June 1990
Order Number: 311567-004

iPSC®/2 and iPSC®/860
C LANGUAGE
REFERENCE MANUAL

intel® Corporation

REV.	REVISION HISTORY	DATE
–001	Original Issue	3/88
-002	Revision	11/88
-003	Revision	03/89
-004	Revision	06/90

Disclaimer

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

RESTRICTED RIGHTS

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at 52.227-7013. Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

PREFACE

This manual tells how to use the Green Hills C compilers for the following iSC products: iPSC®/2, iPSC®/2S, iPSC®/860, iPSC®/860S, and iPSC®/860Plus.

NOTE

This manual uses the term "iPSC system(s)" to refer to these products.

This manual assumes that you are an application programmer proficient in the C and Fortran languages and the UNIX operating system. The manual provides you with enough detail to begin using your iPSC system.

ORGANIZATION

Green Hills Software C-386 User's Guide
Describes the C compiler for the iPSC/2 systems.

Green Hills Software C-860 User's Guide
Describes the C compiler for the iPSC/860 systems.

Green Hills Software User's Manual C Library
Describes the C Library.

APPLICABLE DOCUMENTS

For more information, refer to these iPSC, Intel and other manuals:

iPSC® System Manuals

iPSC®/2 and iPSC®/860 Math Libraries Reference Manual
Describes the math libraries available on the iPSC system.

iPSC®/2 and iPSC®/860 Programmer's Reference Manual
Describes iPSC system commands and system calls (both C and Fortran).

iPSC®/2 and iPSC®/860 System Acceptance Test User's Guide
Tells how to use the System Acceptance Test.

iPSC®/2 and iPSC®/860 System Administrator's Guide
Describes the system administration tasks related to operating and maintaining an iPSC system.

iPSC®/2 and iPSC®/860 User's Guide
Overviews the iPSC system, including hardware and software architectures. Tells how to develop and run programs.

iPSC®/2 and iPSC®/860 VME Interface Reference Manual
Describes the installation and development of software drivers for the VME Interface Adapter board.

iPSC®/2 DECON User's Guide
Tells how to use DECON, the iPSC/2 concurrent debugger.

iPSC®/2 DECON User's Guide RX Beta Change Notice
Describes RX Beta Changes to DECON commands.

iPSC®/2 Simulator Manual
Tells how to use the iPSC/2 Simulator for software development.

iPSC®/2 VAST2 User's Guide
Tells how to use the iPSC/2-VX version of VAST2 software.

iPSC®/2-VX User's Guide
Describes development of programs for the iPSC/2-VX processing system.

Intel® Manuals

Intel® NFS for System V/386 Programmer's Guide and Reference
Describes the NFS programming environment and tools.

Intel® NFS for System V/386 User's/System Administrator's Guide and Reference
Describes the NFS programming environment and provides user and system administration information.

Intel® TCP/IP for SYSTEM V/386 Administrator's Guide and Reference
Describes TCP/IP Network administration.

Intel® TCP/IP for SYSTEM V/386 Programmer's Guide and Reference Manual
Describes the TCP/IP Network programming environment and provides information on programming tools.

Intel® TCP/IP for SYSTEM V/386 User's Guide and Reference
Describes the TCP/IP Network programming environment and provides user information.

i860™ 64-Bit Microprocessor Assembler and Linker Reference Manual
Tells how to use the i860 assembler and linker.

i860™ 64-Bit Microprocessor Programmer's Reference Manual
Tells how to use the i860 microprocessor.

SYP301 Installation and User's Guide
Tells how to install and start the System Resource Manager. Also provides hardware technical data.

Other Manuals

C: A Reference Manual - Harbison and Steele
Describes the C programming language.

The C Programming Language - Kernighan and Ritchie
Describes the C programming language.

UNIX System V Manual Set
Describes UNIX System V.

C-386 User's Guide

Version 1.8.4

C Compiler for the 386™ microprocessor

August 1989
Green Hills Software, Inc.

Disclaimer

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

Table of Contents

CHAPTER 1

Overview

1.1. How to Use This Manual

Each Green Hills compiler is specified by three components: the Language, the Target, and the Host. The Language, in this case C, is the computer language that the compiler translates. The Target, in this case the 80386, is the machine on which your program will run. The Host is the computer system on which the compiler runs. The organization of this manual is given below.

Overview

> The Overview describes the structure of the documentation for C-386.

The C Language

> The C Language section specifies the C language features and extensions that are supported. It also explains restrictions and indicates how closely C-386 matches other C compilers.

80386 Target

> The 80386 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 C-386 to improve program performance. It also gives you general ideas as to how to get the best performance out of your program.

Porting Programs to C-386

> This chapter tells you about difficulties that you may encounter in moving a program developed with another compiler to C-386. It gives specific examples of difficulties that may be encountered and how to resolve them.

Compile Time Options

> This chapter describes how to adjust the output of C-386 to accommodate your needs by using the many variations that have been implemented.

1.2. Related Documentation

You will need documents in addition to this manual in order to use C-386. These documents will tell you how to install C-386 and how to compile, link, execute, and debug C-386 programs. You may also need documentation describing your 80386 assembler and linker, and the 80386 architecture.

C Language Reference Manual

> This manual assumes that you have a standard reference text on the C Language.

If you do not have such a text, the C Language section will refer you to one.

C Library Documentation

This manual does not contain a description of the C library functions which can be called with C-386. The availability of runtime library routines depends on the target environment. On the iPSC/2, the C library is the ATT SYSTEM V release 3.0 and is documented in the UNIX System V/386 Programmer's Reference Manual.

CHAPTER 2

The C Language

2.1. Introduction

C-386 is a complete implementation of the C programming language. The basic C language is defined in "The C Programming Language" by Kernighan and Ritchie (Prentice-Hall, 1978). This specification is very terse, often imprecise, occasionally misleading, and far from complete. There is an ANSI standardization committee working on a standard definition of C, but at this time there is no other authoritative definition of the C language. 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 our definition of the C language.

C-386 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 below, but this is by no means a complete list.

If you have a UNIX version of this compiler, the documentation provided with UNIX (Kernighan and Ritchie and the Western Electric extensions), and this document constitute the user documentation of C-386.

If you have any other version of C-386, the user documentation consists of Kernighan and Ritchie (which may be obtained from Green Hills) and this document.

2.2. Additions to the basic C Language

2.2.1. Preprocessor

C-386 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-386 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.

2.2.2. Backslash v

Lower case v is a special backslash character denoting vertical tab.

2.2.3. void type

There is a type named void. There are no operations defined on the type void. Void is used as the return type for functions which do not return a result.

2.2.4. __LINE__

__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.

2.2.5. __FILE__

__FILE__ is a predefined preprocessor symbol whose value is a character string which consists of the ASCII representation of the current file name.

2.2.6. Structure and Union Extensions

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 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.

2.2.7. Enumeration Type

There is an enumeration type similar to that of Pascal. Its syntax is similar to that of the struct and union declarations.

```
<enum-specifier>:
        enum { <enum-list> }
        enum <identifier> { <enum-list> }
        enum <identifier>

<enum-list>:
        <enumeration-declaration>
        <enumeration-declaration> , <enum-list>

<enumeration-declaration>:
        <identifier>
        <identifier> = <constant-expression>
```

Example:

```
        enum color {red, white=4, blue};
```

The enumerated type name may be the same as the name of a variable in the same scope but may not be the same as the name of any struct or union in the scope. Each enumeration-declaration declares a scalar constant of the enumeration type. If a constant-expressions appears in an enumeration-declaration it specifies the ordinal value of the constant. If no constant-expression is given in an enumeration-declaration, the value of the constant for the first enumeration-declaration is zero, and for subsequent enumeration-declarations the value is one greater than the value of the previous enumeration-

declaration.

Enum types are signed by default for compatibility with PCC. A compile time option is described below which changes the definition of enum types to unsigned, which is a more rational form.

2.2.8. The VARARGS(3) Facility

C-386 supports the UNIX VARARGS(3) facility. The VARARGS(3) 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(3) facility:

(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(3) 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(3) 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.

```
/* Sum returns the sum of a variable number of "int" arguments. */
#include <varargs.h>
Sum(x, va_alist)
int x;
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);
}
```

2.3. Bit Fields

C-386 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.

2.4. 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-386.

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".

2.5. Unsigned Char and Unsigned Short Int

The data types "unsigned char" and "unsigned short int" are not defined in the Kernighan & Ritchie C manual. But they are supported by C-386 and by many implementations of PCC.

There appear to be numerous bugs and/or inconsistencies in the way different versions of PCC evaluate expressions involving unsigned char and unsigned short. Green Hills has attempted to follow the VAX 4.2BSD compiler whenever possible.

2.6. asm Statement

The asm statement (for in line assembly code) in C-386 is somewhat different than the asm construct in PCC. In C-386 the asm statement can be used anywhere a statement can appear. In PCC, the asm construct is also allowed to appear in declarations and between functions.

Since the code generated by C-386 is substantially different than the code generated by other compilers it is usually necessary to modify most asm statements.

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

2.7. C Runtime Libraries

On UNIX systems, C-386 can use the standard C library. This is the recommended approach because the UNIX libraries are very extensive.

CHAPTER 3

80386 Target

3.1. Introduction

This chapter describes the 80386 target environment for C-386. There are currently two target environments: UNIX System V, and MS-DOS or cross development using Phar Lap tools.

3.2. 80386 Characteristics

The 80386 memory is byte addressed with 32 bit addresses. Bytes are ordered with the least significant byte of a multiple byte value stored at the lowest address, as on the DEC VAX, opposite of the IBM/370. Bits are numbered with bit zero as the least significant bit.

Floating point is IEEE 754 format (32 and 64 bits), least significant byte at the lowest address. The Intel 80287/80387 is supported by default; the Weitek 1167 coprocessor is supported with the -X143 switch.

Character encoding is ASCII.

The stack is always four byte aligned. Bit fields are allocated starting at bit zero. 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	32
unsigned	32	32
unsigned short	16	16
unsigned char	8	8
enum (default)	32	32
enum (-X6)	8,16,32	8,16,32

3.3. UNIX System V Target Environment

The output of the compiler is UNIX System V 80386 Assembler Language by default. Phar Lap code, for use under MS-DOS or other environments, can be generated with the -X214 switch.

The -g option generates Common Object File Format (COFF) ".def" symbolic debug pseudo-ops in the System V 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-386 output.

3.3.1. Calling Conventions

Arguments are evaluated from right to left. Each argument is pushed on the stack after it is evaluated.

The stack is always aligned on a four byte boundary.

Each scalar argument is extended to a 32 bit value before it is pushed on the stack. Each floating point argument is extended to a 64 bit value before it is pushed on the stack.

All other values are extended to a multiple of 4 bytes and pushed on the stack (the extra bytes are at the high addresses).

Scalar values are returned in EAX. When the size of the return value is specified as less than 32 bits only the required number of bits can be depended on in EAX.

Floating point values are returned in the top stack entry (FP0) in the 80287/80387 environment. They are returned in f2 or f2/f3 in the Weitek 1167 environment.

A call to a function uses the "call" instruction. The return from a function uses the "ret" instruction.

A function is assumed to destroy EAX, ECX, and EDX. The Weitek registers FP1 through FP23 are also assumed to be destroyed. The condition codes are undefined at the return of a function. All other registers are saved and restored by a function if they are used.

The compiler has two options for generating the local frame for a function. By default, no frame pointer is saved, all accesses to parameters on the stack are done with the ESP relative addressing mode, and EBP is used as a scratch register.

If the -g or -ga compile time options is specified, or if there is local data space, the function will save the old frame pointer and set up a new one on entry and restore the old frame pointer on exit. Accesses to parameters or local stack storage will be made with EBP relative addressing modes.

Following the return of the function, any arguments pushed on the stack are removed. Parameters are removed from the stack by an add immediate to the stack pointer.

CHAPTER 4

Optimization

4.1. Introduction

C-386 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-386 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.

4.2. General Optimizations

General Optimizations always make programs smaller and faster.

4.2.1. 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-386 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-386 is much better than most C compilers in its register allocation.

In the following example, C-386 allocates i and j to the same register because their lifetimes do not overlap.

```
proc()
{
        int i, j;

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

```
          C-386                                  UNIX PCC
          ----------------------                 ----------------
proc:                                    proc:
          pushl   %esi                           pushl   %ebp
                                                 movl    %esp,%ebp
                                                 subl    $8,%esp
          movl    $1,%esi                        movl    $1,-4(%ebp)
.L7:                                     .L17:
          call    f
          incl    %esi
                                                 movl    $10,%eax
          cmpl    $10,%esi                       cmpl    %eax,-4(%ebp)
          jl      .L7                            jl      .L18
          movl    $1,%esi                        movl    $1,-8(%ebp)
                                                 jmp     .L22
                                         .L18:
                                                 call    f
                                                 incl    -4(%ebp)
                                                 jmp     .L17
.L4:                                     .L23:
          call    g                              call    g
          incl    %esi                           incl    -8(%ebp)
                                         .L22:
                                                 movl    $10,%eax
          cmpl    $10,%esi                       cmpl    %eax,-8(%ebp)
          jl      .L4                            jl      .L23
          popl    %esi                           leave
          ret                                    ret

/i        %esi    local
/j        %esi    local
          ----------------                       ----------------
          35 bytes                               62 bytes
```

The savings by C-386 can be summarized as:

Put i and j in %esi	15 bytes
Improve enter/exit code	2 byte
Use cmpl $10	6 bytes
Rotate loop	4 bytes

The improvement by the C-386 optimizer can be summarized as:

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

4.2.2. Memory Allocation

C-386 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".

4.2.3. 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-386 does not set up a frame pointer in each function. C-386 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.

4.2.4. Static Address Elimination

A valuable optimization performed by C-386 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);
}
```

C-386		UNIX PCC	
p:		p:	
pushl	%esi	pushl	%ebp
		movl	%esp,%ebp
movl	$f,%esi		
pushl	$1	pushl	$1
call	*%esi	call	f
popl	%ecx	popl	%ecx
pushl	$2	pushl	$2
call	*%esi	call	f
popl	%ecx	popl	%ecx
pushl	$3	pushl	$3
call	*%esi	call	f
popl	%ecx	popl	%ecx
pushl	$4	pushl	$4
call	*%esi	call	f
popl	%ecx	popl	%ecx
popl	%esi	leave	
ret		ret	
28 bytes		37 bytes	

The savings by C-386 can be summarized as:

Static Address Elimination	7 bytes
Simplified entry code	2 bytes

4.2.5. 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-386 will use the destination register in the computation so as to save unnecessary copies of the source registers into scratch registers.

For example the C-386 compiler will compile the statement "i = i*100+ j;" as fol-

lows (i is in %edi and j is in %esi):

C-386		UNIX PCC	
imull	$100,%edi,%edi	imull	$100,%edi,%eax
addl	%esi,%edx	addl	%esi,%eax
		movl	%eax,%edi

4.2.6. 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.

4.2.7. 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.

4.3. 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-386 that most computation is performed in inner loops. When this compile time option is specified, C-386 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.

4.3.1. 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.

4.3.2. 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.

CHAPTER 5

Porting Programs to C-386

Some programs which appear to compile and operate correctly when compiled with other C compilers, may not operate correctly when compiled with C-386. The C Language specifications define legal programs in such a way that legal programs will always work with all C compilers, including C-386. The problem is that many programmers make illegal assumptions about the machine or compiler that they are using. This chapter discusses many illegal assumptions which can cause programs to fail when compiled with C-386.

5.1. 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 with in your C-386 program.

The implementation of each Green Hills C Compiler is the same for each Green Hills Target. Therefore, legal programs written in C-386 can be moved to any other Green Hills C Compiler.

C-386 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.

5.2. 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.

5.3. 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 decendents of the PDP-11, such as the VAX, 8086/88/286/386, NS32000, and Clipper have followed the DEC convention. These two groups seem to be so well entrenched that no agreement on byte ordering is possible.

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.

5.4. Alignment Requirements

C-386 always aligns multiple byte data items on appropriate address multiples so that all accesses will be legal and efficient. The optimal alignment is the largest alignment required by any data type for optimal access. It is typically the word size or the external bus width. The alignment conventions for C-386 are defined in the 80386 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 alignment of any component data type. The compiler always aligns parameters and local variables within the stack at an optimal offset from the beginning of the frame. The compiler always rounds up the size of the frame to a boundary of the largest optimal alignment of any data type on the 80386. 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.

All variables within the global frame are allocated at an optimal offset from the base of the global frame. If the assembler and/or linker allocates the global frame with the maximum optimal alignment of the 80386, all global data references will be optimal.

C-386 will always ensure that components of a data structure requiring alignment will appear only at an optimal offset from the beginning of the data structure. If all allocation routines always return pointers which are aligned to the maximum optimal alignment of the 80386 and the program does not use (or correctly uses) integer arithmetic for pointer computations, all references to dynamically allocated memory will be optimal.

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

5.5. 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-386 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" appear 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 depence on the character collating sequence. If a program is designed to operate with a collating sequence other that ASCII it may be necessary to modify string and character comparison code to operate with ASCII.

5.6. 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.

5.7. 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-386. Refer to your target operating system documentation for a description of legal file names for your environment.

5.8. 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.

5.9. 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-386 evaluation order is not 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-386.

Some implementations of the C Language evaluate the arguments to a function from right to left, others from left to right. See the 80386 Target chapter for details of the C-386 calling conventions.

Expressions with side effects, such as function calls and the operators "++", "--", "+=", etc., may be executed in a different order by C-386 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 each point in the

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

C-386 may allocate some pointer variables not declared "register" to registers. This may allow C-386 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-386.

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-386 evaluates the question-mark operator 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-386.

5.10. C Preprocessor Incompatibilities

The C Preprocessor that is provided with PCC has many undocumented features. Most of these undocumented features are implemented in C-386.

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 */
```

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-386 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-386 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 the compiler can construct an identifier as:

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

Which becomes (in both PCC and C-386)

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

5.11. 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-386, 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.

5.11.1. Problems with Setjmp and Longjmp

Under the default configuration of C-386, 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 contents of the "label" variable and continues executing after the call to setjmp. Under the portable C compiler 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 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 (dbnamemem) 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.

5.11.2. 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.

5.11.3. Memory Allocation Assumptions

Memory is allocated by C-386 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-386 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-386 may not allocate unused variables. Some programs depend on knowing that certain variables will be allocated in memory. C-386 will allocate certain variables to registers that PCC and other compilers would always allocate to memory. Programs compiled with C-386 must not make assumptions regarding the order of allocation of variables in memory (except where the C language standard specifies it).

5.11.4. -OM Restrictions

The -OM and -OLM compile time options should only be used in algorithmic programs, that is, 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 mapped 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.

For example, most 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

5.11.5. Problems with Source Level Debuggers

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.

5.12. Problems with Compiler Memory Size

C-386 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-386 requires 300 Kbytes just for the program. It is designed to work best when it has at 1 Mbyte 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 into 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.

C-386 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 80386 code generator. The code generator produces an internal representation of the 80386 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 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. Simple functions 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 more than a megabyte of memory to compile.

5.13. Detection of Portability Problems

Many of the problems associated with porting programs to C-386 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.

CHAPTER 6

Compile Time Options

This chapter describes the command line switches and compile time configuration options that may be used with the C-386 compiler. The compiler has been pre-configured to easily generate programs for execution in the following four environments:

- Standalone program running on a system resource manager.
- Host program running on a system resource manager.
- Node program using the 80387 floating point coprocessor.
- Node program using the SX scalar floating point option.

You may want to modify the default configuration options if you want to use the compiler for an environment other than these or if you have unusual requirements. Modifications to the default configuration options may produce undefined and/or undesirable results, as Intel Scientific Computers has tested the compiler only for the environments listed above. The default configurations are changed by enabling and/or disabling one or more of the configuration options.

The compiler generates 80387 floating point coprocessor code for system resource manager programs. If your system contains both the VX vector option and SX scalar option, you can generate your node programs for the SX environment to take advantage of both options at the same time.

Use the following command line switches to generate executable code for the four supported environments.

Standalone system resource manager program: (no switches)
System resource manager host program: -host
Node program using 80387: -node
Node program using SX option: -node -sx

These and the other command line switches are explained below.

Command Line Switches

-both

Links in single and double precision routines for vectorization.(Only for VX Option)

-B

Implement a general 'debug enviroment' with all default compile optimizations disabled and code generation for iPSC's debugger DECON enabled.

-c

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.

-double

Links in double precision routines for vectorization.(Only for VX Option)

-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

For file names ending with the ".F" extension, define "name" to the preprocessor with the value "string". This is equivalent to putting "#define name string" at the beginning 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 cannot generate output as fast as the UNIX "cpp" program, so use "cpp" for big jobs.

-host

Link in the libraries required for host programs running on a system resource manager.

-g

Generate source level symbolic debug information and a frame pointer for stack traces.

-ga

Generate a frame pointer for stack traces. The default compiler setting is to optimize the program to the point that stack traces may become impossible, making program debugging difficult. When debugging a program this option should be used. This option does not imply "-g".

-Istring

For file names ending with the ".F" extension, search for include files which are not specified with absolute path names (do not start with "/") in the directory "string", before searching the standard include file directories. Multiple -I options can be specified, and directories will be searched in the order that the -I options are listed. This option applies only to files that are included by the C preprocessor, using a '#include "filename" ' directive. This option does not affect the Fortran INCLUDE statement.

-lname

Include the specified library at link time. The normal UNIX library search sequence is used.

-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.

-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.

-OLM

This option is equivalent to -OL and -OM.

-OML

This option is equivalent to -OLM.

-node

Link in libraries required for node programs.

-p

Generate calls for execution profiling. The UNIX profiler must be available; a profiler is not part of the library provided by Green Hills.

-R

Put all data in the text section.

-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".

-single

Links in single precision routines for vectorization.(Only for VX Option)

-sx

Generate code for the SX scalar option and substitute the appropriate libraries at link time.

-Uname

Undefine the predefined preprocessor symbol "name". This is equivalent to putting "#undef name" at the top of the source file.

-v

Have the compiler driver print out the program name and command line arguments as it runs each subprocess.

-vec

Links in the appropriate VecLib library.(Only for VX Option)

-vecdbg

Links in the appropriate debug version of Veclib library.(Only for VX Option)

-vx

Directs the linker to allocate data to VX memory and link in appropriate Veclib routines.(Only for VX Option, see iPSC/2 VX User's Guide for essential details.)

-w

Suppress warning diagnostics.

The remainder of the chapter lists the compiler configuration options. To see how the options are set when you select one of the four supported environments (no switch, -host, -node, or -sx), add the -v switch to the command line when compiling a program. The general form of these options is as follows:

-Xnnn
: Turn on configuration option number nnn, where nnn is an unsigned integer constant.

-Znnn
: Turn off configuration option number nnn, where nnn is an unsigned integer constant. This is the reverse of the X option, and is useful if you want to turn off an option that is enabled by default.

Configuration Options

-X6 Allocate each enum type as the smallest size predefined type which allows representation of all listed values (that is, from the list: "char", "short", "int", "unsigned char", "unsigned short", 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.

-X32 Display the names of files as they are opened. This is 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 fields of type int, short, and char be signed. The default is for all 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-386.

-X84 Generate error messages for C anachronisms. By default the old assignment operators (=+ =- ...), initialization (int i 1), and references to members of other structures compile correctly but generate warning messages.

-X85 (UNIX Target only) Generate ".lcomm" (BSD 4.2) or ".bss" (UNIX System V) for zero initialized statics. The default is to allocate initialized data.

-X89 Pack structures with no space between members (even if it makes them impossible to access!)

-X105 Allow redefinition of #define symbols to the preprocessor.

-X143 Generate Weitek floating point code instead of 80387.

-X153 Enable ANSI C extensions. Some ANSI extensions to C have not been implemented yet.

-X164 Do not stop in the event of a code generator abort or "Internal Compiler Error" error message. 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.

-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.

-X211 Suppress optimizations that generate inline code for external calls.

-X214 Eliminate branches by copying code at branch destination in place of branch.

-X219 Suppress elimination of jumps to jumps.

-X226 This is the compiler switch to indicate an 80387 rather than an 80287.

-X229 Use a non-standard convention whereby 4 of the 80X87 registers are used as registers.

-X230 Suppress common subexpression elimination and value propogation, except for trivial cases.

-X233 Functions that return the type "float" return a single precision value, not a double precision value.

-X237 Apply associative rules in common subexpression elimination.

-X252 Pure Intel asm386.

-X255 Print a brief description of -X switches on the terminal.

-X264 Suppress phase that removes useless sign and zero extend instructions.

-X265 Suppress register database phase.

-X266 Repeat the peephole phase until the code fails to inprove.

-X268 Suppress the lastload phase.

-X271 Suppress the phase that merges and removes excess move instructions.

-X272 Suppress the realvar code in database phase.

-X278 Don't merge index calculation into load instruction.

-X285 Suppress block merge phase.

-X297 Always load NARGS before a call.

-X302 Align double precision variables on 8-byte boundaries, subject to certain limitations.

-X304 Truncate names to eight characters on input.

-X306 C "asm" inline directive not recognized.

-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.

-X325 Return large items in a reentrant fashion, rather than following old UNIX customs.

-X326 Allocate gettarget temporaries as a round robin instead of a stack.

-X329 Generate "stabd" pseudo-ops for line numbers instead of stabn line numbers.

-X331 Allocate unused variables if symbolic debugging enabled (-g).

-X332 Try to avoid generating floating divides if a multiply can be used instead.

-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.

-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.

-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 described in (some versions of) the proposed ANSI C standard, that is the compiler may not associate over them.

Green Hills Software

C-860 User's Guide

Version 1.8.5

C Compiler for the i860™ processor

January 1990

Green Hills Software, Inc.

DISCLAIMER

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

Green Hills Software, Inc.
510 Castillo St.
Santa Barbara, Ca. 93101
(805) 965-6044
Fax: 965-6343

Oasys Division (Sales & Support)
230 Second Ave.
Waltham, Ma. 02154
(617) 890-7889
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 gives you general ideas as to 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.

C Runtime Library

This section describes the C-860 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.

CHAPTER 2

The C Language

2.1. Introduction

C-860 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 our default definition of the C language.

C-860 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 below, but this is by no means a complete list.

Currently, the user documentation for C-860 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-860 User's Guide.

2.1.1. Preprocessor

C-860 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-860 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, the preprocessed output is sent directly to the next compilation step and no temporary output is retained.

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 identifiers without the underscore prefixes are not available in FullANSI mode.

__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 and __LittleEndian are predefined preprocessor symbols that reflect 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. __i860 is a predefined preprocessor symbol on Intel i860 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 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 FullANSI 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. Functions Returning Structures or Unions

For compatibility with the PCC implementation of C, returning a structure or union from a function is done in a non-reentrant fashion.

A structure or union return value is returned by copying the return value into a static variable in the function. A pointer to this static variable is then returned. The calling function then uses the returned pointer to copy the static variable. A problem occurs if an interrupt or signal occurs after a function returns but before the caller had time to copy the return value and the interrupt or signal handler calls the function which was interrupted. The function call in the interrupt routine modifies the static return variable that the interrupted routine is using. When the interrupted routine continues, it accesses the value of the static variable set in the interrupt level routine instead of the value it would have accessed had there been no interrupt or signal.

2.2.3. 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.

2.2.4. The VARARGS(3) Facility

C-860 supports the UNIX VARARGS(3) facility. The VARARGS(3) 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(3) facility:

(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(3) 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(3) 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);
}
```

2.2.5. Bit Fields

C-860 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.

2.2.6. 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-860.

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".

2.2.7. asm Statement

The asm statement (for inline assembly code) in C-860 is the same as in PCC. In C-860 the asm statement can be used anywhere a statement or declaration can appear (even between functions).

Since the code generated by C-860 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.

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)
-inline	Turn on inlining optimizations (-X249 et al)
-QI *name*	Inline specified routine
-ansi	Select ANSI mode (-X316)
-ANSI	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.

2.3.3. Preprocessing Directives

A number of new preprocessing directives are introduced in 1.8.5 to conform to the new ANSI standard.

#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 only in ANSI mode. 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 mode. Its value is a character string literal in the form "hh:mm:ss" containing the system time that the source was compiled.

__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 type qualifiers, **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 AND either -ansi or -ANSI is specified, -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

Intel i860 CPU Target

3.1. Introduction

This chapter describes the Intel i860 cpu target environment for C-860.

3.2. i860 Characteristics

The i860 memory is byte addressed with 32 bit addresses. Bytes are ordered with the least significant byte of a multiple byte value stored at the lowest address, as on the i386™ microprocessor, opposite of the IBM/370. Bits are numbered with bit zero as the least significant bit.

Floating point is IEEE 754 format (32 and 64 bits), least significant byte at the lowest address.

Character encoding is ASCII.

The stack is always sixteen byte aligned.

Bit fields are allocated starting at bit 0. 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
enum (-X6)	8,16,32	8,16,32

3.3. Compiler Output Format

The output of the compiler is i860 Assembly Language, either as described in the i860 Programmer's Reference Manual, or in the i860 Application Binary Interface.

The -g option generates Common Object File Format (COFF) ".def" symbolic debug pseudo-ops in the assembler language output. The assembler and linker (will) understand and process the symbolic debug entries in the object files. The "sdb" symbolic debugger could be used with C-860 output.

3.4. Register Usage

There are 32 general purpose registers used for integer values and another 32 floating point registers. Double precision floating point values are contained in two adjacent floating point registers.

Register	Use <in PRM>
r0	Always contains a zero value
r1	Return address to calling function
r2	Stack pointer
r3	Frame pointer
r4..r15	Permanent registers saved across calls
r16	Integer return value
r16	Pointer to structure/character string returned
r16..r27	Register arguments to called function
r29	Static link register (for pascal)
r28,r30,r31	Temporary registers not saved across calls
f0/f1	Always contains a zero value
f2..f15	Permanent registers saved across calls
f16	Single precision return value
f16..f17	Double precision return value
f16..f27	Register arguments to called function
f28..f31	Temporary registers not saved across calls

Register	Use <in ABI>
r0	Always contains a zero value
r1	Return address to calling function
r2	Stack pointer
r3	Frame pointer
r4..r15	Permanent registers saved across calls
r16	Integer return value
r16	Pointer to structure/character string returned
r16..r27	Register arguments to called function
r28	Pointer to start of memory arguments (if any)
r29	Static link register (for pascal)
r30,r31	Temporary registers not saved across calls
f0/f1	Always contains a zero value
f2..f7	Permanent registers saved across calls
f8	Single precision return value
f8/f9	Double precision return value
f8..f15	Register arguments to called function
f16..f31	Temporary registers not saved across calls

3.5. Calling Conventions

The compiler supports two calling conventions (selectable by a compile time switch), the first is that specified in the PRM, the second in the ABI.

In order for a function to be able to save registers effectively and access double precision arguments and local variables on the stack, a convention has been adopted that the stack must be located at a 16 byte boundary at the entry to a function. The first function or the Operating System must insure that the stack is aligned on a 16 byte boundary. If this is done the compiler will insure that it remains aligned.

3.5.1. PRM

The following is the calling convention set forth in the 860 Programmer's Reference Manual.

Scalar values are returned in r16, sign or zero extended to 32 bits for types smaller than 32 bits. Single precision floating point values are returned in f16, double precision floating point values are returned in the register pair f16/f17.

In function calls all large data types (structures, records, arrays, etc.) that are passed by value are passed on the stack. Any arguments of integral type (integers, short integers, addresses, etc.) among the first 12 arguments are passed in registers, and the remainder on the stack. And any floating point arguments among the first 6 are passed in registers and the remainder are passed on the stack. All floating arguments are assumed to take up 2 registers. If a routine is called with an integer, two single precision floats, and a double precision float then these arguments will be placed in r16, f18, f20 and f22/f23 respectively.

If there are any stack arguments then before the call of the function, an argument area is allocated on the stack large enough to hold all of the arguments (except those in registers). The size of this area is rounded up to a 16 byte boundary to preserve the stack at a 16 byte boundary. The area is allocated by subtracting from the stack pointer. After the function returns the argument area is removed from the stack by adding to the stack pointer. C-860 minimizes adds and subtracts to the stack pointer by coalescing them when possible.

3.5.2. ABI

The following is the calling convention set forth in the 860 Application Binary Interface.

Scalar values are returned in r16, sign or zero extended to 32 bits for types smaller than 32 bits. Single precision floating point values are returned in f8, double precision floating point values are returned in the register pair f8/f9.

In function calls all large data types (structures, records, arrays, etc.) that are passed by value are passed on the stack. Any arguments of integral type (integers, short integers, addresses, etc.) among the first 12 integral arguments are passed in a register, and the remainder on the stack. And any floating point argument that fits in the 8 floating parameter registers will be passed in them, and remainder are passed on the stack. Single precision arguments are packed in the registers (one register per argument), double precision arguments require two registers and must be aligned (thus gaps may be left). If a routine is called with an integer, two single precision floats, and a double precision float then these arguments will be placed in r16, f8, f9 and f10/f11 respectively.

If there are any stack arguments then before the call of the function, an argument area is allocated on the stack large enough to hold all of the arguments (except those in registers) and r28 is set to the address of the start of this area. The size of this area is rounded up to a 16 byte boundary to preserve the stack at a 16 byte boundary. The area is allocated by subtracting from the stack pointer. After the function returns the argument area is removed from the stack by adding to the stack pointer. C-860 minimizes adds and subtracts to the stack pointer by coalescing them when possible.

Arguments are evaluated from right to left. Each argument, unless it fits in an argument register(s), is stored in the argument area at the offset corresponding to its position in the argument list. If an argument requires 8 (or 16) byte alignment and the offset of the argument would not have been 8 (nor 16) byte aligned, then 4 (or 8 or 12) extra unused bytes are assumed to be left before the argument to align it on the correct boundary.

In ANSI C in a prototype procedure call, each argument is converted to the type of the formal parameter after it is evaluated (except that shorts and chars are promoted to integers). Otherwise each scalar argument is extended to a 32 bit value after it is evaluated. Each floating point argument is converted to a 64 bit double precision value after it is evaluated. All other values are extended to a multiple of 4 bytes after being evaluated (the extra bytes are at the high addresses).

When calling a function which returns a structure, the address of a temporary of the return type is passed in r16. The arguments are passed in r17-r27 and on the stack as if the temporary address had been the first argument. The function returns the structure value by copying the return value to the address pointed to by r16 on entry.

A call to a function either uses a call (or calli) instruction (which saves the return address in r1), or on A2-step chips a combination of loading the return address into r1 and executing a bri instruction (for an indirect call). The return from a function uses a "bri r1" instruction.

A function call is assumed to destroy r1, r16..r31 and f8..f31 (in PRM f16..f31). The special call 'alloca' is assumed to change r2 (the stack pointer). No other registers are destroyed by a function.

Accesses to parameters are made relative to r28 (or a copy of it) in the ABI and relative to either the frame pointer or the stack pointer in the PRM. Accesses to local stack storage are made relative to either the stack pointer or the frame pointer. A stack frame pointer is set up if the stack frame is large enough for it to be useful or if source level debugging is required.

CHAPTER 4

Optimization

4.1. Introduction

C-860 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-860 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.

4.2. General Optimizations

General Optimizations always make programs smaller and faster.

4.2.1. 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-860 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-860 is much better than most C compilers in its register allocation.

In the following example, C-860 allocates i and j to the same register because their lifetimes do not overlap.

```
proc()
{
        int i, j;

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

```
_proc:
        adds    -16,sp,sp
        st.l    r1,12(sp)
        st.l    r4,8(sp)
        or      1,r0,r4
.L5:
        call    _f
         adds   1,r4,r4
        or      10,r0,r30
        subs    r4,r30,r0
        bc      .L5
        or      1,r0,r4
.L9:
        call    _g
         adds   1,r4,r4
        or      10,r0,r30
        subs    r4,r30,r0
        bc      .L9
        ld.l    12(sp),r1
        ld.l    8(sp),r4
        bri     r1
         adds   16,sp,sp
//_i    r4      local
//_j    r4      local
```

The improvement by the C-860 optimizer can be summarized as:

Put i and j in r4	2 memory references per iteration
No frame pointer	3 instructions per call
Rotate Loop	
Instruction Reordering	1 instruction per call instruction

4.2.2. Memory Allocation

C-860 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".

4.2.3. 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-860 does not set up a frame pointer in each function. C-860 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.

For functions which require saving at least one floating point register the stack decrement for the entire stack frame can often be folded into a store (using a predecrement mode), saving one instruction.

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.

4.2.4. Static Address Elimination

A valuable optimization performed by C-860 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.

```
p0
{
   int i;
   static int x;

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

```
_p:
        adds    -16,sp,sp
        st.l    r4,12(sp)
        orh     h%_x,r0,r16
        or      l%_x,r16,r16
        or      8,r0,r18
        adds    -1,r0,r19
        bla     r19,r18,.L24
        or      1,r0,r17
.L24:
.L5:
        ld.l    0(r16),r28
        adds    r17,r28,r28
        adds    1,r17,r17
        bla     r19,r18,.L5
        st.l    r28,0(r16)
        ld.l    12(sp),r4
        bri     r1
        adds    16,sp,sp
//_i    r17     local
//_x    _x      import
```

The improvement by the C-860 optimizer can be summarized as:

Static Address Elimination	2 instructions per interation
No frame pointer	3 instructions
Conversion to bla (sob) loop	2 instructions per iteration
Instruction Reordering	2 instructions and 1 gap per iteration

4.2.5. 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-860 will use the destination register in the computation so as to save unnecessary copies of the source registers into scratch registers.

For example the C-860 compiler will compile the statement "i = i*100+j;" as follows (i is in r16, j in r17)

```
        shl     3,r16,r28
        subs    r16,r28,r16
        shl     2,r28,r28
        adds    r28,r16,r16
        shl     2,r16,r16
        adds    r16,r17,r16
```

The final add stores its result in r16, saving a move instruction. But the most obvious improvement is replacing the multiply by a series of shifts and adds, a potential savings of 4 cycles.

4.2.6. Passing Parameters in Registers

In C-860, 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 r16 and y is in r17)

Register Parameters		Parameters on the Stack	
		adds	-32,sp,sp
		adds	-32,sp,sp
		adds	r16,r17,r16
call	_f	call	_f
adds	r16,r17,r16	st.l	r16,12(sp)
		adds	16,sp,sp
call	_g	call	_g
nop		st.l	r16,12(sp)
		adds	16,sp,sp

4.2.7. 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.

After the compiler has rotated the loop it will often find that it can now use a 'bla' (subtract one and branch) instruction, which in the best case will eliminate an increment and compare instruction from the loop (and also make it more likely that the delay slot of the branch will be filled). If this instruction is not useful for a given loop, the compiler will still detect the presence of a loop and use the delayed forms of 'bc' and 'bnc'.

4.2.8. 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.

4.3. 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-860 that most computation is performed in inner loops. When this compile time option is specified, C-860 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.

4.3.1. In Line Multiplication and Division

Since the instructions to do an integer multiplication potentially take 9 cycles to execute, it is often faster to do constant multiplies by a series of shifts and adds (or subtracts). For instance a multiply by 4 is a shift left by 2, multiplying by 5 is a shift left by 2 followed by an add, and 7 is a shift left by 3 followed by a subtract. For an example of this see the code under Register Coalescing.

Integer division is much worse, it takes about 60 cycles to do a divide. When dividing by a constant the compiler can calculate a (floating point) reciprocal at compile time and convert the divide into a floating multiply which only takes about 15 cycles. In certain bizarre cases when using 16 bit integers the compiler can do a divide in an integer multiply and a shift.

Floating point division can often be speeded up by calculating a reciprocal either in the compiler (if division is by a constant), or at the head of a loop if the divisor is a loop invariant.

4.3.2. 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.

4.3.3. 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.

4.4. 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-860 simulates the timing of 80860 instruction sequences. When C-860 detects that an instruction sequence will result in a pipeline delay, C-860 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.

Note this is not the type of pipelining that makes use of the 860's pipelined instructions.

CHAPTER 5

Porting Programs to C-860

Some programs which appear to compile and operate correctly when compiled with other C compilers, may not operate correctly when compiled with C-860. The C Language specifications define legal programs in such a way that legal programs will always work with all C compilers, including C-860. The problem is that many programmers make illegal assumptions about the machine or compiler that they are using. This chapter discusses many illegal assumptions which can cause programs to fail when compiled with C-860.

5.1. 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-860 program.

The implementation of each Green Hills C Compiler is the same for each Green Hills Target. Therefore, legal programs written in C-860 can be moved to any other Green Hills C Compiler.

C-860 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.

5.2. 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.

5.3. 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 decendents of the PDP-11, such as the VAX, 8086/88/286/386, NS32000, and Clipper have followed the DEC convention. These two

groups seem to be so well entrenched that no agreement on byte ordering is possible.

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.

5.4. Alignment Requirements

C-860 always aligns multiple byte data items on appropriate address multiples so that all accesses will be legal and efficient. The alignment conventions for C-860 are defined in the 80860 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 80860, all global data references will be optimal and legal.

C-860 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 80860 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 overlays. It will also lead to problems with programs which make implicit assumptions about the size and offset of objects.

5.5. 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-860 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" appear 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 depence on the character collating sequence. If a program is designed to operate with a collating sequence

other that ASCII it may be necessary to modify string and character comparison code to operate with ASCII.

5.6. 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.

5.7. 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-860. Refer to your target operating system documentation for a description of legal file names for your environment.

5.8. 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.

5.9. 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-860 evaluation order is not 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-860.

Some implementations of the C Language evaluate the arguments to a function from right to left, others from left to right. See the 80860 Target chapter for details of the C-860 calling conventions.

Expressions with side effects, such as function calls and the operators "++", "--", "+=", etc., may be executed in a different order by C-860 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-860 may allocate some pointer variables not declared "register" to registers. This may allow C-860 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-860.

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-860 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-860.

5.10. C Preprocessor Incompatibilities

The C Preprocessor that is provided with PCC has many undocumented features. Most of these undocumented features are implemented in C-860.

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-860 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-860 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 the compiler can construct an identifier as:

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

Which becomes (in both PCC and C-860)

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

5.11. 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-860, 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.

5.11.1. Problems with Setjmp and Longjmp

Under the default configuration of C-860, 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 contents of the "label" variable and continues executing after the call to setjmp. Under the portable C compiler 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 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 (dbnamemem) 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.

5.11.2. 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.

5.11.3. Memory Allocation Assumptions

Memory is allocated by C-860 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-860 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-860 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-860 will allocate certain variables to registers that PCC and other compilers would always allocate to memory. Programs compiled with C-860 must not make assumptions regarding the order of allocation of variables in memory (except where the C language standard specifies it).

5.11.4. -OM Restrictions

The -OM and -OLM compile time options should only be used in algorithmic programs, that is, 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 mapped 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, most 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

5.11.5. Problems with Source Level Debuggers

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.

5.12. Problems with Compiler Memory Size

C-860 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-860 requires 500 Kbytes just for the program. It is designed to work best when it has at 1 Mbyte 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 into 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.

C-860 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 80860 code generator. The code generator produces an internal representation of the 80860 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. Simple functions 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 more than a megabyte of memory to compile.

5.13. Detection of Portability Problems

Many of the problems associated with porting programs to C-860 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.

CHAPTER 6

Compile Time Options

Each C-860 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 compiler invocation documentation that you received with C-860. It should also document all options that are supported by your implementation.

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 (Unix only) 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.

-ansi Recognized as either -ansi or -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. If -ANSI is selected, the compiler is placed in FullANSI mode. FullANSI mode is 100% compliant with the ANSI X3J11 standard and disallows any non-compliant constructs. -X153 is equivalent to -ANSI. '-X316 -D__STDC__' is equivalent to -ansi alone.

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

-inline a separate compilation with -X244 -X247 -X251 -X345 and -QR *file*.l for each file. The -inline option is currently recognized only in lowercase. This option turns on the automatic inlining optimizations. A library archive file containing inline information is created, with the extension ".l", when -inline is selected. A temporary assembly language file, a.s is created during compilation, rather than the standard *file*.s.

Routines from separate source files can be inlined because the inlining database stores each routine in a library (.l) file and retrieves it as required.

-inline=*routinename1,routinename2,...*
Routines that are to be inlined, in addition to those determined by the compiler or specified within the source code, can be specified with -inline=*routine1,routine2,....* This is given to the compiler as a set of -QI *routinename* arguments.

-noauto The compiler can be directed to not make any inlining decisions (i.e., inline only those routines specified by -QI options or in the source code) with -noauto, which corresponds to not specifying -X251 on the final compilation.

-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.

-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 (C only) 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.

-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
(UNIX Host only) 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 (or -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.

-QI *routine*
The QI option directs the compiler to inline the subroutine name *routine*.

-s Used in conjunction with -ansi, -s places the compiler in FullANSI mode, providing 100% compatibility with the ANSI X3J11 standard and disallowing any non-compliant constructs.

-S (UNIX Host only) 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".

-Uname Undefine the predefined preprocessor symbol "name". This is equivalent to putting "#undef name" at the top of the source file.

-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.

-Xnnn Where nnn is an unsigned integer constant. Turn on compile time option number nnn. The available compile time options are listed below.

-Znnn Where nnn is an unsigned integer constant. Turn off option number nnn. 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", "short", "int", "unsigned char", "unsigned short", 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.

-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 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 fields of type int, short, and char be signed. The default is for all fields to be unsigned.

-X57 Generate bounds checking code for subranges and arrays.

-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-860.

-X84 Generate error messages for C anachronisms. By default the old assignment operators (=+ =- ...), initialization (int i 1), and references to members of other structures compile correctly but generate warning messages.

-X85 (UNIX Target only) 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 (UNIX Target only) Target is UNIX BSD 4.2

-X115 (UNIX Target only) Target is UNIX BSD 4.1

-X151 Do not allow dollar signs in names. The default allows dollar signs for VMS compatibility.

-X153 Place the compiler in FullANSI mode. This is set automatically when -ansi -s is specified. If this option is used in conjunction with -O, -OM is automatically selected too.

-X164 Do not stop in the event of a code generator abort or "Internal Compiler Error" error message. 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.

-X211 Suppress optimizations that generate inline code for external calls.

-X219 Suppress elimination of jumps to jumps.

-X230 Suppress common subexpression elimination and value propagation, except for trivial cases.

-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.

-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.

-X326 Allocate gettarget temporaries as a round robin instead of a stack.

-X329 Generate "stabd" pseudo-ops for line numbers instead of stabn line numbers.

-X331 Allocate unused variables if symbolic debugging enabled (-g).

-X332 Try to avoid generating floating divides if a multiply can be used instead.

-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.

-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.

-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.

-X393 Align large standalone data items on 16 byte boundaries. (see -X422)

-X394 Don't eliminate simple divides, such as those used by Drystone during common subexpression elimination.

-X401 Enable CEXTERNAL declaration.

-X402 Append two underscores instead of a dollar-sign to the external name of COMMONs in VMS compatibility.

-X403 Accept noalias keyword in C.

-X405 Type 'char' is unsigned type.

-X407 Turn on IEEE machine independent software floating point.

-X410 Modify front end to parse i860 preprocessor commands.

-X412 Force a call to a precise divide routine.

-X414 Use 96-bit objects for long doubles.

-X415 Allow C++ style references.

-X416 Allow functional casts (ie. double(3)) struct n.

-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.

-X425 Generate code fixes for known A-step chip bugs, do not use the indirect call instruction and assume that 8 and 16 bit loads are unsigned.

-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 No special tricks 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 Don't propagate constants in their own pass.

-X469 Don't recognize sin, cos, etc. as pure functions.

-X470 Suppress tail recursion.

-X474 Suppress Common Subexpression Elimination (CSE).

-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.

-X490 Use following string for .file directive.

-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.

-Z501 Assume that 8 and 16 bit loads are unsigned, and do not use the indirect call instruction (ie. assume a perfect A step part).

-X502 Generate code fixes for B0-step chip bugs

-X505 Enable new loop optimizer.

-X509 Complain about locals read before written.

-X510 Suppress .stab 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. Only variables of scalar types (real, integral or pointer) are checked. Fields of records or structs (or classes) which are scalar are checked (this is done by changing the record by adding a shadow field corresponding to every checked field, the compiler assumes that these shadow fields will be initialized to 0 (false). For automatic (stack based) records this is not the case and some uninitialized fields may go unreported. Similarly when allocating memory, an allocator should be used that zeros memory (a pascal new will do this). Members of arrays are not checked. Things pointed to by pointers are not checked. The act of taking the address of a variable is assumed to initialize it. In C external and static variables which are not explicitly initialized are treated as being uninitialized (even though the language guarantees that they will be 0). Overlaying two different record structures (by casting pointers, or unions or variant records) will probably fail.

If this flag is used it should be applied to the entire program (since it changes the layout of records, and loses track of variables set in other files).

-X599 Compress structures by removing all internal alignment requirements (so all types, except bit fields, are aligned on byte boundaries).

-X611 Output make dependencies on standard output rather than compiling the program. This is a list of include files opened in a convenient format.

-X614 Put the input source lines into the assembler output file. Lines from include files are not copied, macro substitution will not appear.

-X615 Use the calling conventions specified in the PRM.

-X616 Use the calling conventions specified in the ABI.

-X617 Use the assembler syntax specified in the ABI.

-X618 Align doubles in structures as they would be on the i386 processor (on 4 byte boundaries rather than 8 byte).

CHAPTER 7

Runtime Error Messages

The following table lists error messages generated when the compiler inserts debugging checks into the output program. The switches that control the compiler precede each message.

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

CHAPTER 8

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
Asm statement illegal under object code option
Bit field not legal as operand of sizeof
Cannot open file:
Cannot take the address of this object
Case expression not constant
Case not in switch statement
Character constant too long
Constant expected
Could not disambiguate overloaded procedure name:
Declaration not legal here
Duplicate case
Duplicate field:
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 initial value
Illegal octal digit

Illegal operation
Illegal option:
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: Can not save nested routines.
Inliner: Cannot process ENTRY statements.
Integer expression required
Internal Compiler Error
Invalid type coercion
Label Expected
Label not defined:
Multiple definition of identifier:
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 a parameter name:
Not enough arguments given
Not inside a loop
Not inside a loop or switch
Nothing declared in a declaration
Null dimension
Only one
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
Redeclaration of:
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 error message reserved for the inliner.
This error message reserved for the inliner.
This error message reserved for the inliner.
This error message reserved for the inliner.
This is a binary file
This is not an lvalue
This object has no defined size
This use of an incomplete type is illegal
This warning message reserved for the inliner.
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 declarations of:
Two storage classes specified
Type mismatch
Type size exceeds implementation limit
Undefined member:
Undefined symbol:
Unexpected end of file
Unknown size for parameter
Unmatched #endif
Variable expected
Void type argument illegal
Void type for
Warning, Inline call parameter mismatch for:
Warning, can not inline static routine:
Warning, inliner saved long name:
Warning: Inline routine has complex inits of non-local variables, expansion suprressed.
Warning: this compiler does not have any inlining capability.
const ignored
illegal function
illegal member use:
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: Illegal member use:
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: 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
warning: redefinition of:

Green Hills Software User's Manual

C LIBRARY

September 1, 1986

Green Hills Software, Inc.

Green Hills Software, Inc.
425 E. Colorado, Suite 710
Glendale, CA 91205
(818)246-5555

Table of Contents

CHAPTER 1

Customizing and Compiling the Green Hills Software C Library

This chapter explains how to customize and compile the Green Hills Software C Library source files. If you received the Green Hills Software C Library already compiled then you can ignore this chapter and begin using it immediately.

The material in this chapter assumes that you are familiar with The Green Hills Software User's Manual for your version of C.

1.1. Compile Time Options

On the compilation command line for each module you can specify certain definitions that will affect the functionality of the resulting object code library. The definitions are specified with the -D command line definitions of C (if you don't know what -D is, you are in over your head already). The available compile time options are given below. Each of these options affects only a few files, but you should specify the same options for all files. If you specify different options for different files you may create an incorrect library.

-Dno_float

If you do not intend to do formatted floating point input or output with printf or scanf you may specify -Dno_float. If you specify -Dno_float, printf and scanf will ignore the floating point format characters e, f, and g. The result of specifying floating point arguments to printf and scanf will be quite confusing (and definitely wrong).

The reason that this option is provided is that many embedded programs do not use floating point, but they need printf for diagnostic messages. printf and scanf learn the types of their arguments only at runtime by examining the format character string argument. The linker cannot know whether any floating point arguments will ever be given to printf and scanf. Therefore, if printf and scanf are compiled with the default they will load in a huge amount of floating point library code in anticipation of having to do formatted floating point I/O, even if the program does not use any floating point at all. Specifying -Dno_float will suppress the loading of floating point code solely for the purpose of implementing formatted I/O.

-DUN_BUFFERED_STD

By default, the stdin and stdout streams are line buffered for compatibility with UNIX. This buffering takes 1024 bytes of buffer space. In many embedded systems this overhead is intolerable. If -DUN_BUFFERED_STD is specified then stdin and stdout will not be buffered. Output will occur immediately after a putchar() is executed instead of waiting until the end of the line and getchar() will return as soon as a character is typed instead of waiting for an end of line to be typed.

-Dprompts — If -Dprompts is specified when compiling the library args() will prompt the user on standard output with ">". At this point the user can type in a UNIX style command. The UNIX redirection operator "<" can be used to open stdin. The UNIX redirection operator ">" can be used to open stdout.

-Ddo_roman — If you want to be able to input and output Roman Numerals in printf and scanf specify -Ddo_roman when compiling the library.

-Dvaxd_float — If your floating point format is the standard Green Hills MC68000 VAX Floating Point Format Software Emulation you may specify -Dvaxd_float when compiling the library. It will improve the floating point library performance.

-Dieee_float — If your Target uses IEEE floating point numbers you may specify -Dieee_float when compiling the library. It will improve the floating point library performance.

If -Dieee_float is specified you must examine the math.h include file. At the top of math.h are a number of #ifdef's for various computer architectures. If the type of your target machine is specified in the #ifdef's then there is nothing more for you to do.

If your computer architecture is not specified in the #ifdef's you must find out whether your processor conforms to the LittleEndian or BigEndian format. LittleEndian must be defined if your target machine stores the exponent part of an IEEE floating point number in memory in the first two bytes of the representation. If the machine is neither BigEndian or LittleEndian you may not use -Dieee_float.

1.2. Customizing the Low Level System Interface

The module "os.c" is a simple implementation of the low level system interface for the Green Hills Software C Library. As it was written it is only minimally useful. Since every system has different memory addresses and I/O devices you must customize these low level functions to work with your system in order to be able to use the Green Hills Software C library.

The functions as written in "os.c" will execute unchanged in most environments with limited functionality. You will be unable to do any output until you modify OUTPUT() to read a character from some terminal or other I/O device. You will be unable to do any input until you modify INPUT() to read a character from some terminal or other I/O device. You will be unable to gracefully terminate execution until you modify _exit() to return control to you. You will be unable to access command line arguments in your main program until you modify args() to find them.

But most important of all you will not be able to do anything at all until you figure out how to execute the START() function in "crt.0.c"! This is the most system dependent part of the library. START() may be called by the operating system, a monitor, a debugger, or on system reset. YOU must figure out how to execute the START() function. In your environment it may be automatic. If not, you will have to inform the Link Editor (Linker) that program execution is to start at the top of the START() function.

In addition, you may have to modify the START() function for your particular environment. START() is responsible for initializing the processor, stack pointer, I/O, and memory systems. Then it calls the program's main() function with the command line arguments. If the main() functions returns to START(), START() exits by calling exit().

As written, START() is a C function. It may be possible to leave START() written in C (possibly using "asm" statements), but more than likely you will find it necessary to compile START() into assembly language and then edit the assembly code so that it can be used. In particular, there may be procedure entry code at the top of START() to save registers. This code may depend on a stack pointer (if it hasn't already been initialized), you may need to delete the function entry code.

The implementations of the system call interface in the "os.c" module is very simple. It provides only simple formatted I/O for a single terminal and dynamic allocation of memory from a static array. It assumes the following:

(a) There is a way of entering the program at the function START().

(b) There is a way to exit the program and return control to the operating system, monitor, or debugger. This is done by calling exit(code). To exit cleanly from programs you must modify the _exit() routine in this module. As it is written, it will terminate with a divide by zero exception. If your execution environment does not stop on a divide by zero you must modify _exit() or the program will never stop and terrible things will certainly happen!

(c) There is one input device of interest. Characters can be read from it by calling the function INPUT() in this module. It returns a value 0...225 or EOF for error. As it is written it always returns EOF (end of file). To do any input at all you must modify this routine!

(d) There is one output device of interest. Characters can be written to it by calling OUTPUT(ch). Its return will be ignored. As it is written it doesn't output anything. To do any output at all you must modify this routine.

(e) 10000 bytes is the maximum amount of dynamically allocated memory that will be needed. This 10000 bytes is permanently allocated in a static array in "os.c". If this size is too large or too small you should change the constant MEMSIZE in "os.c" to the correct amount of dynamically allocated memory.

After implementing items (a) through (e) above, you have the functionality needed in an embedded system, simple printf and scanf to a terminal and memory allocation from a fixed pool. No file I/O is supported.

If you are ambitious and you want to use all of the functions of the Green Hills Software C library you must fully implement the functions outlined in "os.c". Each function is a standard UNIX system call. A complete implementation of the UNIX system call is not necessary. The functionality that is necessary is specified with each function definition. Some hints and suggested implementations are given in comments in "os.c".

1.3. Customizing Include Files

The include files should operated perfectly well as you have received them, but you may wish to tune them to your system. The include files should be in a directory named "include".

BUFSIZ — The constant BUFSIZ in stdio.h is the buffersize used in buffered I/O. The smaller BUFSIZ is, the less memory will be used in I/O buffers (one for each open file). The larger BUFSIZ, the fewer I/O operati0ons must be done to disk (or any other I/O device).

BLOCKSIZE — The constant BLOCKSIZE in stdio.h is the block size which malloc uses to call sbrk() to get more memory from the operating system. The larger BLOCKSIZE is, the fewer times sbrk() will be called. The smaller BLOCKSIZE is the less memory will be wasted at the greatest extent of the program.

_NFILE — The constant _NFILE is stdio.h is the maximum number of files that can be open at one time (this number includes stdin, stdout, and stderr). It is also possible that the actual limit is less, if the underlying operating system or the "os.c" module has a lower limit on the number of simultaneously open files.

1.3.1. Custom Floating Point Formats

You do not need to read this section if your system has IEEE floating point, the Green Hills Software MC68000 VAX Floating Point Format Software Emulation, or no floating point at all.

If your system has a non-standard floating point format then you may either settle for the default floating point configuration which will work with any floating point format fitting the constraints given below in the Implicit Library Assumptions section, or you may customize the math.h include file, be prepared for some tricky debugging.

The performance of the floating point functions can be greatly improved if you CORRECTLY define the following macros in the include file math.h. It should be located in a directory called "include".

KNOWN_FLOATING_POINT_FORMAT
: Announces to the library that you have CORRECTLY defined the following macros, so the library can use them at will.

EXTRACTEXPONENT(x)
: Return the exponent field of the double, x, such that for some y:
0.5 <= y && y < 1.0 && x == y*2**(EXTRACTEXPONENT(x))

EXTRACTFLOATEXPONENT(x)
: Return the exponent field of the float, x, such that for some y:
0.5 <= y && y < 1.0 && x == y*2**(EXTRACTFLOATEXPONENT(x))

SETEXPONENT(x, exp)
: For double x: x = x*2**(exp - (EXTRACTEXPONENT(x))

SETFLOATEXPONENT(x, exp)
: For float x: x = x*2**(exp - (EXTRACTFLOATEXPONENT(x))

AUGMENTEXPONENT(x, inc)
: For double x: x = x*2**inc

AUGMENTFLOATEXPONENT(x, inc)
: For float x: x = x*2**inc

MINIMUMEXPONENT
: The minimum possible value for EXTRACTEXPONENT

MINIMUMFLOATEXPONENT
: The minimum possible value for EXTRACTFLOATEXPONENT

MAXIMUMEXPONENT
: The maximum possible value for EXTRACTEXPONENT

MAXIMUMFLOATEXPONENT
: The maximum possible value for EXTRACTFLOATEXPONENT

HUGE
: A number of very close to the maximum legal value for a double floating point number. A little less than 2**MAXIMUMEXPONENT.

HUGEFLOAT
: A number very close to the maximum legal value for a float floating point number. A little less than 2**MAXIMUMFLOATEXPONENT.

1.4. Installing the Green Hills Software C Library

When you have made all of the customizations required for your system, compile every module of the library except "test-c.c". For each compilation specify the compile time options that you selected from the "Compile Time Options" section above. Be sure to specify the same compile time options on every file. Do not compile the file "test-c.c", it is a test program and should not be included in the library.

When all of the files have been compiled, they must be collected together into a library. The exact means for creating a library varies from system to system. You should consult your librarian or linker documentation to find out how to make a library.

If you cannot figure out how to make a library, you can just leave all of the object files for the library in a directory. Every time you link a program, link in all of the C library object files. This will produce a very large program. If this is not acceptable, you can delete any library files that you do not use. In this way you can get the same effect as if you had a librarian.

1.5. Green Hills Software C Library Test Program

When the library has been built you are ready to compile and run the Green Hills Software C Library Test Program. This Test Program is designed to test the Green Hills Software C Library. The source for the Green Hills Software C Library Test Program is the file "test-c.c".

Before you compile "test-c.c" you may have to make a few adjustments. There are three compile time options that can be specified when compiling the Test Program.

-Dtest_stdin Test input from stdin. By default only output is tested.

-Dtest_files Test I/O on files. By default only terminal I/O is tested.

-DTempFilename=file-name
If you specify -Dtest_files you must also define the symbol TempFilename to be the name of a file that can be created and written to. If TempFilename is not specified it defaults to FOO.BAR.

When the Test Program is run, the first thing that should happen is that it should print the following line on your terminal:

```
Begin test
```

It does this by calling write with the single character 'B' and specifying file number 1 (standard output); followed by a call to "write" with the single character 'e' specifying file number 2 (standard error); next it calls write to the standard output with the four character string "gin"; finally it calls write to standard error with the five string character set "test\n". If this line does not come out correctly you have not implemented the "write" function correctly.

The second line of output should be:

```
C library test version 1.0 (test putchar,fflush)
```

This line is output by a series of putchar calls followed by fflush(stdout). If this line doesn't come out you probably have problems with your include files. Make absolutely sure that the "stdio.h" file in the include directory supplied with this library is being included and not some other "stdio.h" file from another directory (such as /usr/include/stdio.h).

The third line of output should be:

```
Basic eprintf test
```

This line is output by a call to eprint with a single string argument. If this line doesn't come out there is a serious problem with your compiler.

The fourth line of output should be:

```
More advanced eprintf test(all error msgs are produced with
eprintf)
```

This line is output be a call to eprintf. If this line does not come out you probably have a problem with the <varargs.h> file. It is absolutely essential that you have a correctly working VARARGS(3) facility in order to compile the printf and scanf modules. Make sure that the "vargargs.h" file that is being included in these compilation is the correct one for the Green Hills Software User's Manual for your version of C.

The fifth line of output should be:

```
Basic printf test
```

This line is output by a call to printf with a single string argument. If this line doesn't come out there is something wrong with your file number 1 (standard output).

The next three lines of output print out the argc, argv, and environment arguments that are passed to main(). The default is:

```
argc=1
argv={""}
env={}
```

If you have implemented a more complex command line argument scheme, all of the information passed to main will be printed out.

If there are no errors detected, no more output will be produced. Whenever an error is detected a message is printed out on standard error.

This test program is not exhaustive, but may prove helpful.

1.6. Implicit Library Assumptions

This C Library implementation makes certain assumptions about its environment. If these assumptions are invalid this C Library cannot be expected to work without modifications.

If your machine does not conform to these constraints you will probably have some problems with the library. We are interested in hearing about (and trying to fix) any problems you may have in running this library on other types of machines.

This C Library has been implemented to work with the Green Hills Software C compilers. It is not intended that anything in the library would prohibit it from working with other C compilers, but it has never been tested with any other compiler, and under Green Hills Software licensing policies it is only licensed to be used with Green Hills Software compilers. The only Green Hills Software specific aspect of the library is that the "register" declaration has been used less often than is normal in C programs. This is because the Green Hills Software compiler allocates variables to registers even if "register" is not specified. It is recommended that "register" not be used so that the compiler has the greatest freedom to optimize the code. But old habits die hard and there are still "register" declarations in the code.

The Green Hills Software C Library implicitly assumes several things about the machine it is running on. It assumes that it is a byte addressed machine. That is, there are 8 bits in a char, and sizeof(char) equals 1. It assumes there are 16 bits in a short, and sizeof(short) equals 2. It assumes there are 32 bits in a long, and sizeof(long) equals 4. It should work if sizeof(int)==sizeof(short) or if sizeof(int)==sizeof(long). However, it has never been tested when sizeof(int)==sizeof(short). The library works on both a Little-Endian architecture (least significant byte of a multiple byte integer is stored at the lowest address) and a Big-Endian architecture (most significant byte of a multiple byte integer is stored at the lowest address).

The library assumes two's complement integer arithmetic.

The library assumes (but only in a few places, such as ctype.c) that the character set is ascii.

The library assumes that sizeof(float)==4 and sizeof(double)==8. It assumes that floating point exponents are radix 2 (as opposed to radix 10). It assumes that exponents are not less than 8 bits long and not greater than 11 bits long. It assumes that there is at most 56 bits of precision in double precision floating point, and at most 24 bits of precision in single precision floating point. The library also assumes that if x !=0.0 then 1/x will not overflow. IEEE floating point with gradual underflow and trap on overflow will fail this assumption!

CHAPTER 2

C Library Function Groups

This chapter is a list of the functions and variables implemented by the Green Hills Software C Library organized by function groups. The C Library source file name is given in parentheses at the end of the line.

Math Functions

abs:	Integer absolute value (abs.c)
cabs:	Complex absolute value (cabs.c)
fabs:	Floating point absolute value (fabs.c)
ffs:	Find first set bit (ffs.c)
frexp:	Floating point return exponent and mantissa (frexp.c)
labs:	Long absolute value (labs.c)
ldexp:	Load excponent (ldexp.c)
modf:	Return integer and fractional part of floating point (modf.c)

Allocation

calloc:	C allocate memory (calloc.c)
cfree:	C free memory (cfree.c)
free:	Free memory (malloc.c)
malloc:	Memory allocate (malloc.c)
realloc:	Reallocate memory (realloc.c)

Formatted I/O

atof:	Ascii to floating conversion (atof.c)
atoi:	Ascii to integer conversion (atoi.c)
atol:	Ascii to long conversion (atol.c)
ecvt:	E format conversion (ecvt.c)
eprintf:	Print formatted output to standard error (printf.c)
fcvt:	F format conversion (ecvt.c)
fprintf:	Print formatted output to a FILE (printf.c)
fscanf:	Read formatted input from a FILE (scanf.c)
gcvt:	G Format conversion (gcvt.c)
perror:	Print error message (perror.c)

printf:	Print formatted output to standard output (printf.c)
scanf:	Read formatted input from standard input (scanf.c)
sprintf:	Print formatted output to a string (printf.c)
sscanf:	Read formatted input from a string (scanf.c)
strtol:	String to long conversion (strtol.c)
sys_errlist:	System error list (syserrlst.c)

File I/O

_cleanup:	Clean up (flush) all file buffers (cleanup.c)
clearerr:	Clear error indicator on a FILE (<stdio.h>)
EOF:	End of FILE character value (<stdio.h>)
exit:	Close all files and exit from program (exit.c)
fclose:	FILE Close (fclose.c)
fdopen:	FILE open with descriptor (fopen.c)
feof:	End of FILE query (<stdio.h>)
ferror:	Error on FILE query (<stdio.h>)
fflush:	FILE flush buffer (fflush.c)
fgetc:	FILE get character (fgetc.c)
fgets:	FILE get string (fgets.c)
_filbuf:	Fill FILE input buffer (filbuf.c)
FILE:	FILE type definition (<stdio.h>)
fileno:	I/O channel number associated with a FILE (<stdio.h>)
_flsbuf:	Flush FILE output buffer (flsbuf.c)
fopen:	FILE open (fopen.c)
fputc:	FILE put character (fputc.c)
fputs:	FILE put string (fputs.c)
fread:	FILE read (fread.c)
freopen:	FILE re-open (fopen.c)
fseek:	FILE seek (fseek.c)
ftell:	FILE tell position (ftell.c)
fwrite:	FILE write (fwrite.c)
getc:	Get a character from file (<stdio.c>)
getchar:	Get character from standard input FILE (<stdio.h>)
getchar:	Get character from standard input FILE (getchar.c)
getl:	Get long from FILE (getl.c)
gets:	Get string from FILE (gets.c)
getw:	Get int from FILE (getw.c)
NULL:	Null FILE and pointer value (<stdio.h>)
putc:	Output a character to a FILE (<stdio.h>)
putchar:	Output a character to a standard output FILE (<stdio.h>)
putchar:	Output a character to a standard output FILE (putchar.c)
putl:	Output a long to a FILE (putl.c)
puts:	Output a string to standard output FILE (puts.c)
putw:	Output a word (int) to a FILE (putw.c)
rewind:	Seek to beginning of a FILE (rewind.c)
setbuf:	Set a user I/O buffer for a FILE (setbuf.c)

setlinebuf: Set line buffering on a FILE (setlinebuf.c)
stderr: Standard error file (<stdio.h>)
stdin: Standard input file (<stdio.h>)
stdout: Standard output file (<stdio.h>)
ungetc: Put a character back on a FILE (ungetc.c)

Character Types

ctype: Character type array (ctype.c)
isalnum: Is alphanumeric (<ctype.h>)
isalpha: Is alphabetic character (<ctype.h>)
isascii: Is ascii character (<ctype.h>)
iscntl: Is control character (<ctype.h>)
iscsym: Is C identifier character (<ctype.h>)
iscsymf: Is C identifier initial character (<ctype.h>)
isdigit: Is decimal digit (<ctype.h>)
isgraph: Is a graphics character (<ctype.h>)
islower: Is lower case ascii character (<ctype.h>)
isprint: Is printable character (<ctype.h>)
ispunct: Is punctuation character (<ctype.h>)
isspace: Is white space character (<ctype.h>)
isupper: Is upper case ascii character (<ctype.h>)
isxdigit: Is hexadecimal digit (<ctype.h>)
toascii: Convert character to ascii (<ctype.h>)
tolower: Convert character to ascii (<ctype.h>)
_tolower: Convert character to lower case (<ctype.h>)
toupper: Convert character to upper case (<ctype.h>)
_toupper: Convert character to upper case (<ctype.h>)

String Functions

index: Return the index of a character in a string (index.c)
rindex: Reverse search for character in string (strrchr.c)
strcat: String concatenate (strcat.c)
strchr: Return the position of a character in a string (strchr.c)
strcmp: String Compare (strcmp.c)
strcpy: String Copy (strcpy.c)
strindex: String index (strindex.c)
strlen: String length (strlen.c)
strncat: String concatenate with maximum length (strncat.c)
strncmp: String compare with maximum length (strncmp.c)
strncpy: String copy with maximum length (strncpy.c)
strrchr: Reverse search for character in string (strrchr.c)
strrindex: Reverse search for string in string (strrindex.c)
strsave: String save (strsave.c)

Byte Strings

bcmp:	Byte string compare (bcmp.c)
bcopy:	Byte string copy (bcopy.c)
bufcpy:	Buffer copy (bufcpy.c)
bzero:	Byte string zero (bzero.c)
filln:	Fill n bytes of memory (filln).c)
clearn:	Clear n bytes of memory (clearn.c)
swab:	Swap byte string (swab.c)

Miscellaneous

args:	Return argument list, argument count, and environment (os.c)
errno:	Error number (errno.c)
Init_IO	Inititalize I/O (os.c)
INPUT:	Input a character from the controlling terminal (os.c)
mktemp:	Make temporary name (mktemp.c)
OUTPUT:	Output a character to the controlling terminal (os.c)
rand:	Return a random integer (rand.c)
srand:	Initialize random number generator used in rand() (rand.c)
START:	Program transfer address (crt0.c)

UNIX System Call Functions

access:	Check accessibility of file name (os.c)
close:	Close an I/O channel (os.c)
creat:	Create file and open an I/O channel to it (os.c)
_exit:	Exit program (os.c)
fstat:	Get device and file number connected to I/O channel (os.c)
ftruncate:	Truncate file connected to I/O channel (os.c)
getpid:	Get Process ID (os.c)
isatty:	Is a tty connected to I/O channel (os.c)
lseek:	Long seek on an I/O channel (os.c)
open:	Open an I/O channel (os.c)
read:	Read from an I/O channel (os.c)
sbrk:	Set memory address break (os.c)
stat:	Get device and file number of file name (os.c)
unlink:	Unlink (Delete) file (os.c)
write:	Write to an I/O channel (os.c)

CHAPTER 3

Alphabetized List of C Library Functions

This chapter is an alphabetized list of the functions, macros, and variables exported by the Green Hills Software C Library. The source file containing the definition is given with each function in parentheses.

abs: Integer absolute value (abs.c)

```
int abs(x)
int x;
/*****************************************************************/
/* Return the absolute value of "x".
/*****************************************************************/
```

access: Check accessibility of file name (os.c)

```
access(name, prot)
char *name;
/*****************************************************************/
/* The Green Hills Software C Library does not use the access function, it is only
/* needed if the Green Hills Software Fortran Library is being used.
/*
/* "name" is the name of a file, "prot" is the logical or of the following flags
/*                  4: readable
/*                  2: writable
/*                  1: executable or directory searchable (Unused)
/* If "prot" is zero then test only for file existence.
/* Return 0 if all of the specified operations are possible on the file.
/* Return -1 if any of the specified operations are not possible on the file
/* and set errno appropriately.
/*****************************************************************/
```

args: Return argument list, argument count, and environment (os.c)

```
char **args(argc, envp)
int *argc;
char ***envp;
/*****************************************************************/
/* Parse a UNIX style argument list. Return a pointer to a NULL terminated
/* array of null terminated strings. Each string in "*envp" corresponds
/* to one blank or tab terminated argument on the command line. The first
/* argument is the name of the command. The number of arguments is returned
/* in "argc". "*envp" is a pointer to a NULL terminated array of null terminated
/* strings. Each string in the array corresponds to environment variables
/* passed down from the invoking program (such as the UNIX shell).
/*****************************************************************/
```

atof: Ascii to floating conversion (atof.c)

```
double atof(str)
register char *str;
/*****************************************************************/
/* Convert a null terminated ascii string to a "double" and return it.
/*****************************************************************/
```

atoi: Ascii to integer conversion (atoi.c)

```
int atoi(str)
char *str;
/*****************************************************************/
/* Convert a null terminated ascii string to an "int" and return it.
/*****************************************************************/
```

atol: Ascii to long conversion (atol.c)

```
long atol(str)
char *str;
/*****************************************************************/
/* Convert a null terminated ascii string to a long and return it.
/*****************************************************************/
```

bcmp: Byte string compare (bcmp.c)

```
bcmp(b1, b2, length)
register char *b1;
register char *b2;
/*****************************************************************/
/* Compare two character strings of length, length. Return a negative
/* number if "b1" < "b2"; a 0 if "b1" == "b2"; and a positive number if
/* "b1" > "b2". The machine dependent character ordering is used.
/* Null (' ') is considered as an ordinary character.
/*****************************************************************/
```

bcopy: Byte string copy (bcopy.c)

```
bcopy(from, to, n)
register char *to;
register char *from;
register int n;
/*****************************************************************/
/* Copy n bytes from the address, "from", to the address, "to".
/*****************************************************************/
```

bufcpy: Buffer copy (bufcpy.c)

```
bufcpy(to, from, n)
register char *to;
register char  *from;
register int n;
/*****************************************************************/
/* Copy "n" bytes from the address, "from", to the address, "to".
/*****************************************************************/
```

bzero: Byte string zero (bzero.c)

```
bzero(pt, n)
register char *pt;
register int n;
/*****************************************************************/
/* Stores zeros in the "n" bytes pointed to by "pt".
/*****************************************************************/
```

calloc: C allocate memory (calloc.c)

```
char *calloc(num, size)
unsigned num, size;
/*****************************************************************/
/* Allocate a block of zero filled memory large enough to hole "num"
/* items of size, "size". Return a pointer to the new block of memory.
/*****************************************************************/
```

cfree: C free memory (cfree.c)

```
cfree(item)
char *item;
/*****************************************************************/
/* Free a block of memory allocated by calloc(). The memory is no
/* longer available after it is freed. It is imperative that there
/* exist no active pointers to the memory after it is freed.
/*****************************************************************/
```

_cleanup: Clean up file buffers (cleanup.c)

```
_cleanup()
/*****************************************************************/
/* fclose() all opened FILEs.
/*****************************************************************/
```

clearerr: Clear error indicator on a FILE (<stdio.h>)

```
clearerr(file)
FILE *file;
/*****************************************************************/
/* Macro to clear the error indicator returned by ferror() for the FILE, "file".
/*****************************************************************/
```

clearn: Clear n bytes of memory (clearn.c)

```
clearn(n, pt)
register int n;
register char *pt;
/*****************************************************************/
/* Store zeros in the "n" bytes pointed to by "pt".
/*****************************************************************/
```

close: Close an I/O channel (os.c)

```
close(fno)
/*****************************************************************/
/* Close the file associated with the file number "fno" (returned by open()
/* or creat()). Return 0 if all goes well. Return -1 if something went
/* wrong and set errno appropriately.
/*****************************************************************/
```

creat: Create file and open an I/O channel to it (os.c)

```
creat(filename, prot)
char *filename;
/*****************************************************************/
/* Create and open for writing a file named "filename" with protection
/* as specified by "prot". "filename" is a null terminated string.
/* "prot" is expressed in UNIX format, it is the logical or of:
/*
/*              0400: owner read
/*              0200: owner write
/*              0100: owner execute / search directory
/*              0040: group read
/*              0020: group write
/*              0010: group execute / search directory
/*              0004: other (world) read
/*              0002: other (world) write
/*              0001: other (world) execute / search directory
/*
/* For things like terminals that cannot be created it should just open the
/* device. If successful, return a small integer file number. On failure
/* return -1 and set errno appropriately.
/*****************************************************************/
```

ctype: Character type array (ctype.c)

```
unsigned char _ctype_[];
/*****************************************************************/
/* _ctype_ defines the character types for the <ctype.h> character type macros.
/*****************************************************************/
```

_doprnt: Print variable argument list formatted output to file (printf.c)

```
_doprnt(format, args, stream)
char *format;
va_list *args;
FILE *stream;
/*****************************************************************/
/* Write formatted output to the output file, "stream". The
/* format specification is given by the null terminated string, "format".
/* The output variables are obtained from the variable argument list, "args",
/* by the varargs mechanism. Normally only used by printf, sprintf, fprintf,
/* and eprintf.
/*****************************************************************/
```

_doscan: Read variable argument list formatted input from a FILE (scanf.c)

```
_doscan(format, args, stream)
char *format;
va_list *args;
FILE *stream;
/*****************************************************************/
/* Read formatted input from the input file, "stream". The format
/* specification is given by the null terminated string, "format".
/* The input variables are obtained from the variable argument list, "args",
/* by the varargs mechanism. Normally only used by scanf, sscanf, fscanf.
/*****************************************************************/
```

ecvt: E format conversion (ecvt.c)

```
char *ecvt(value, ndig, decpt, sign)
double value;
int ndig;
int *decpt;
int *sign;
/*****************************************************************/
/* Convert the floating point value, "value", to an ascii string and return a
/* pointer to the string. Generate a properly rounded digit string for "ndidg"
/* digits of precision in (%e") format. Leading zeros may be suppressed.
/* The decimal point position is specified by "*decpt" as relative to the first
/* character of the returned string. If "*decpt" is zero the decimal point is
/* immediately to the left of the first character of the returned string. If
/* "*decpt" is positive, the decimal point is to the left of the character
/* numbered "*decpt" (the first character is numbered zero). If "*decpt" is
/* negative, leading zeros have been suppressed and the decimal point occurs
/* "-*decpt" characters to the left of the first character of the string. "*sign"
/* is set to non-zero if "value" is negative, otherwise "*sign" is set to 0.
/*****************************************************************/
```

EOF: End of FILE character value (<stdio.h>)

```
#define EOF     (-1)
/*****************************************************************/
/* The end of FILE character returned by getc() when
/* the end of file is reached.
/*****************************************************************/
```

eprintf: Print formatted output to standard error (printf.c)

```
int eprintf(format, va_alist)
char *format;
va_dcl
/*****************************************************************/
/* Write formatted output to the standard error file. The format specification
/* is given by the null terminated string, "format". The output variables are
/* obtained from the variable argument list by the varargs mechanism. Return
/* the number of characters output.
/*****************************************************************/
```

errno: Error number (errno.c)

```
int errno;
/*****************************************************************/
/* The error number global variable.  It is initialized to zero.  Whenever
/* an error occurs the error number, as defined in <errno.h> and sys_errlist.c
/* is stored into errno.  errno is not cleared when no error occurs!
/*****************************************************************/
```

_exit: Exit program (os.c)

```
_exit(code)
/*****************************************************************/
/* Exit from the program with a status code specified by code.
/* DO NOT RETURN!
/*****************************************************************/
```

exit: Close all files and exit from program (exit.c)

```
exit(val)
int val;
/*****************************************************************/
/* Call _cleanup() to flush all files.  Then call _exit() to exit the
/* program.  Never returns.
/*****************************************************************/
```

fabs: Floating point absolute value (fabs.c)

```
double fabs(x)
double x;
/*****************************************************************/
/* Return the absolute value of x.
/*****************************************************************/
```

fclose: FILE Close (fclose.c)

```
fclose(file)
register FILE *file;
/*****************************************************************/
/* Flush out any data in the buffer corresponding to the FILE, "file", out
/* to the I/O channel corresponding to "file".  Then close "file".
/* Returns 0 if the close is successful, -1 if the close fails.
/*****************************************************************/
```

fcvt: F format conversion (ecvt.c)

```
char *fcvt(value, ndig, decpt, sign)
double value;
int ndig;
int *decpt;
int *sign;
/*****************************************************************/
/* Convert the floating point value, "value", to an ascii string and return a
/* pointer to the string. Generate a properly rounded digit string for "ndig"
/* digits to the right of the decimal point in fixed point ("%f") format.
/* Leading zeros may be suppressed. The decimal point position is specified by
/* "*decpt" as relative to the first character of the returned string. If "*decpt"
/* is zero the decimal point is immediately to the left of the first character
/* of the returned string. If "*decpt" is positive, the decimal point is to the
/* left of the character numbered "*decpt" (the first character is numbered zero).
/* If "*decpt" is negative, leading zeros have been suppressed and the decimal
/* point occurs "-*decpt" characters to the left of the first character of the
/* string. "*sign" is set to non-zero if "value" is negative, otherwise "*sign" is
/* set to 0.
/*****************************************************************/
```

fdopen: FILE open with descriptor (fopen.c)

```
FILE *fdopen(fno, mode)
char *mode;
/*****************************************************************/
/* OPen and return a FILE associated with the I/O channel, "fno", with
/* the access mode specified by the null terminated string, "mode" (as
/* described below in fopen()).
/*****************************************************************/
```

feof: End of FILE query (<stdio.h>)

```
feof(file)
FILE *file;
/*****************************************************************/
/* Return non-zero if the FILE, "file" is at the end of the file.
/*****************************************************************/
```

ferror: Error on FILE query (<stdio.h>)

```
ferror(file)
FILE *file;
/*****************************************************************/
/* Return non-zero if an error has occurred on the FILE, "file".
/*****************************************************************/
```

fflush: FILE flush buffer (fflush.c)

```
fflush(file)
register FILE *file;
/*****************************************************************/
/* Flush out any data in the buffer corresponding to the FILE, "file", out
/* to the I/O channel corresponding to "file".
/*****************************************************************/
```

ffs: Find first set bit (ffs.c)

```
int ffs(i)
int i;
/*****************************************************************/
/* Return the number of the least significant bit in "i" that is a one (counting
/* the least significant bit as 1!). If "i" is zero, return 0.
/*****************************************************************/
```

fgetc: FILE get character (fgetc.c)

```
fgetc(file)
register FILE *file;
/*****************************************************************/
/* Read and return a character from the input FILE, "file. Returns EOF on
/* end of file. fgetc() is identical in operation to getc(), except that
/* it is a function instead of a macro.
/*****************************************************************/
```

fgets: FILE get string (fgets.c)

```
char *fgets(str, n, file)
char *str;
int n;
register FILE *file;
/*****************************************************************/
/* Read a new line terminated line from the FILE, "file. Place up to the first
/* n-1 characters from the line, including the new line character into the
/* buffer, "str". Terminate the line with a null character. Return "str".
/*****************************************************************/
```

_filbuf: Fill FILE input buffer (filbuf.c)

```
_filbuf(file)
register FILE *file;
/*****************************************************************/
/* Fill the buffer associated with the FILE, "file", from the corresponding
/* I/O channel. The buffer must be empty when this operation is done.
/* Returns the next character from "file" or EOF if the end of file
/* is reached. Normally this function is only called from the getc() macro.
/*****************************************************************/
```

FILE: FILE type definition (<stdio.h>)

```
#define FILE struct iobuf
/*****************************************************************/
/* The type definition for the FILE data type.
/*****************************************************************/
```

fileno: I/O channel number associated with a FILE (<stdio.h>)

```
fileno(file)
FILE *file;
/*****************************************************************/
/* Macro to return the channel number associated with the FILE, "file".
/*****************************************************************/
```

filln: Fill n bytes of memory (filln.c)

```
filln(n, pt, fill)
register int n;
register char *pt;
register char fill;
/*****************************************************************/
/* File the n bytes of memory pointed to by "pt" with the character, "fill".
/*****************************************************************/
```

_flsbuf: Flush FILE output buffer (flsbuf.c)

```
_flsbuf(ch, file)
register FILE *file;
char ch;
/*****************************************************************/
/* Output the character "ch" to the FILE, "file". Then output any data stored in
/* the buffer associated with the FILE to the corresponding I/O channel.
/* Normally this function is only called by the putc() macro.
/*****************************************************************/
```

fopen: FILE open (fopen.c)

```
FILE *fopen(name, mode)
char *name, *mode;
/*****************************************************************/
/* Open the file named by the null terminated string, "name", with the
/* access mode specified by the null terminated string, "mode". The access
/* modes are "r", "w", "a", "r+", "w+", "a+". "r" specifies read only.
/* "w" specifies write only and that the file is to be created or replaced.
/* "a" specifies write only and that the file is to be created if doesn't
/* exist or if it exists it is to be appended to. "r+" specifies that the file
/* must exist, that it may be read or written, and that it is to be initially
/* opened at the beginning. "w+" specifies that the file is to be created or
/* replaced but that it may either read or written. "a+" specifies that the
/* file opened such that the first operation will occur at the end of the file.
/* The file may be either read or written.
/*****************************************************************/
```

fprintf: Print formatted output to a FILE (printf.c)

```
int fprintf(stream, format, va_alist)
FILE *stream;
char *format;
va_dcl
/*****************************************************************/
/* Write formatted output file, "stream". The
/* format specification is given by the null terminated string, "format".
/* The output variables are obtained from the variable argument list
/* by the varargs mechanism. Return the number of characters output.
/*****************************************************************/
```

fputc: FILE put character (fputc.c)

```
fputc(ch,file)
int ch;
register FILE *file;
/*****************************************************************/
/* Output the character, ch, to the output FILE, "file". fputc is identical
/* in operation to putc, except that it is a function instead of a macro.
/*****************************************************************/
```

fputs: FILE put string (fputs.c)

```
fputs(str,file)
register char *str;
register FILE *file;
/*****************************************************************/
/* Output the null terminated string, "str", to the output FILE, "file".
/* The terminating null character is not written to stdout.
/*****************************************************************/
```

fread: FILE read (fread.c)

```
fread(pt,size,nitems,file)
register char *pt;
register FILE *file;
/*****************************************************************/
/* Read "nitems" items of size, "size", beginning at the pointer, "pt", to the
/* FILE, "file. Return the number of items actually read. Return 0
/* at end of file.
/*****************************************************************/
```

free: Free memory (malloc.c)

```
free(pt)
char *pt;
/*****************************************************************/
/* Free a block of memory allocated by malloc(). The memory is no
/* longer available after it is freed. It is imperative that there
/* exist no active pointers to the memory after it is freed.
/*****************************************************************/
```

freopen: FILE re-open (fopen.c)

```
FILE *freopen(name, mode, file)
char *name, *mode;
FILE *file;
/*****************************************************************/
/* If the FILE, "file", is open, close it. Then open the file named
/* by the null terminated string, "name", with the access mode,
/* "mode" (as specified in fopen above), on the FILE, "file".
/*****************************************************************/
```

frexp: Floating point return exponent and mantissa (frexp.c)

```
double frexp(value, eptr)
double value;
int *eptr;
/*****************************************************************/
/* Return the mantissa of the floating point value, "value". Return the
/* binary radix exponent value in "*iptr". The return value is chosen
/* such that 0.5 <= frexp(value, &iptr) < 1.0 and such that
/* value == frexp(value, &iptr) * 2 ** iptr
/*****************************************************************/
```

fseek: FILE seek (fseek.c)

```
fseek(stream, offset, ptrname)
long offset;
FILE *stream;
/*****************************************************************/
/* Seek to a new position within the file, "stream". If "ptrname" is 0, seek to
/* offset bytes from the beginning of the file. If "ptrname" is 1, seek to
/* offset bytes from the current position in the file. If "ptrname" is 2 seek
/* to offset bytes from the end of the file. fseek does no I/O. The next
/* I/O operation to the file, "stream", will begin at the new position in the
/* file. fseek() undoes the effect of any ungetc().
/*
/* Return 0 if the seek is successful. If an error occurs return -1.
/*****************************************************************/
```

fscanf: Read formatted input from a FILE (scanf.c)

```
fscanf(stream, format, va_list)
FILE *stream;
char *format;
va_dcl
/*****************************************************************/
/* Read formatted input from the input FILE, "stream". The format
/* specification is given by the null terminated string, "format".
/* The input variables are obtained from the variable argument list
/* by the varargs mechanism. Return the number of input variables assigned
/* to.
/*****************************************************************/
```

fstat: Get device and file number connected to I/O channel (os.c)

```
fstat(fno, statptr)
struct stat *statptr;
/*****************************************************************/
/* The Green Hills Software C Library does not use the fstat function, it is only
/* needed if the Green Hills Software Fortran Library is being used.
/*
/* "fno" is a file number returned by open() or creat(). In the two fields
/* st_dev and st_ino place values which uniquely identify the file or device
/* opened on file number "fno". In UNIX these are the device number and the
/* inode (file) number on the device.
/*
/* Return 0 if the file status is correctly returned. Return -1 if no file
/* is opened on file number "fno", and set errno appropriately.
/*****************************************************************/
```

ftell: FILE tell position (ftell.c)

```
long ftell(stream)
FILE *stream;
/*****************************************************************/
/* Return the current position from the beginning of the FILE, "stream".
/*****************************************************************/
```

ftruncate: Truncate file connected to I/O channel (os.c)

```
ftruncate(fno, length)
/*****************************************************************/
/* The Green Hills Software C Library does not use the ftruncate function, it is
/* only needed if the Green Hills Software Fortran Library is being used.
/*
/* "fno" is a file number returned by open() or creat(). The file number "fno"
/* must be opened for writing. If the file opened for writing on file number
/* "fno" is longer that "length" bytes, it is truncated to "length" bytes.
/*
/* Return 0 if the named file is truncated or was previously less than
/* length bytes. Return -1 if the file is not open for writing or if the
/* file cannot be truncated, and set errno appropriately.
/*****************************************************************/
```

fwrite: FILE write (fwrite.c)

```
fwrite(pt,size,nitems,file)
register char *pt;
register FILE *file;
/*****************************************************************/
/* Write "nitems" items of size, size beginning at the pointer, pt, to the
/* FILE, "file". Return the number of items actually written.
/*****************************************************************/
```

gcvt: G Format conversion (gcvt.c)

```
char *gcvt(value, ndig, buf)
double value;
int ndig;
char *buf;
/*****************************************************************/
/* Convert the floating point value, "value", to an ascii string in the
/* buffer, "buf". If possible generate "ndig" digits after the decimal
/* point in "%f" format. If not possible generate in "%e" format.
/* Suppress trailing zeros. Return a pointer to the string.
/*****************************************************************/
```

getc: Get a character from a file (<stdio.c>)

```
getc(f)
FILE *f
/*****************************************************************/
/* A macro to read and return a character from the input FILE, "file".
/* Returns EOF on end of file.
/*****************************************************************/
```

getchar: Get character from standard input FILE (<stdio.h>)

```
getchar()
/*****************************************************************/
/* A macro to read and return a character from the standard input FILE, stdin.
/* Returns EOF on end of file. It has the same effect as the getchar()
/* function below, but it takes more code space, but less execution time.
/*****************************************************************/
```

getchar: Get character from standard input FILE (getchar.c)

```
getchar()
/*****************************************************************/
/* A function to read and return a character from the standard input
/* FILE, stdin. Returns EOF on end of file. It has the same effect as the
/* getchar() macro above, but it takes less code space, but more execution
/* time.
/*
/* If you have included <stdio.h>, in order to use this function you must
/* place a line containing "#undef getchar" before the first use. From then
/* on the getchar() macro will be inaccessible.
/*****************************************************************/
```

getl: Get long from FILE (getl.c)

```
long getl(file)
register FILE *file
/*****************************************************************/
/* Read and return a "long" from the FILE, "file".
/*****************************************************************/
```

getpid: Get Process ID (os.c)

```
getpid()
/*****************************************************************/
/* getpid returns the current process number.  getpid is used to create
/* filenames which will not conflict with filenames created by another
/* process at the same time.
/*****************************************************************/
```

gets: Get string from the standard input FILE (gets.c)

```
char *gets(str)
char *str
/*****************************************************************/
/* Read a new line terminated string from the standard input FILE, stdin.
/* Return a pointer to a string containing the line from the file with
/* the new line changed to a null ('\0') character.
/*****************************************************************/
```

getw: Get int from FILE (getw.c)

```
getw(file)
register FILE *file;
/*****************************************************************/
/* Read and return an "int" from the FILE, "file".
/*****************************************************************/
```

Init_IO: Initialize I/O (os.c)

```
Init_IO()
/*****************************************************************/
/* Init_IO is called from START() to initialize the I/O system.
/*
/* Open file numbers 0, 1, and 2 like the UNIX shell.
/*
/* File number 0 is the standard input file.  It is usually the controlling
/* terminal for the program or a sequential character input file.
/*
/* File number 1 is the standard output file.  It is usually the controlling
/* terminal for the program or a sequential character output file.
/*
/* File number 2 is the standard error file.  It is usually the controlling
/* terminal for the program.  Error messages are usually written on this
/* file rather than standard output so that if the standard output is a file,
/* the error message can still be seen by the person running the program.
/* In batch jobs, file number 2 is usually a sequential character output file.
/*
/* On non-UNIX systems "/dev/tty" should be changed to whatever is appropriate
/* ("SYS$INPUT": on VMS, "TTY": on tops-10, etc.).
/*
/* If these files are already opened by the operating system, or by the UNIX
/* shell, delete the calls to open.
/*****************************************************************/
```

INPUT: Return character input from the controlling terminal (os.c)

```
int INPUT()
/*****************************************************************/
/* Read a character from the controlling terminal and return it.
/*
/* If you want to remain compatible with UNIX stdio and the Green Hills Software
/* C Test Program you must convert carriage returns ('\r') to line feeds ('\n').
/*****************************************************************/
```

isalnum: Is alphanumeric (<ctype.h>)

```
isalnum(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is an alphanumeric character. Returns 0
/* otherwise.
/*****************************************************************/
```

isalpha: Is alphabetic character (<ctype.h>)

```
isalpha(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is an alphabetic character. Returns 0
/* otherwise.
/*****************************************************************/
```

isascii: Is ascii character (<ctype.h>)

```
isascii(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is an ascii character. Returns 0
/* otherwise.
/*****************************************************************/
```

isatty: Is a tty connected to I/O channel (os.c)

```
isatty(fno)
/*****************************************************************/
/* Return 1 if the file number "fno" (returned by open() or creat())
/* is connected to a terminal, otherwise return 0.
/*****************************************************************/
```

iscntl: Is control character (<ctype.h>)

```
iscntl(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is a control character. Returns 0
/* otherwise.
/*****************************************************************/
```

iscsym: Is C identifier character (<ctype.h>)

```
iscsym(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" could be a character of
/* a C identifier. Returns 0 otherwise.
/*****************************************************************/
```

iscsymf: Is C identifier initial character (<ctype.h>)

```
iscsymf(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" could be the initial character of
/* a C identifier. Returns 0 otherwise.
/*****************************************************************/
```

isdigit: Is decimal digit (<ctype.h>)

```
isdigit(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is a decimal digit. Returns 0
/* otherwise.
/*****************************************************************/
```

isgraph: Is a graphics character (<ctype.h>)

```
isgraph(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is a graphics character. Returns 0
/* otherwise.
/*****************************************************************/
```

islower: Is lower case ascii character (<ctype.h>)

```
islower(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is a lower case letter. Returns 0
/* otherwise.
/*****************************************************************/
```

isprint: Is printable character (<ctype.h>)

```
isprint(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is a printable character. Returns 0
/* otherwise.
/*****************************************************************/
```

ispunct: Is printable character (<ctype.h>)

```
ispunct(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is a punctuation character. Returns 0
/* otherwise.
/*****************************************************************/
```

isspace: Is white space character (<ctype.h>)

```
isspace(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is a white space character. Returns 0
/* otherwise.
/*****************************************************************/
```

isupper: Is upper case ascii character (<ctype.h>)

```
isupper(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is an upper case letter. Returns 0
/* otherwise.
/*****************************************************************/
```

isxdigit: Is hexadecimal digit (<ctype.h>)

```
isxdigit(ch)
char ch;
/*****************************************************************/
/* Macro which returns non-zero if "ch" is a hexadecimal digit. Returns 0
/* otherwise.
/*****************************************************************/
```

index: Return the index of a character in a string (index.c)

```
char *index(str, ch)
register char *str;
register char ch;
/*****************************************************************/
/* Return a pointer to the first instance of the character in the string, "str" (or
/* a 0 if the character is not there. Note that this will work if the
/* character is NUL. Identical to strchr().
/*****************************************************************/
```

labs: Long absolute value (labs.c)

```
long labs(x)
long x;
/*****************************************************************/
/* Return the absolute value of x.
/*****************************************************************/
```

ldexp: Load exponent (ldexp.c)

```
double ldexp(value,exp)
double value;
/*****************************************************************/
/* Return "value" * 2 ** "exp". That is, return "value" with the binary exponent
/* incremented by "exp".
/*****************************************************************/
```

lseek: Long seek on an I/O channel (os.c)

```
lseek(fno, offset, end)
long offset;
/*****************************************************************/
/* Seek to a new position within the file connected to the file number "fno"
/* (returned by open() or creat()). If "end" is 0, seek to offset bytes from
/* the beginning of the file. If "end" is 1, seek to offset bytes from the
/* current position in the file. If "end" is 2, seek to offset bytes from
/* the end of the file. lseek does no I/O. The next I/O operation to
/* file number "fno" will begin at the new position in the file.
/*
/* Return the offset from the beginning of the file after the seek takes
/* place. If an error occurs return -1 and set errno appropriately.
/* If an error occurs the file position is not changed.
/*****************************************************************/
```

malloc: Memory allocate (malloc.c)

```
char *malloc(siz)
unsigned siz;
/*****************************************************************/
/* Allocate and return a new data area of size, "siz". If no more memory
/* is available return NULL.
/*****************************************************************/
```

mktemp: Make temporary name (mktemp.c)

```
char *mktemp(str)
char *str;
/*****************************************************************/
/* The null terminated input string, "str", must have 'X's as its last six
/* characters. These characters are replaced by a letter and the process
/* id returned by getpid(). Return "str".
/*****************************************************************/
```

```
modf:     Modulo 1 floating point (modf.c)
              double modf(value, iptr)
              double value;
              double *iptr;
              /*****************************************************************/
              /* Split "value" into its integer and fractional parts, such that
              /* 0 <= modf(value, iptr) < 1.0 && (value == modf(value, iptr) + iptr)
              /*
              /* *iptr = floor(x) and return (x - *iptr)
              /*****************************************************************/

NULL:     Null pointer value (<stdio.h>)
              #define NULL        0
              /*****************************************************************/
              /* The NULL pointer value.  Also used as the NULL FILE value.
              /*****************************************************************/

open:     Open an I/O channel (os.c)
              open(filename, mode)
              char *filename;
              /*****************************************************************/
              /* Open the file named "filename" with the indicated mode.
              /* "filename" is a null terminated string and "mode" is one of the following
              /*
              /*              0 => open for reading
              /*              1 => open for writing
              /*              2 => open for reading & writing
              /*
              /* If the open is successful return a small integer file number.
              /* On failure return -1, and set errno in accordance with the error.
              /*****************************************************************/

OUTPUT:   Output a character to the controlling terminal (os.c)
              OUTPUT(ch)
              /*****************************************************************/
              /* Write the character "ch" to the controlling terminal.
              /*
              /* If you want to remain compatible with UNIX stdio you must convert a
              /* line-feed ('\n') to a carriage-return ('\r') and a line-feed ('\n').
              /*****************************************************************/
```

perror: Print error message (perror.c)

```
perror(str)
char *str;
/*****************************************************************/
/* Print on the standard error file, stderr, the null terminated string, "str",
/* followed by the string ":", followed by the error message corresponding
/* to the current value of global "int" variable, errno. The error message
/* list is specified by the array sys_errlist defined in sys_errlist.c.
/*****************************************************************/
```

printf: Print formatted output to standard output (printf.c)

```
int printf(format, va_alist)
char *format;
va_dcl
/*****************************************************************/
/* Write formatted output to the standard output file, stdout. The
/* format specification is given by the null terminated string, "format".
/* The output variables are obtained from the variable argument list
/* by the varargs mechanism. Return the number of characters output.
/*****************************************************************/
```

putc: Output a character to a FILE (<stdio.h>)

```
putc(ch,f)
char ch;
FILE *f;
/*****************************************************************/
/* A macro to output the character, "ch", to the output FILE, "f".
/*****************************************************************/
```

putchar: Output a character to a standard output FILE (<stdio.h>)

```
putchar(ch)
/*****************************************************************/
/* A macro to output the character, "ch", to the standard output
/* FILE, stdout. It has the same effect as the putchar() function below, but
/* it takes more code space, but less execution time.
/*****************************************************************/
```

putchar: Output a character to a standard output FILE (putchar.c)

```
putchar(ch)
/*****************************************************************/
/* A function to output the character, "ch", to the standard output
/* FILE, stdout. It has the same effect as the putchar() macro above, but
/* it takes more code space, but less execution time.
/*
/* If you have included <stdio.h>, in order to use this function you must
/* place a line containing "#undef putchar" before the first use. From then
/* on the putchar() macro will be inaccessible.
/*****************************************************************/
```

putl: Output a long to a FILE (putl.c)

```
long putl(l, file)
long l;
register FILE *file;
/*****************************************************************/
/* Output the long, "l", to the FILE, "file". Return "l".
/*****************************************************************/
```

puts: Output a string to the standard output FILE (puts.c)

```
puts(str)
register char *str;
/*****************************************************************/
/* Output the null terminated string, "str", to the standard output FILE,
/* stdout. The terminating null character is not written to stdout, in
/* its place a new line ('\n') is written.
/*****************************************************************/
```

putw: Output a word (int) to a FILE (putw.c)

```
putw(w, file)
int w;
register FILE *file;
/*****************************************************************/
/* Output the int, "w", to the file, "file".
/*****************************************************************/
```

rand: Random integer (rand.c)

```
int rand()
/*****************************************************************/
/* Return a random positive integer.
/*****************************************************************/
```

read: Read from an I/O channel (os.c)

```
read(fno, buf, size)
char *buf;
/*****************************************************************/
/* Read at most "size" bytes into "buf" from the file connected to "fno" (where
/* "fno" is one of the file numbers returned by open() or creat()).
/* Return the number of bytes read. Return 0 at end of file. Return -1 on
/* error and set errno appropriately.
/*
/* The Green Hills Software C Test Program assumes that (as in UNIX) on line
/* buffered terminals input carriage returns ('\r') are converted to line feeds ('\n').
/*****************************************************************/
```

realloc: Reallocate memory (realloc.c)

```
char *realloc(old, new_size)
char *old;
unsigned new_size;
/*********************************************************************/
/* The memory block pointed to by "old" must have been allocated by malloc().
/* Return the address of a new block of memory of the size, "new_size",
/* which contains the same data as the old block.  The old block is freed
/* with free().
/*********************************************************************/
```

rewind: Seek to beginning of a FILE (rewind.c)

```
rewind(stream)
FILE *stream;
/*********************************************************************/
/* Seek to the beginning of the FILE, "stream".
/* Equivalent to fseek(stream, 0L, 0).
/*********************************************************************/
```

rindex: Reverse search for character in string(strrchr.c)

```
char *rindex(str, ch)
register char *str;
register char ch;
/*********************************************************************/
/* Return a pointer to the last location of the character, "ch", in the string,
/* "str", 0 if not found.
/*********************************************************************/
```

sbrk: Set memory address break (os.c)

```
char *sbrk(size)
/*********************************************************************/
/* Return a pointer to at least "size" bytes of contiguous read/write memory.
/* The memory returned by sbrk on different calls need not be contiguous
/* with each other (although they should be for compatibility with UNIX).
/* Return -1 on an error and set errno = ENOMEM.
/*********************************************************************/
```

scanf: Read formatted input from standard input (scanf.c)

```
scanf(format, va_alist)
char *format;
va_dcl
/*****************************************************************/
/* Read formatted input from the standard input file, stdin. The
/* format specification is given by the null terminated string, "format".
/* The input variables are obtained from the variable argument list
/* by the varargs mechanism. Return the number of input variables assigned to.
/*****************************************************************/
```

setbuf: Set a user I/O buffer for a FILE (setbuf.c)

```
setbuf(stream, buf)
FILE *stream;
char buf[BUFSIZ];
/*****************************************************************/
/* Use the data area specified by "buf" as an I/O buffer for the FILE, "stream".
/* If "buf" is NULL, then use unbuffered output on the FILE, "stream".
/*****************************************************************/
```

setlinebuf: Set line buffering on a FILE (setlinbuf.c)

```
setlinebuf(stream)
FILE *stream;
/*****************************************************************/
/* Set the FILE, "stream", to be line buffered. That is buffer output
/* until a '\n' character is encountered, then flush the output to
/* the output channel.
/*****************************************************************/
```

sprintf: Print formatted output to string (printf.c)

```
sprintf(s, format, va_alist)
char *s;
char *format;
va_dcl
/*****************************************************************/
/* Write null terminated formatted output to the string, "str". The
/* format specification is given by the null terminated string, "format".
/* The output variables are obtained from the variable argument list
/* by the varargs mechanism. Return the number of characters output
/* to the string (not counting the null).
/*****************************************************************/
```

srand: Initialize random number generator used in rand() (rand.c)

```
srand(val)
int val;
/*****************************************************************/
/* Initialize the random number generator used by rand().
/*****************************************************************/
```

sscanf: Read formatted input from a string (scanf.c)

```
sscanf(str, format, va_alist)
char *str;
char *format;
va_dcl
/*****************************************************************/
/* Read formatted input from the null terminated string, "str". The
/* format specification is given by the null terminated string, "format".
/* The input variables are obtained from the variable argument list
/* by the varargs mechanism. Return the number of input variables assigned to.
/*****************************************************************/
```

START: Program transfer address (crt0.c)

```
START()
/*****************************************************************/
/* START is the function that starts all C programs. This file must be
/* specified to the Link Editor (linker) in such a way that program execution
/* starts at the top of this function.
/*
/* START() initializes the processor, stack pointer, I/O, and memory systems
/* then it calls the program's main() function with the command line
/* arguments. If the main() function returns to the START function, START
/* exits by calling exit().
/*
/* As written, this file is a C function. It may be possible to leave this
/* function written in C (possible using "asm" statements), but more than
/* likely you will find it necessary to compile this function into assembly
/* language and then edit the assembly code so that it can be used. In
/* particular, there may be procedure entry code at the top of this function
/* to save registers. This code may depend on a stack pointer having already
/* been initialized. But since it is the job of this function to load a
/* stack pointer (if it hasn't already been loaded), you may need to delete
/* the function entry code.
/*****************************************************************/
```

stat: Get device and file number of file name (os.c)

```
stat(name, statptr)
char *name;
struct stat *statptr;
/*****************************************************************/
/* The Green Hills Software C Library does not use the stat function, it is only
/* needed if the Green Hills Software Fortran Library is being used.
/*
/* "name" is the name of a file. In the two fields st_dev and st_ino place
/* values which uniquely identify the named file or device. In UNIX, these
/* are the device number and the inode (file) number on the device.
/*
/* Return 0 if the file status is correctly returned. Return -1 if no file
/* with that name exists, and set errno appropriately.
/*****************************************************************/
```

stdin: Standard input file (<stdio.h>)

```
#define stdin (&_iob[0])
/*****************************************************************/
/* The standard input FILE.
/*****************************************************************/
```

stdout: Standard output file (<stdio.h>)

```
#define stdout (&_iob[1])
/*****************************************************************/
/* The standard output FILE.
/*****************************************************************/
```

stderr: Standard error FILE (<stdio.h>)

```
#define stderr (&_iob[2])
/*****************************************************************/
/* The standard error FILE.
/*****************************************************************/
```

strcat: String concatenate (strcat.c)

```
char *strcat(str2, str1)
char *str2;
register char *str1;
/*****************************************************************/
/* Copy the null terminated string, "str1", onto the end of the null terminated
/* string, "str2". The first character of "str1" replaces the null terminating "str2".
/* No test is made for overflowing "str2".
/*****************************************************************/
```

strchr: Return the position of a character in a string (strchr.c)

```
char *strchr(str, ch)
register char *str;
register char ch;
/*****************************************************************/
/* Return a pointer to the first instance of the character, "ch", in the
/* null terminated string, "str". Return 0 if the character does not appear
/* in "str". Note that this will work if the character is null (' ').
/* Identical to index().
/*****************************************************************/
```

strcmp: String compare (strcmp.c)

```
strcmp(str1, str2)
register char *str1;
register char *str2;
/*****************************************************************/
/* Compare two null terminated strings. Return a negative number if
/* "str1" < "str2"; a 0 if "str1" == "str2"; and a positive number if "str1" > "str2".
/* The machine dependent character ordering is used.
/*****************************************************************/
```

strcpy: String Copy (strcpy.c)

```
char *strcpy(str2, str1)
char *str2;
register char *str1;
/*****************************************************************/
/* Copy the null terminated string, "str1", to "str2" up to and including the null
/* character that terminates "str1". Note that there is no check for an overflow
/* of "str2".
/*****************************************************************/
```

strindex: String index (strindex.c)

```
strindex(str, sub)
char *str;
char *sub;
/*****************************************************************/
/* Find the position of a null terminated substring, "sub", in another null
/* terminated string, "str". Return the offset to the substring's position
/* (0 based). Return -1 if the substring is not found in the string.
/*****************************************************************/
```

strlen: String length (strlen.c)

```
strlen(str)
char *str;
/*****************************************************************/
/* Return the length of the null terminated string "str".
/*****************************************************************/
```

strncat: String concatenate with maximum length (strncat.c)

```
char *strncat(str2, str1, n)
char *str2;
register char *str1;
register int n;
/*****************************************************************/
/* Copy character from "str1" onto the end of the null terminated string,
/* "str2", until a null ('\0') is moved in "str1" or until n characters
/* are moved.
/*****************************************************************/
```

strncmp: String compare with maximum length (strncmp.c)

```
strncmp(str1, str2, n)
register char *str1;
register char *str2;
register int n;
/*****************************************************************/
/* Compare two null terminated strings. Return a negative number if
/* "str1" < "str2"; a 0 if "str1" == "str2"; and a positive number if "str1" > "str2".
/* Uses the machine dependent character ordering. If "str1" and "str2" do not
/* differ in the first "n" characters then they are equal (assuming there is
/* not a null (' ') in the first "n" characters).
/*****************************************************************/
```

strncpy: String copy with maximum length (strncpy.c)

```
char *strncpy(str2, str1, n)
char *str2;
register char *str1;
register int n;
/*****************************************************************/
/* Move exactly "n" characters into "str2", these characters are taken from "str1"
/* until a '\0' appears there, after which "str2" is padded with nulls (' ').
/*****************************************************************/
```

strrchr: Reverse search for a character in a string (strrchr.c)

```
char *strrchr(str, ch)
register char *str;
register char ch;
/*****************************************************************/
/* Return a pointer to the last location of the character, "ch", in the
/* null terminated string, "str". Return 0 if the character does not
/* appear in the string.
/*****************************************************************/
```

strrindex: Reverse search for string in string (strrindex.c)

```
strrindex(str, sub)
char *str;
char *sub;
/*****************************************************************/
/* Search for the last occurrence of a null terminated substring, "sub", in
/* another string, "str". Return the offset to the substring's position (0
/* based). Return -1 if the substring is not found in the string.
/*****************************************************************/
```

strsave: String save (strsave.c)

```
char *strsave(str)
char *str;
/*****************************************************************/
/* Return a pointer to a newly allocated copy of the null terminated string,
/* "str". Return 0 if no memory can be allocated.
/*****************************************************************/
```

strtol: String to long conversion (strtol.c)

```
long strtol(str, ptr, base)
register char *str;
register char **ptr;
register int base;
/*****************************************************************/
/* Convert the null terminated ascii string, "str", into a long value and
/* return it. Interpret the ascii string in the base (radix) specified
/* by "base". Return a pointer to the end of the digit string in "*ptr".
/*****************************************************************/
```

swab: Swap byte string (swab.c)

```
swab(from, to, nbytes)
register char *from;
register char *to;
register int nbytes;
/*****************************************************************/
/* Copy "nbytes" bytes from "from" to "to", swapping adjacent bytes.
/* If "nbytes" is odd then move "nbytes"-1 bytes.
/*****************************************************************/
```

sys_errlist: System error list (syserrlst.c)

```
char *sys_errlist[] = {...};
int sys_nerr = sizeof(sys_errlist)/sizeof(char *);
/*****************************************************************/
/* sys_errlist maps error numbers from errno.h to error message strings.
/*****************************************************************/
```

toascii: Convert character to ascii (<ctype.h>)

```
toascii(ch)
char ch;
/*****************************************************************/
/* Convert the character, "ch", to a legal ascii character, by clearing
/* any high order bits.
/*****************************************************************/
```

tolower: Convert character to lower case (<ctype.h>)

```
tolower(ch)
char ch;
/*****************************************************************/
/* Convert the upper case character, "ch", to its lower case equivalent.
/* Identical to tolower().
/*****************************************************************/
```

_tolower: Convert character to lower case (<ctype.h>)

```
_tolower(ch)
char ch;
/*****************************************************************/
/* Convert the upper case character, "ch", to its lower case equivalent.
/* Identical to _tolower().
/*****************************************************************/
```

toupper: Convert character to upper case (<ctype.h>)

```
toupper(ch)
char ch;
/*****************************************************************/
/* Convert the lower case character, "ch", to its upper case equivalent.
/* Identical to _toupper().
/*****************************************************************/
```

_toupper: Convert character to upper case (<ctype.h>)

```
toupper(ch)
char ch;
/*****************************************************************/
/* Convert the lower case character, "ch", to its upper case equivalent.
/* Identical to toupper().
/*****************************************************************/
```

ungetc: Un-get character from a FILE (ungetc.c)

```
ungetc(ch, file)
int ch;
register FILE *file;
/*****************************************************************/
/* Put the character, "ch", back onto the input FILE, "file" such that
/* the next getc() or other input operation on "file" will return ch.
/* A call to fseek() will erase the effect of a ungetc(). Only one
/* call to getc() may be done between input requests from a FILE.
/*****************************************************************/
```

unlink: Unlink (Delete) file name from directory (os.c)

```
unlink(name)
char *name;
/*****************************************************************/
/* The Green Hills Software C Library does not use the unlink function, it is only
/* needed if the Green HIlls Software Fortran Library is being used.
/*
/* "name" is the name of a file. The named file is deleted.
/* Return 0 if the named file is deleted. Return -1 if there is no such
/* file or if the file cannot be deleted, and set errno appropriately.
/*****************************************************************/
```

write: Write to an I/O channel (os.c)

```
write(fno, buf, size)
char *buf;
/*****************************************************************/
/* Write at most "size" bytes into the file connected to "fno" (where "fno" is
/* one of the file numbers returned by open() or creat()) into "buf".
/* Return the number of bytes written, or -1 to indicate an error and
/* set errno appropriately. On line buffered terminal output (as in UNIX)
/* line feeds ('\n') are converted to a carriage return ('\r') and a
/* line feed ('\n').
/*****************************************************************/
```

CHAPTER 4

Incompatible Changes From Previous Releases

If you have worked with previous releases of the Green Hills Software C Library you may need to know about some incompatible changes that have been made. For the most part this new version is fully compatible with previous versions. However, certain bugs and inconsistencies with UNIX have been reconciled by sacrificing compatibility. In addition, confusing default configuration options have been changed to make the Library more easily understood by the new user. The following changes are INCOMPATIBLE with the previous (May 1985) release.

- The file crtlib.h and its programming style customization macros have been eliminated. The indenting style has been converted to Green Hills Standard.

- The default configuration is now to load in floating point code for printf and scanf. In the previous release it was necessary to specify -Ddo_float to have printf and scanf recognize floating point arguments. This caused some people to believe that floating point was not supported at all. Now, the default is as if -Ddo_float was specified in the previous version. Now, if -Dno_float is specified, then printf and scanf no longer recognize floating point arguments, as was the case in the previous version by default.

- To be compatible with UNIX, the second argument of both index and rindex is now of type char, instead of (char *) as in May 1985 release. The previous version was in error. The old index function has been renamed to strindex. The old rindex function has been renamed to strrindex.

- The function putw now outputs an "int" instead of a "short". The previous version was in error.

- The function getw now inputs an "int" instead of a "short". The previous version was in error.

- The new library functionality demands more user supplied functions in the low level I/O module. For full functionality the additional UNIX calls lseek(), isatty(), and getpid() are needed. Trivial versions of these functions are supplied in the file "os.c".

- fopen(name, "a") now calls a user supplied routine lseek() to seek to the end of the file. The _OP_EOF flag to read() has been discontinued.

- The new function fseek() calls the user supplied routine lseek(). The function mktemp() calls the use supplied routine getpid().

- The Green Hills Software C Library Test Program has been renamed from "testsuite.c" to "test-c.c". The Math Library Test Program has been renamed from "testmath.c" to "test-m.c" for consistency with other test program names.

- The Green Hills Software C Library Test Program option -Dtest_nofiles has been replaced. In the previous version, the library as delivered would only pass the Test Program if -Dtest_nofiles was specified. This was confusing. Now, by default no test is made which requires a file system. That is, in this release the default is as if -Dtest_nofiles was specified to the previous version. In order to test a user implemented file system interface with the Green Hills Software C Library Test Program, it is necessary to specify -Dtest_files when compiling "test-c.c".

CHAPTER 5

Revision history

May, 1984	First Release

May, 1985	Fixed fwrite
	Fixed so open/creat/close do not have to initialize _iob structures
	Added more comments about how to test
	Added perror, sys_errlist, sys_nerr
	Added fdopen
	Fixed *printf, strcpy, strncpy to return as documented under Sys V.
	Fixed floating point output, corrected some problems in others
	Add gcvt, ecvt, fcvt

August, 1986	Many bugs fixed "crtlib.h" deleted. Math library separated and algorithms improved. _cleanup, fseek, freopen added. "a+", "r+", "w+" modes added to fopen, freopen, fdopen. setbuf, setlinbuf, and line buffering added. varargs added to printf and scanf, _doprnt, _doscan added labs, cabs, hypot, bzero, ffs, bcmp, bcopy added Floating point customizability added to math.h Customization documentation improved. Fields in stdio.h substantially modified. Modules reorganized and renamed in many cases. Documentation much improved.