March, 1988
Order Number: 311567-001

iPSC®/2 GREEN HILLS

C LANGUAGE REFERENCE MANUAL

(Preliminary)

ADVANCE INFORMATION

This is a draft copy of the manual. Material in this document is for informational purposes only and is subject to change without notice. Intel Corporation assumes no responsibility for any errors which may appear in this document.

intel Corporation

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.

REV.	REVISION HISTORY	DATE
-001	Original issue	03/88

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. This manual follows the organization of the compiler. The chapters are given below.

Overview

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

The C Language

> The C Language chapter specifies C language features and extensions are supported. It also explains restrictions, known bugs, and indicates of 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

There are a number of documents, in addition to this one, that you will need in order to use C-386. You may already have some of this documentation on your Host computer system. When you receive your compiler you will receive additional Host documentation

customized to your particular environment.

Installation Documentation

If C-386 does not come standard with your computer system, the Host documentation includes Installation Instructions that tell you how to install C-386 on your computer system.

Invocation Documentation

The Host documentation includes Invocation Documentation that tells you how to run the compiler on your particular Host computer system. It gives restrictions that the system imposes on the compiler, such as memory requirements, file naming conventions, and the exact sequence of commands needed to run the compiler.

Assembler and/or Linker Reference Manuals

In order to use C-386 you will have to assemble and/or link its output into a final program. The Host documentation will contain an Assembler and/or Linker reference manual.

Debugger Reference Manual

In order to debug your programs you will need some sort of debugger. The Host documentation includes a Debugger reference manual.

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 chapter 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. If you are using the Green Hills Software C Library then you should have received the Green Hills Software User's Manual for the C Library.

80386 Architecture Manual

You may need to know details of the instruction set and register organization of the 80386 in order to debug your programs. If you do not already have a 80386 Architecture Manual, you should contact the manufacturer of the 386 or the manufacturer of the particular target system that you are dealing with, in order to obtain one.

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 if you do not have one) 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.

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.

2.2.7. 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. For non-UNIX systems which do not already have a C library, Green Hills supplies the Green Hills Software C library. This library is supplied as either object code or C source code, depending on the environment.

The Green Hills Software C Library comes with a Green Hills Software User's Manual for the C Library. This manual describes the functions supported by the C Library. It also gives installation instructions.

To use the Green Hills Software C Library you need a standard C-386 compiler license. Under this license, unlimited distribution of programs which are linked with Green Hills Software C Library object code is permitted without charge. However distribution of the Green Hills Software C Library source code or object code is not permitted.

CHAPTER 3

80386 Target

3.1. Introduction

This chapter describes the 80386 target environment for C-386. There is currently only one target environment: UNIX System V.

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, NS32000, and Fairchild Clipper, opposite of the IBM/370, MC68000, and Z8000. Bits are numbered with bit zero as the least significant bit.

Floating point is IEEE 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.

The -g option generates Common Object File Format (COFF) ".def" symbolic debug pseudo-ops in the assembler language output. The assembler and linker understand and process the symbolic debug entries in the object files. The "sdb" symbolic debugger can be used with C-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 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 when it is advantageous, and eliminates redundant jumps and unreachable code.

4.2. General Optimizations

General Optimizations always make programs smaller and faster. Therefore, by default, these optimizations are always performed.

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 -O 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

```
proc:
        pushl   %esi

        movl    $1,%esi
.L7:
        call    f
        incl    %esi

        cmpl    $10,%esi
        jl      .L7
        movl    $1,%esi

.L4:
        call    g
        incl    %esi

        cmpl    $10,%esi
        jl      .L4
        popl    %esi
        ret

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

UNIX PCC

```
proc:
        pushl   %ebp
        movl    %esp,%ebp
        subl    $8,%esp
        movl    $1,-4(%ebp)
.L17:

        movl    $10,%eax
        cmpl    %eax,-4(%ebp)
        jl      .L18
        movl    $1,-8(%ebp)
        jmp     .L22
.L18:
        call    f
        incl    -4(%ebp)
        jmp     .L17
.L23:
        call    g
        incl    -8(%ebp)
.L22:
        movl    $10,%eax
        cmpl    %eax,-8(%ebp)
        jl      .L23
        leave
        ret

        ----------------
        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

4.2.1.1. C Local Static Variables

Since C-386 knows the complete data flow within a function, it checks each local "static" variable to see if the variable is ever referenced before it is stored into. If so, it is using the value from a previous call to the function, and the variable must be allocated in memory. However, some local "static" variables are initialized before they are referenced, and these variables are treated as automatic variables and are candidates for allocation to registers!

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 optimized short addressing modes to access commonly used variables. If the compiler allocated some very large variable first, the short addressing modes 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 the short addressing modes. 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 store frequently used static addresses in registers. Unless a program requires less than 64K bytes, static addresses must be 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 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

```
p:
        pushl   %esi
        movl    $f,%esi
        pushl   $1
        call    *%esi
        popl    %ecx
        pushl   $2
        call    *%esi
        popl    %ecx
        pushl   $3
        call    *%esi
        popl    %ecx
        pushl   $4
        call    *%esi
        popl    %ecx
        popl    %esi
        ret
-----------
28 bytes
```

UNIX PCC

```
p:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   $1
        call    f
        popl    %ecx
        pushl   $2
        call    f
        popl    %ecx
        pushl   $3
        call    f
        popl    %ecx
        pushl   $4
        call    f
        popl    %ecx
        leave
        ret
-----------
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. 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 a move from one register to another, followed by a move in the reverse direction.

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. Speed Optimizations

The speed optimizations are selected by the -O compile time option. This increases the speed of the program but usually at the cost of making the program larger.

Programs which execute for long periods of time execute millions or billions of instructions. Since most programs consist of thousands or tens 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 -O compile time option should only be used on modules in which most processing occurs in loops. If -O is not specified, the compiler does everything it can to reduce program size. The -O compile time option will sacrifice program size to increase the performance of loops. It will allocate nearly all of the registers to the variables and temporaries used in the innermost loop of a function. This will prohibit them from being used for variables which are used elsewhere. If -O is specified, the compiler could go to a great deal of work to optimize some loop which is rarely executed. This would result in the program getting larger but no faster!

The meaning of -O is different in C-386 than the meaning of -O in PCC. In PCC, -O makes programs both smaller and faster by running a separate peephole optimization phase. The only reason not to use -O with PCC is to save 10% in compilation time. All of the optimizations done by -O under PCC are done by default under C-386. The -O compile time option calls for more optimizations (primarily involving loops). Since these optimizations will increase the size of the program but may only make a small increase in speed in some programs, it is recommended that -O be removed from UNIX makefiles when converting to C-386. Only after examining the code to determine if the loop optimizations will improve program performance (or after experimentation) should -O be used in production code.

4.3.1. 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.

4.3.2. 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.

The -X54 (dbmemoryoptimizations) compile time option performs more advanced loop invariant analysis. This compile time option is set implicitly by -O2 (which is the recommended method for invoking these optimizations). The -X54 compile time option is separate from -O because it will only work on algorithmic programs. That is, programs in which the compiler is in complete control of the processor and memory. It can cause programs which depend on things in the outside world such as interrupts, UNIX signals, I/O registers and shared memory to fail to operate correctly.

The -X54 compile time option causes the loop optimizer to make an extra check for loop invariants. If there are no writes to memory in a loop (and no calls that might write into memory) the value of any memory location will be assumed to be invariant within the loop. Therefore the value of a global variable (or array element) may be removed from the loop.

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.

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 other 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 maximum 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 exact 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 maximum alignment optimal for any component data type. The compiler always aligns parameters and local variable within the stack at an optimal offset from the beginning of the frame. The compiler always rounds up the size of the frame to the maximum optimal alignment of the 80386. If the initial stack pointer is aligned to the maximum optimal alignment of the 80386 and if the program involves no explicit (or only correct) 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. 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.

5.6. 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.

5.7. Expression Evaluation Order

The C-386 expression evaluation order is not identical to the evaluation order of other C compilers, although it may appear similar. The C Language specification permits expressions to be evaluated in any order, therefore programs which depend on the evaluation order are illegal. Unfortunately many illegal programs have gone undetected for years because they have only been compiled with one compiler.

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.

5.8. 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 *
#define O 1
x/**/y A comment */
int val;
main()
{
   va/**/O = 1;
}
```

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

```
/* A comment */
int val;
main()
{
   val = 1;
}
```

Due to the one pass nature of C-386 it is not possible for its builtin preprocessor to manufacture a token such as /* or an identifier as in the example above. 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.

5.9. 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.

5.9.1. 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.

5.9.2. 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.

5.9.3. 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).

5.9.4. -O2 Bugs

The -O2 and -X54 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 -O2 and -X54 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 -O2 and -X54 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 -O2 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. When the memory location is changed by an external device, under -O2, the loop will not stop!

5.9.5. 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 the variable. This may cause a source level debugger to give incorrect results. If you ask for the value of a variable at a point at which the variable is about to be assigned into, the compiler may have temporarily assigned that register to some other purpose. Always check results after they are assigned to, 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.

5.10. 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 200 Kbytes just for the program (and that is after using a Green Hills compiler to compile it). It is designed to work best when it has at least 700 Kbytes 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 function of less than 100 lines should not cause memory size problems. Procedures which are more than 1000 lines can require more than a megabyte of memory to compile.

5.11. 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(e), 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-386 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-386.

If you want to use the compiler in an environment other than the one that was intended, or if you have unusual requirements, you may find that the default options are not what you want. The compiler invocation documentation that you have 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.

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.

-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.

-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.

-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 (4.2BSD 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 Optimize the program to be as fast as possible even if it is necessary to make the program bigger. This compile time option often causes confusion among people familiar with PCC. To PCC, -O means make the program smaller and faster. One wonders why this is not the default since the -O compile time option only slows down compilation by 10%. For C-386, the default is to perform all of the optimizations performed by PCC. The -O compile time option will perform further optimizations which may make the program faster but larger. It is counter productive to specify -O on code which is rarely executed as it will make the whole program larger but no faster. Therefore, when converting to C-386 remove the -O option from makefiles and command files! After experimenting with a program it is possible to discover which modules benefit from -O and which ones do not.

-O2 Allow 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.

-R (UNIX Host only) Put all data in the text section.

-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.

-X18 Do not allocate programmer-defined local variables to a register unless they are declared register.

-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.

-X54 Inform the optimizer that there are no memory locations that can change value asynchronously with respect to the running program. "-O2" sets this compile time option. (see the discussion under -O2 above).

-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 code hoisting 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.

-X87 Turn off the optimization that deletes all code that stores into or modifies variables which are never read from.

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

-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

-X143 Generate Weitek floating point code instead of 80387.

-X153 Enable ANSI C extensions. Most of the 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. Occasionally useful in determining the cause of a compiler failure. Awful things may happen to you if you use this option.

-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.

-X213 Use Weitek software emulation library, requires -X143 to work.