send your order form with payment to: Data General Corporation
   ATTN: Educational Services/TIPS G155
   4400 Computer Drive
   Westboro, MA 01581–9973

   b) TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	**SHIP TO:** (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME ______	COMPANY NAME ______
ATTN: ______	ATTN: ______
ADDRESS ______	ADDRESS (NO PO BOXES) ______
CITY ______	CITY ______
STATE ______ ZIP ______	STATE ______ ZIP ______

Priority Code ______ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side)	Title	Date	Phone (Area Code)	Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

□ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

□ UPS Blue Label (2 day shipping)
□ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 – $149.99	0%
$150 – $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) ______

ORDER TOTAL	
Less Discount See B	–
SUB TOTAL	
Your local* sales tax	+
Shipping and handling – See A	+
TOTAL – See C	

C PAYMENT METHOD

□ Purchase Order Attached ($50 minimum)
P.O. number is________. (Include hardcopy P.O.)
□ Check or Money Order Enclosed
□ Visa □ MasterCard ($20 minimum on credit cards)

Account Number □□□□□□□□□□□□□□□□ Expiration Date □□□□

______
Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

134-755-02

moisten & seal

CUSTOMER DOCUMENTATION COMMENT FORM

Your Name ______________________ Your Title ______________

Company ______________________ Phone ______________

Street ____

City ______________________ State ________ Zip ______

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ______________________ Manual No. ______________

Who are you?	☐EDP/MIS Manager	☐Analyst/Programmer	☐Other ________
	☐Senior Systems Analyst	☐Operator	________
	☐Engineer	☐End User	

How do you use this manual? *(List in order: 1 = Primary Use)*

___ Introduction to the product	___ Tutorial Text	___ Other
___ Reference	___ Operating Guide	________

		Yes	No
About the manual:	Is it easy to read?	☐	☐
	Is it easy to understand?	☐	☐
	Are the topics logically organized?	☐	☐
	Is the technical information accurate?	☐	☐
	Can you easily find what you want?	☐	☐
	Does it tell you everything you need to know?	☐	☐
	Do the illustrations help you?	☐	☐

If you wish to order manuals, use the enclosed TIPS Order Form (USA only) or contact your sales representative or dealer.

Comments:

Cut here and insert in binder spine pocket

DataGeneral

Data General Corporation, Westboro, Massachusetts 01580

069-100230-01

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 WESTBORO, MA 01581

POSTAGE WILL BE PAID BY ADDRESSEE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

DataGeneral

Customer Documentation
MS E-111
4400 Computer Drive
P.O. Box 4400
Westboro, MA 01581-9890

134-755-02

Form 702
Rev. 8/87

DATA GENERAL CORPORATION TECHNICAL INFORMATION AND PUBLICATIONS SERVICE TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Illegal octal digit
Illegal operation
Illegal preprocessor command
Illegal size for field
Illegal storage class
Illegal symbol
Illegal to initialize extern variable
Illegal to take the address of this object
Illegal type for field
Illegal type for function
Illegal type for member
Illegal use of typedef
Illegal variable or expression
Include nested too deeply
Incompatible type-specifiers
Indexing not allowed
Initializer too large
Inliner Fatal Error: Variable not on decllist.
Inliner: Cannot process ENTRY statements.
Integer expression required
Internal Compiler Error
Invalid type coercion
Label Expected
Label not defined:
Must be a structure
Must have at least one stack argument
No arguments may be specified after ...
No main or MAIN_ routine, reverting to context free inlining rules.
No name given for declaration
No name given for some parameter
No name given in declaration
Not enough arguments given
Not inside a loop
Not inside a loop or switch
Nothing declared in a declaration
Null dimension
Only one "default" case per switch statement allowed
Operand must be an lvalue
Pointer to procedure not legal here
Pointer to void not legal here
Preprocessor expression must be constant
REAL operand not allowed here
Ran out of string space
Redeclaration of prototype parameters
Redefinition of this builtin macro not permitted:
Storage class illegal here
Structure/union must have at least one member
The size of this variable is not defined
This compiler must be used on a licensed system.
This is a binary file
This is not an lvalue

-X491	Kanji character support.
-X496	Check to make sure all args/vars are used.
-X498	In ANSI C make string literals be const char*.
-X500	Don't delete .s file if errors encountered.
-X509	Complain about locals read before written.
-X510	(BSD only) Do not generate debug information for enum types.
-X523	Use same technique as -X424 except use for mod operator.
-X524	Equivalent to -X424 and X523, except for signed operands.
-X525	For machines without 33-bit shift capability for which you wish to activate -X424, -X523 and/or -X524. Replaces a divide or mod operation, which has a constant divisor, with two multiplies and a right shift.
-X540	Output .ident assembly code for #ident in C (-X187 overrides -X540).
-X543	Suppress slow divide optimizations. Equivalent to -Z424, -Z523, -Z524 and -Z525.
-X588	Put in code to generate a runtime error when an uninitialized variable is accessed. This allocates a shadow variable for each user variable, so it may require significant amounts of memory at runtime.
-X593	Allocate all static variables in the outer scope, even if they are never used. Intended for use with SCCS id strings.
-X598	Generate an extra instruction to trap on division by zero due to bug in 88000 chip.
-X611	Do not compile the program, but output a list of included files in a format convenient for use in a makefile.
-X614	Put the input source lines into the assembler output file. Lines from include files are not copied, macro substitution will not appear.
-X623	Unroll loops up to 8 times instead of the default of 4. Requires -OL.
-X625	Do not call memcpy to perform structure copies. Instead call GHS supplied assembly language routines (__gh_mvw, __gh_mvh, or __gh_mvb).
-X629	Output OCS compatible "tdesc" information for debugging if -g also specified.
-X640	Do not recognize _ _**inline** class specifier.

-X211	Suppress optimizations that generate inline code for strcpy() and strcmp() with constant arguments.
-X211	Suppress optimizations that generate inline code for external calls.
-X219	Suppress elimination of jumps to jumps.
-X230	Suppress common subexpression elimination and value propogation, except for trivial cases. This turns off a large class of optimizations, some of which may be disabled separately using -X468 and/or -X474.
-X233	Functions which return the type ''float'' return a single precision value, not a double precision value.
-X237	Apply associative rules in common subexpression elimination.
-X239	The host operating system of the compiler is MS-DOS. Change include file conventions etc. appropriately.
-X253	Do not remove code which will never execute because it is within an **if** statement with a constant expression equal to zero.
-X255	Print a brief description of enabled -X switches on the terminal.
-X264	Suppress phase that removes useless sign and zero extend instructions.
-X265	Suppress register database phase of peephole.
-X266	Repeat the peephole phase until the code doesn't get any better.
-X304	Truncate names to eight characters on input.
-X306	C ''asm'' inline directive not recognized. Note that the _ _asm directive is still recognized, only the asm directive without leading underscores is affected by this option.
-X307	Turn off instruction reordering (don't attempt to fill gaps between instructions). This makes output easier to read, but much slower.
-X308	Perform tail recursion optimizations.
-X311	Don't make multiple copies of blocks in merge blocks phase.
-X312	Suppress recognition of ?: operators as absolute value and min/max.
-X316	Enable all ANSI C extensions which are sensible in a UNIX environment. This is equivalent to -ansi. This is a subset of -X153. When this option is used in conjunction with -O, -OM is implied.
-X328	Disable register caching.
-X329	(BSD only) Generate ''stabd'' pseudo-ops for line numbers instead of stabn line numbers.
-X331	Allocate unused variables if symbolic debugging enabled (-g).
-X333	Suppress passing of front end information to the peephole optimizer and instruction scheduler.
-X334	The usual arithmetic rules will apply to operator assignments, as ANSI requires, rather than the Berkeley ''left side prevails'' rule. E.g., ''charvar *= 0.5'' will be performed using floating arithmetic.
-X338	Do not delete unused instructions during fixup phase.
-X344	Suppress adrconst optimizations. Do not try to undo ineffective allocation of constants to temporaries.
-X350	In ANSI C, allow /**/ to be a concatenation operator in a macro, as it is in the portable C compiler.

7.2. Special Compile Time Options

Each C-88000 compiler is configured to enable some of the compile time options described in this chapter and to disable the rest. Your compiler should have been configured so that it can be used in its intended environment without needing to specify any special compile time options. All normal compile time options are documented in the previous user option section.

However, if you need to use the compiler in an environment other than the one for which it was intended, or if you have unusual requirements, you may find that your other documentation may not give you enough information. Over the years, Green Hills has implemented many minor variations in the compiler for different customers. It is quite possible that you may find just the option you need in the list below. However, you should be warned that using option combinations that have not been recommended may produce strange or incorrect results.

There are a number of options which are intentionally left undocumented. The undocumented options are disabled, obsolete, or are for compiler debugging only. Using undocumented options may generate poor or incorrect code. Before the description of each option, enclosed in parentheses, there may be a restriction on the use of the option. It may specify a particular manufacturer or operating system. The option is only to be used when that restriction applies. Using an option when it is not allowed may cause all sorts of errors.

Most options are case sensitive and are only recognized as shown in the documentation. The compiler accepts but ignores any invalid option specifications, therefore the -v and/or -X255 options are often useful to verify which options were accepted and are in use by the compiler for the current compilation.

The -X prefix options are used to turn **on** a specific function. To negate the effect of the -X prefixed option, the -Z prefix is used instead. For example, -X308 turns on tail recursion optimizations. If this option is set by default for your compiler, using -Z308 will turn off tail recursion optimizations.

GREEN HILLS DOES NOT GUARANTEE THAT THE COMPILER WILL ACT AS YOU EXPECT WHEN YOU USE THESE OPTIONS. GREEN HILLS RETAINS THE RIGHT TO ABOLISH, CHANGE, OR WITHDRAW SUPPORT FOR ANY OPTION OR COMBINATION WITHOUT NOTICE.

-Xnnn	Turn on option number nnn, where nnn is an unsigned integer constant. The available compile time options are listed below.
-Znnn	Turn off option number nnn, where nnn is an unsigned integer constant. This is the reverse of the X option. This option is useful if a version of the compiler has some option turned on by default, and you want to turn it off.
-X6	Allocate each enum type as the smallest size predefined type which allows representation of all listed values (that is, from the list: ''char'', ''unsigned char'', ''short'', ''unsigned short'', ''int'' or ''unsigned''). The default is to allocate as an ''int''.
-X9	Disable local (peephole) optimizer.
-X13	Suppress code generation. An empty output file will be created.
-X18	Do not allocate programmer-defined local variables to a register unless they are declared register.

cannot generate output as fast as the UNIX ''cpp'' program, so use ''cpp'' for big jobs.

-g	(UNIX Target only) Generate source level symbolic debug information (if such a capability exists for the target system) and a frame pointer for stack traces. The amount and form of debug information varies with the capabilities of the target system. This option does not imply -ga.
-ga	Generate a frame pointer for stack traces. The default compiler setting is to optimize the program to the point that stack traces become impossible on some machines. This makes program debugging difficult. When debugging a program this option should be used. This option does not imply ''-g''.
-Istring	Include file names which are not absolute (do not start with ''/'' in UNIX) are searched for in the directory ''string'' before a standard list of directories. Multiple -I options can be specified. They will be searched in the order encountered.
-k+r	The -k+r option is recognized as either -k+r or -K+R. It is equivalent to -noansi and causes the C compiler to interpret the source code as standard (Kernighan & Ritchie) C. This is the default PCC compatible mode.
-p	(UNIX Host and Target only) Generate calls for execution profiling. The UNIX profiler must be available; a profiler is not part of the library provided by Green Hills.
-pg	(BSD UNIX Host and Target only) Generate more profiling information, and force all routines to have frames.
-o filename	Place the executable file output into the file named ''filename''. If this option is not specified the executable file will be named ''a.out''. This option is ignored if ''-c'' or ''-S'' is present.
-O	The -O option activates the Green Hills optimizers which are safe for use on all programs, except for the loop optimizer. If used in conjunction with -ansi (-X316) or -ANSI (-X153), -OM is assumed.
-OA	The -OA option provides algorithmic optimizations.
-OM	This option is eqivalent to -O except that it also allows the optimizer to assume that memory locations do not change except by explicit stores. That is, the optimizer is guaranteed that no memory locations are I/O device registers that can be changed by external hardware and no memory locations are being shared with other processes which can change them asynchronously with respect to the current process. This compile time option must be used with extreme caution (or not at all) in device drivers, operating systems, shared memory environments, and when interrupts (or UNIX signals) are present.
-OL	Optimize the program to be as fast as possible even if it is necessary to make the program bigger. In particular, most of the available resources are allocated to optimizations of the innermost loops. The -OL compile time option will perform optimizations which may make the program faster but larger. It is counter-productive to specify -OL on code which contains no loops or that is rarely executed as it will make the whole program larger but no faster. After experimenting with a program it is possible to discover which modules benefit from -OL and which ones do not. The -X482 option may be used in conjunction with -OL to allow various loop optimizations to be performed without turning on loop unrolling.
-OLM	This option is equivalent to -OL and -OM.
-OML	This option is equivalent to -OLM.
-S	Do not produce object files or executable files, produce only assembly language files. For each source language file specified, compile the source language file into assembly language output. Put the assembly language output into a file whose name ends in ''.s''.

C-88000 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 M88000 code generator. The code generator produces an internal representation of the M88000 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 reclaimed 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. Procedures which are more than 1000 lines or contain very complex statements can require several megabytes of memory to compile.

6.13. Detection of Portability Problems

Many of the problems associated with porting programs to C-88000 from other compilers can be detected with the UNIX utility program ''lint''. 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 programs that rely 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.

6.11.2. 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 instance, programs relying on ''register'' variables being allocated sequentially to pass hidden parameters will not work. Hidden returns (using ''return;'' and expecting to return the value of the last evaluated expression) will not work either.

6.11.3. Memory Allocation Assumptions

Memory is allocated by C-88000 in a different way than by PCC and other C compilers. Therefore, there can be 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. C-88000 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. C-88000 may not allocate unused variables. The -X331 (dbdebugallocateall) compile time option can be used to force all variables to be allocated even if they are never used. Some programs depend on knowing that certain variables will be allocated in memory. C-88000 will allocate certain variables to registers that PCC and other compilers would always allocate to memory. Programs compiled with C-88000 must not make assumptions regarding the order of allocation of variables in memory (except where the C language standard specifies it).

6.11.4. -OM Restrictions

The -OM and -OLM compile time options should only be used programs in which memory cannot change except under control of the compiler. The -OM and -OLM compile time options tell the compiler that memory locations do not change asynchronously with respect to the running program. In particular, if the compiler reads or writes some memory location, three instructions later it can assume that the same value is still in the memory location.

This simple assumption is not true for many parts of operating systems, device drivers, memory med I/O locations, shared memory environments, multiple process environments, interrupt driven routines, and when UNIX style signals are enabled. The -OM and -OLM compile time options MUST NOT be used in these cases. Use -O or -OL instead.

For example, many UNIX device drivers use memory locations which are I/O registers that can change at any time. In particular, a typical loop waiting for a device register to change is:

```
while (!io_register);
```

If -OM is specified when compiling this loop, the compiler will read the value of io_register only once. If io_register is zero when the loop is entered, zero will be loaded into a register and on each iteration of the loop the register value will be tested instead of the memory location. Whether or not the memory location is changed by an external device, under -OM the loop will never stop.

In ANSI and FullANSI modes, this problem can be avoided by declaring io_register with the **volatile** specifier. The compiler will then never assume it knows the value of that volatile object without reading it. Because the **volatile** specifier is available in ANSI and FullANSI modes, -OM is implied by -O.

identical to the evaluation order of other C compilers, some of these illegal programs which operate as expected with another C compiler may not operate the same way when compiled with C-88000.

Some implementations of the C Language evaluate the arguments to a function from right to left, others from left to right. See the M88000 Target chapter for details of the C-88000 calling conventions.

Expressions with side effects, such as function calls, and the operators ''++'', ''--'', ''+='', etc., may be executed in a different order by C-88000 and other C 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.

C-88000 may allocate some pointer variables not declared ''register'' to registers. This may allow C-88000 to generate more efficient sequences for post increment operators than other C 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 under C-88000.

A particular case of evaluation order dependency is the use of the ''?:'' operator in an expression which is an argument to a function call. C-88000 evaluates all question-mark operators before any other arguments, and keeps the result in a temporary. 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 C-88000.

6.10. PCC Mode Incompatibilities

The C Preprocessor that is provided with PCC has many undocumented features. Most of these undocumented features are implemented in C-88000 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 instance:

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

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

```
/* A comment */
int val;
```

Due to the one pass nature of C-88000 it is not possible for its builtin preprocessor to manufacture a token such as ''/*''. In order to compile a program with such constructs it is necessary to run C-88000 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 would normally.

However as a special case (-X350 in PCC mode) the compiler can construct an identifier as:

```
#define O 1
int val;
```

6.3. 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 the MC68000 and Z8000 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, 8086/88/286/386, NS32000, and Cler 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 of this is that some processors such as the Weitek-XL, M88000 and MIPS R2000 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 variable as type ''int'' in one module and as type ''char'' in another, may not work.

6.4. Alignment Requirements

C-88000 always aligns multiple byte data items on appropriate address multiples so that all accesses will be legal and efficient. The alignment conventions for C-88000 are defined in the M88000 Target chapter. It is possible for the compiler to guarantee that there will be no illegal or inefficient references if the programmer follows simple rules.

The size of all compound data types are rounded up to a multiple of the largest optimal and legal alignment of any component data type. The compiler always aligns parameters and local variables within the stack at an optimal and legal offset from the beginning of the frame. The compiler always rounds up the size of the frame to a boundary of the largest optimal and legal alignment of any data type. If the stack pointer is initially aligned to this boundary, and the program involves no explicit manipulation of the stack pointer, all stack references will be optimal and legal.

All variables within the global frame are allocated at an optimal and legal offset from the base of the global frame. If the assembler and/or linker allocates the global frame with the maximum optimal and legal alignment of the M88000, all global data references will be optimal and legal.

C-88000 will always ensure that components of a data structure requiring alignment will appear only at an optimal and legal offset from the beginning of the data structure. If all allocation routines always return pointers which are aligned to the maximum optimal and legal alignment of the M88000 and the program does not use (or correctly uses) integer arithmetic for pointer computations, all references to dynamically allocated memory will be optimal and legal.

Variables within a frame or components within a larger data type are optimally packed together in memory. When a data type has an alignment requirement, the least possible unused space is left between the end of the previous item and the next item so that the next item can be optimally aligned.

In satisfying different alignment requirements, complex data types may be allocated differently on different machines. This will lead to the usual problems with programs which rely on memory

5.3.2. Strength Reduction

Strength reduction is found only in the most advanced compilers. It applies to loops which have an index variable which is incremented by a constant on each iteration of the loop. When a loop index variable is used as the subscript for an array, most compilers will multiply the loop index by the size of the array elements and add this offset to the base of the array. Each such reference will typically require at least three instructions. After the application of strength reduction, outside of the loop, a register is loaded with the address of the array element to be accessed on the first iteration of the loop. The array access is replaced by an indirect register addressing mode. On each iteration, the element size is added to the register so that it contains the address of the element to be accessed on the next iteration of the loop. This optimization results in a four to twenty times speed improvement.

Strength reduction also reduces multiplication of the loop index by a loop invariant value to addition of a constant to a register.

5.4. Pipeline Instruction Scheduler

Sometimes the time it takes to execute an instruction depends on the instructions that precede it. When an instruction is executing and a second instruction is encountered which attempts to access the result register or the same functional unit before the first instruction has completed execution, the execution of the second instruction encounters a pipeline delay until the first instruction has completed execution. Another instruction which operates on different registers and functional units may be executed at no cost in the pipeline delay between the two instructions.

C-88000 simulates the timing of M88000 instruction sequences. When C-88000 detects that an instruction sequence will result in a pipeline delay, C-88000 attempts to reorder the instruction sequence so that the execution results remain the same, but the total execution time is reduced. Pipeline delays tend to happen frequently, since the result of an expression is often used immediately after it is computed. The instruction scheduler can often greatly reduce the total time that a sequence of instructions will take.

```
        jmp.n   r1
        addu    r31,r31,8
```

The improvement by the C-88000 optimizer can be summarized as:

Static Address Elimination	6 instructions
No frame pointer	3 instructions

5.2.5. Register Coalescing

Register Coalescing organizes the computation of expressions to ensure that values end up in the registers where they will be needed. This eliminates shuffling the values in registers to get them set up as needed. Most microprocessor compilers will copy the arguments of a computation into scratch registers; do the computation in the scratch registers; then copy the result to the destination. C-88000 will use the destination register in the computation in order to avoid unnecessary register to register copies.

For example the C-88000 compiler will compile the statement ''i = i*100+j;'' as follows (i is in r2, j is in r3):

```
        mul     r9,r2,100
        addu    r2,r9,r3
```

The improvement by the C-88000 optimizer is: 4 bytes and 2 instructions.

5.2.6. Passing Parameters in Registers

In C-88000, most parameters are passed to a function in registers rather than by pushing them on the stack. This avoids the memory accesses involved in pushing parameters onto the stack and in accessing the parameters from within the function. Further improvement comes from organizing the computation of parameter values so that the value ends up in the register in which the value is to be passed to the function. Finally, the necessity of removing the parameters from the stack after the call returns is eliminated with register parameters. This optimization reduces the code size of most programs by twenty percent.

For example, the expression ''g(f(x+y))'' will compile as follows:

(x is in r8 and y is in r9)

```
    Register Parameters           Parameters on the Stack
                                  subu      r31,r31,16
                                  addu      r9,r8,r9
    bsr.n       _f                bsr.n     _f
    addu        r2,r8,r9          st        r9,r31,0
                                  addu      r31,r31,8
    bsr         _g                bsr.n     _g
                                  st        r2,r31,0
                                  addu      r31,r31,8
    proc()
    {
        int i, j;

        for (i = 1; i < 10; i++)
            f();
        for (j = 1; j < 10; j++)
            g();
    }
```

```
_proc:
        subu    r31,r31,16
        st      r1,r31,12
        st      r25,r31,0
        or      r25,r0,1
L%7:
        bsr     _f
        addu    r25,r25,1
        cmp     r10,r25,10
        bb1     lt,r10,L%7
        or      r25,r0,1
L%4:
        bsr     _g
        addu    r25,r25,1
        cmp     r10,r25,10
        bb1     lt,r10,L%4
        ld      r1,r31,12
        ld      r25,r31,0
        jmp.n   r1
        addu    r31,r31,16

; _i    r25     local
; _j    r25     local
```

The improvement by the C-88000 optimizer can be summarized as:

Put i and j in r25	2 memory references per iteration
Rotate Loop	1 instruction per iteration

5.2.2. Memory Allocation

C-88000 allocates variables based on their size, frequency of use, and other factors. Variables which are never used are usually not allocated. Variables are usually sorted to allocate the smaller and more frequently used variables first, and the larger and less frequently used variables later. This allows the use of small address offsets to access commonly used variables. If the compiler allocated some very large variable first, small address offsets might not be able to access variables allocated after it. By putting the smallest and most frequently used variables first, the compiler makes the greatest possible use of small address offsets. Some variables which other compilers would allocate in memory are allocated in registers as explained in the section ''Register Allocation by Coloring''.

enum (-X6)	8,16,32	8,16,32

4.3. Compiler Output Format

The output of the compiler is UNIX System V M88000 Assembler Language.

The -g option generates Common Object File Format (COFF) ''sdef'' symbolic debug pseudo-ops in the assembler language output. The assembler and linker understand and process the symbolic debug entries in the object files. The ''sdb'' symbolic debugger can be used with C-88000 output.

4.4. Register Usage

There are 32 general purpose registers used for both integer and floating point values. Double precision 64-bit floating point values are contained in two adjacent 32-bit registers. The high word is stored in an even numbered register, the low word is stored in the adjacent register, for example r14 and r15.

Register	Use
r0	Always contains a zero value
r1	Return address to calling function
r2..r9	Register arguments to called function
r10..r13	Temporary registers not saved across calls
r12	Structure return address (upon entry as required)
r14..r25	Permanent registers saved across calls
r26..r29	Unused, reserved for system software use
r30	Frame pointer
r31	Stack pointer

4.5. Calling Conventions

In order for a function to be able to access double precision arguments and local variables on the stack, a convention has been adopted that the stack must always be located at an 8 byte boundary at the entry to a function. If the first function (called by the operating system or system library) is called correctly then every other function will be called correctly.

Arguments are evaluated from right to left.

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

Each floating point argument is extended to a 64 bit value after it is evaluated, unless the corresponding formal parameter is either single precision or scalar. If the formal parameter is single precision, the argument is truncated to a 32 bit floating point value. If the formal parameter is scalar, the argument is truncated to a 32 bit scalar value.

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

2.3. New 1.8.5 Features

2.3.1. Compile Time Options

A number of new compile time options are introduced in C Version 1.8.5. These new options are summarized in the table below. Additional details on each option can be found in the *Compile Time Options* chapter.

Option	Description and -X equivalents (if any)
-fullinline	Turn on inlining optimizations (-X249)
-QI *name*	Inline specified routine
-ansi	Select ANSI mode (-X316)
-ansi -s	Select FullANSI mode (-X153)
-k+r	Select PCC mode (default)
-OA	Algorithmic optimizations

2.3.2. asm

The **asm** statement may now be placed outside of a function, when the -X436 option is used.

2.3.3. Preprocessing Directives

The following new preprocessing directives are introduced in 1.8.5 to conform to the new ANSI standard: **#error**, **#pragma**, and the # and ## operators.

#pragma ident *version-string* allows programmer-supplied version information to be stored in the executable image.

#ident *version-string* is an alternate form of the #pragma ident directive and is intended for programmers wishing to develop more portable C code. A warning message will be generated if code containing an #ident directive is compiled in FullANSI mode.

2.3.4. Predefined Preprocessing Macros

_ _DATE_ _ is a predefined preprocessor symbol available in ANSI and FullANSI modes. Its value is a character string literal in the form "Mmm dd yyyy" containing the system date that the source was compiled.

_ _TIME_ _ is a predefined preprocessor symbol available only in ANSI and FullANSI modes. Its value is a character string literal in the form "hh:mm:ss" containing the system time that the source was compiled.

(1) The line ``#include <varargs.h>'' must appear before the first function definition.

(2) The last parameter to 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''. There must not be a semicolon between ``va_dcl'' and the initial left brace(``{'') of the function.

(4) There must be a variable declared in the function of type ``va_list''.

(5) 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''.

(6) 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.

(7) 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>
Sum(x, va_alist)            /* Sum returns the sum of a variable number */
int x;                                /* of ``int'' arguments                    */
va_dcl
{ va_list params;
          int ret = 0;
          va_start(params);
          while (x != 0) {
                    ret += x;
                    x = va_arg(params, int);
          }
          va_end(params);
          return(ret);
}
```

2.2.4. Bit Fields

C-88000 supports signed and unsigned bit fields. Unsigned bit fields are recommended for most applications since they are more efficient to fetch on most machines. For compatibility with the VAX 4.2BSD implementation of C, a compile time option (-X55), is provided which specifies that a field whose type is signed is to be interpreted as a signed quantity. The consequences of having signed fields can be seen in the following example.

```
{
        struct {int x:2;} y;
        y.x = 3;
        i = y.x;
}
```

In this example, if ``x'' is an unsigned field, ``i'' will have the value of 3 at the end of the block. However, if signed fields are accepted, ``i'' will have the value -1 at the end of the block.

2.1.1. Preprocessor

C-88000 includes a preprocessor which is functionally identical to the UNIX C preprocessor. The basics of the preprocessor are explained in Kernighan and Ritchie, but as with the compiler, the actual preprocessor is far more complex. Unlike PCC which depends on an initial text processing pass by a preprocessor program, C-88000 preprocesses the input program in the compiler itself. This makes the compilation process faster because the source program is read only once and one less process is run.

Preprocessed output may be saved to standard output by using the -E compile time option on the command line. Normally, preprocessing is performed concurrently with compilation and no temporary output is generated.

2.1.2. Predefined Identifiers

_ _LINE_ _ is a predefined preprocessor symbol whose value is a character string which consists of the ASCII representation of the current line number within the current file. It is available in all three compiler modes.

_ _FILE_ _ is a predefined preprocessor symbol whose value is a character string which consists of the ASCII representation of the current file name. It is available in all three compiler modes.

The following predefined identifiers have two different representations. The identifiers with two leading underscores (_ _) are available in all three compiler modes (PCC, ANSI and FullANSI). The same identifiers without the underscore prefixes are also available in ANSI and PCC modes.

_ _ghs is a predefined preprocessor symbol always defined by all Green Hills C compilers.

_ _unix is a predefined preprocessor symbol on Unix targets.

_ _vms is a predefined preprocessor symbol on DEC VMS target systems.

_ _BigEndian or _ _LittleEndian is a predefined preprocessor symbol that reflects the machine byte order on the target processor.

_ _IeeeFloat and _ _VaxFloat are predefined preprocessor symbols that reflect the type of floating point utilized by the target machine.

_ _m88k is a predefined preprocessor symbol on Motorola 88000 targets.

2.1.3. 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 two structures or unions is done with a memory copy of the data. Comparison is done on a bit by bit basis of the total size of the structure or union.

If there are holes between fields or members of a structure or union due to memory alignment requirements, those holes cannot be accessed. 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

C Runtime Library

This section describes the C-88000 runtime library.

Compile Time Options

This chapter describes how to adjust the output of Green Hills C to accommodate your needs by using the many variations that have been implemented.

Runtime Errors

This table lists the C runtime errors.

Compile Time Errors

This table lists the C compile-time errors.

Green Hills Software C-88000™ User's Manual
069-100230-01

Effective with C Compiler for the M88000, Version 1.8.5