

Green Hills Software User's Manual

C-68000

C Compiler for the 68000/10/20

January 1, 1987

Green Hills Software, Inc.

Copyright © 1983,1984,1985,1986,1987 by Green Hills Software, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Green Hills Software, Inc.

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

UNIX is a trademark of Bell Laboratories.
DEC, VAX, and VMS are trademarks of Digital Equipment Corporation.
4.2BSD is a trademark of the Board of Regents of the University of California at Berkeley.
C-68000 is a trademark of Green Hills Software, Inc.

The information in this publication is subject to change without notice.

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 68000/10/20, 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-68000.

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-68000 matches other C compilers.

68000/10/20 Target

The 68000/10/20 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-68000 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-68000

This chapter tells you about difficulties that you may encounter in moving a program developed with another compiler to C-68000. 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-68000 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-68000. 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-68000 does not come standard with your computer system, the Host documentation includes Install-

tion Instructions that tell you how to install C-68000 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-68000 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-68000. 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.

68000/10/20 Architecture Manual

You may need to know details of the instruction set and register organization of the 68000/10/20 in order to debug your programs. If you do not already have a 68000/10/20 Architecture Manual, you should contact the manufacturer of the 68000 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-68000 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-68000 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-68000.

If you have any other version of C-68000, 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-68000 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-68000 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-68000 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-68000 supports signed and unsigned bit fields. All versions of the 68000 PCC that we have examined only support unsigned bit fields. Therefore, for compatibility with these implementations of PCC bit fields are unsigned by default, even if a signed type is used to declare the field. 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-68000.

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 Ritchie 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-68000 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-68000 is somewhat different than the asm construct in PCC. In C-68000 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-68000 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-68000 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-68000 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

68000/10/20/30 Target

3.1. Introduction

There are two common target environments for C-68000: UNIX and stand alone. These environments are different but they are supported by the same compiler. In the UNIX environment, C-68000 generates code for derivatives of the MIT and AT&T UNIX ports. In the stand alone environment, C-68000 generates code for a Motorola compatible assembler, such as the ones supplied by OASYS and Microtec.

3.2. 68010 Target Environment

C-68000 generates code for the 68000 by default. The compile time option -X12 causes it to generate code for the 68010 instead. The major difference relates to how condition codes are saved and restored. This code is (fortunately) rarely used. It should normally only occur in a complex routine.

3.3. 68020 Target Environment

C-68000 generates code for the 68000 by default. The compile time option -X98 enables 68020 code generation. You should also have the -X12 compile time option on when you use it.

The -X122 compile time option generates 68020 instructions as binary words in the assembly code so that the code can be assembled by a 68000 or 68010 assembler.

The -X130 (full 68020) compile time option generates code for the 68020 with no consideration for compatibility with old code compiled for the 68000 or 68010.

The -X140 (68020 alignment) compile time option aligns data in memory to be optimal for the 68020. This alignment is incompatible with the default alignment used by most 68000 and 68010 compilers (including C-68000).

If -X130 or -X140 is used in compiling any libraries or any modules of your program it must be used in compiling ALL libraries and modules.

The -X99 compile time option specifies to generate code for the 68881 floating point processor.

3.4. Memory Allocation

The 68000 memory is byte addressed.

Bytes are ordered as on the IBM/370 and Z8000, opposite of the 8086, VAX, Clipper, and NS32032. The most significant byte is at the lowest address. Bits are numbered with bit zero as the least significant bit. This is the so-called Big-Endian format.

The character encoding is ASCII.

The stack is always aligned on a 2 byte boundary. On the 68020, it is always aligned on a 4 byte boundary (-X140). All complex data types in memory are 2 byte aligned, and on the 68020 they are 4 byte aligned.

Every packed field must be fully contained in four or fewer bytes. Packed fields are allocated starting at the

most significant bit. No effort is made to reshuffle bits within a packed structure for greater access efficiency.

Data Type	Size	Alignment	
-----	----	-----	-----
		68000/10	68020(-X140)
int	32	word	long
long	32	word	long
*	32	word	long
short	16	word	word
char	8	byte	byte
float	32	word	long
double	64	word	long
unsigned	32	word	long
unsigned char	8	byte	byte
unsigned short	16	word	word
enum (default)	32	word	long
enum (option)	8,16,32	varies	varies

3.5. Calling Conventions

Arguments are evaluated from right to left. Each scalar argument is extended to a 32 bit value and then it is pushed onto the stack. Each floating point argument is extended to a 64 bit value and then it is pushed onto the stack. All other arguments are extended to a multiple of 2 bytes and then pushed onto the stack (the extra bytes are at the high addresses). Under the 68020 alignment (-X140) compile time option, all other arguments are extended to a multiple of 4 bytes and then pushed onto the stack (the extra bytes are at the high addresses).

Scalar values are returned in d0. When the size of the return value is specified as less than 32 bits only the required number of bits should be depended on in d0. The compiler presently fills the additional high order bits for compatibility with other compilers. This is controlled by a compile time option (-X19). In the future, the default setting may be to not fill the high order bits.

Floating point values are returned in d0/d1. They are always double precision. When the 68881 option is enabled with the full-68020 (-X130) compile time option, floating point values will be returned in fp0. Otherwise they will returned in d0/d1 to maintain compatibility.

A call to a function uses either the “jsr” or “bsr” instruction. A return from a function uses the “rts” instruction.

The d0, d1, a0, and a1 registers (and fp0 and fp1 if a 68881 is present) are assumed to be destroyed by a call to a function. 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 several methods of generating the local stack frame for a function. If a function requires no local storage, no frame is generated and all access to variables and parameters on the stack use the a7 relative addressing mode. A6 is not used. If only one register is saved, it is pushed on the stack at the entry to the function and popped at the exit. If more than one register is saved, the moveml instruction is used to push them on entry and pop them on exit. If a 68881 is present, the floating point registers are pushed in a similar fashion.

If a function requires local stack storage, or the -g, -ga or -pg compile time option is specified, the procedure will do a “link” instruction on function entry and an unlink instruction on exit. Accesses to parameters or local stack storage will be made with the a6 relative addressing mode. The stack frame size is always rounded up to the nearest 4 bytes. On the 68020 this allows for a longword aligned stack.

Following the return of a function, any arguments pushed on the stack are (at least conceptually) removed. If several function calls follow each other in a basic block, the stack will continue to grow until the end of the block. If less than eight bytes must be removed from the stack, a “addq1 #xx,sp” instruction is used. If more than eight bytes,

but less than 32768 bytes must be removed from the stack a “`lea x(sp),sp`” instruction is used. If more than 32768 bytes must be removed from the stack, an “`addl #x,sp`” instruction is used.

A function frame that is in excess of 32K bytes is considerably less efficient than a smaller frame. The MC68000/010 addressing modes are designed for 16 bit offsets. When offsets are larger than that, every machine instruction with a large offset will require from 1 to 3 extra instructions to bring the offset within range.

3.5.1. Stack Probes

The compile time option `-X11` generates a stack probe instruction at the start of a function before the frame is allocated. The stack probe is a “`tst.b -x(sp)`” instruction, where “`x`” is about 200 bytes greater than the stack frame size. The “`tst`” instruction is restartable in the event of a trap. This allows a memory management unit to expand the stack on demand. A stack probe instruction will not be generated for a function with a small frame and no calls. This works because the stack probe is generally about 200 bytes ahead of actual usage.

3.6. Local variables

C-68000 may allocate eight bit and sixteen bit local variables to registers. In this case, only the low order bits of the register are valid, and the high order bits are undefined.

3.7. UNIX Version

When configured for UNIX output, C-68000 generates UNIX (MIT) format assembler code. There are several options for the different nuances of the UNIX (MIT) assembler.

Floating point is UNIX (MIT) format by default, most significant byte at the lowest address. This format is similar to the default VAX floating point format. The compiler will optionally generate IEEE 32/64 format (`-X22`).

3.7.1. Symbolic Debugger Support

The `-g` option generates debugging information pseudo-ops in the assembler language output. This allows the program to be debugged by a source level debugger such as “`cdb`”. Berkeley 4.1, 4.2 and System V conventions are supported. A separate compile time option must be used to tell the compiler which of these three formats is desired. The correct option for your system should be set by default. If you want to change it, refer to the “Compile Time Options” chapter. Most MIT format assemblers and linkers do not understand `.stab` entries in the object files. Most AT&T format assemblers accept System V directives.

3.8. Motorola Assembler Version

The compile time option (`-X35`) causes C-68000 to generate Motorola format assembler code. This assembler is quite different than the assembler language supported on most 68000 UNIX systems. The Motorola Assembler is the most common assembler language used in cross development environments.

By default, the compiler will pass through the case of letters in identifiers to the assembler. Some Motorola compatible assemblers will only accept upper case identifiers. The `-X21` compile time option causes C-68000 to convert all identifiers to upper case when they are output in the assembler code.

The `-X14` compile time option causes C-68000 to prepend an underscore to all identifiers when they are output in the assembler code. Some Motorola compatible assemblers will not accept an underscore in an identifier. If the `-X14` and `-X20` compile time options are both specified, C-68000 will prepend identifier names with a period when they are output in the assembler code.

Floating point is MIT format by default, most significant byte at the lowest address. This format is similar to the default VAX floating point format. The compiler will optionally generate IEEE 32/64 format (`-X22`).

The `-g` option generates human readable comments in the assembler code to specify line numbers.

The C language implies the ability to reference variables from multiple modules, with all modules defining the variable in the same way. Most Motorola-compatible assemblers do not support large numbers of named SEGMENTS. Therefore, the following conventions should be observed:

```
extern int x;           /* will emit an XREF to x*/
```

```
int x;           /* will emit XDEF of x, with a DCB.B size(x),0*/
int x = 5;       /* will emit an XDEF of x, with a DC.L 5 */
```

A convenient programming method is to declare each variable in one module, and declare it extern in all other modules. Alternatively, if your assembler supports named sections, the -X93 (dbnamedsections) compile time option may be used. This should also be used with Fortran, if assembler support is available.

Variables will exist in SECTION 14 by default; if the -R compile time option is used, initialized variables will be placed in SECTION 13. This is useful if you want your initialized variables to be separate, e.g. in ROM.

Variables can be placed in other SECTIONs by defining the preprocessor symbols CODESEG, INITSEG, and DATASEG with the desired values. This can be conveniently done from the command line. It can also be done in-line, in which case it will take effect the next time a SECTION statement is output.

There is no default initialization.

The compiler generates obscure numeric names for static variables in C, rather than refraining from XDEFing them.

3.9. Common Compile Time Option Combinations

The compile time options for several common environments are listed below. If you encounter difficulty in conjunction with your assembler, try some of the following combinations.

3.9.1. 68000 Cross Development

Use compile time options -X20, -X29, -X35 with the OASYS or Microtec Assembler. Use -X22 if you will be using IEEE arithmetic, -X12 if you will be operating on a 68010, and -X23 if you have a very small stack.

3.9.2. 68020/68881 Cross Development

Use compile time options -X20, -X29, -X35, -X22, -X98, -X99, -X122, -X129 with the OASYS or Microtec Assembler. Do not use -X140 if you wish to return alignment compatibility with the old 68000 compiler. Use -X130 if you want return values in fp0. These compile time options should be used with ALL of your modules if they are used with any modules.

3.9.3. 68010 Unisoft System V Release 1

Use compile time options -X11 -X12 -X14 -X22 -X42 -X95 -X96 -X97 -X138 -X74.

3.9.4. Sun Microsystems Series 2

Use compile time options -X12 -X14 -X22 -X42 -X114.

3.9.5. Motorola/AT&T System V (e.g. AT&T PC)

Use compile time options -X12 -X74 -X22 -X39 -X42 -X92 -X97. Note that some versions of System V use routine names of the form “lmul__” etc. for 32 bit arithmetic, others use “lmul%%”. You may need to create a glue routine, or process the output with a sedfile.

3.10. Runtime Libraries

The compiler will generate out of line calls for certain operations that are not supported by the MC68000/010, such as 32 bit divides and floating point. The source for this part of the runtime library is provided. The calling con-

ventions and routine names are similar to those of SUN Microsystems.

3.10.1. Motorola and MIT library Entry Points

The arithmetic library routines accept parameters in d0/d1 first, and use the stack if that is insufficient. Return values are placed in d0 or d0/d1.

	Name	Arguments	Result	Function
	-----	-----	-----	-----
	ulmult	d0,d1	d0	(signed multiplication)
	ldivt	d0,d1	d0	(signed division)
	uldivt	d0,d1	d0	(unsigned division)
	lmodt	d0,d1	d0	(signed remainder)
	ulmodt	d0,d1	d0	(unsigned remainder)
	fnegi	d0,d1	d0,d1	(DP negation)
	fabsi	d0,d1	d0,d1	(DP absolute value)
*	fsinglei	d0,d1	d0	(DP->SP)
*	fdoublei	d0	d0,d1	(SP->DP)
	ffixi	d0,d1	d0	(DP-> Integer)
	fflti	d0	d0,d1	(Integer -> DP)
*	ffltis	d0	d0	(Integer -> SP)
	fmuli	d0/d1,Stack	d0/d1	(DP*DP -> DP)
	faddi	d0/d1,Stack	d0/d1	(DP+DP -> DP)
	fdivi	d0/d1,Stack	d0/d1	(DP/DP -> DP)
	fsubi	d0/d1,Stack	d0/d1	(DP-DP -> DP)
	fcmpi	d0/d1,Stack	CCR	(DP-DP -> CCR)
	fsmuli	d0/d1	d0/d1	(SP*SP -> SP)
	fsaddi	d0/d1	d0/d1	(SP+SP -> SP)
	fsdivi	d0/d1	d0/d1	(SP/SP -> SP)
	fssubi	d0/d1	d0/d1	(SP-SP -> SP)
	fscmpi	d0/d1	CCR	(SP-SP -> CCR)
	maovfsub			Overflow destination
	madv0sub			Floating divide by zero

* No actual code is provided for these routines.

3.10.2. Floating Point, IEEE support

Floating point constants will be in MIT format by default, which is what MIT format assemblers expect. The “IEEE” (-X122) compile time option will cause constants to be generated in 32 and 64 bit IEEE format, and will cause the fsinglei and fdoublei instructions to be used where conversions are performed. All operations are performed in double precision.

These routines obey the customary register conventions; a0/a1/d0/d1 can be expected to be destroyed on a function call except for return values.

Chapter 4

Optimization

4.1. Introduction

C-68000 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-68000 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-68000 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-68000 is much better than most C compilers in its register allocation.

In the following example, C-68000 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-68000	UNIX PCC
-----	-----
proc:	proc:
movl d2,sp@-	link a6,#-12
moveq #1,d2	moveml #128,a6@(-12)
	movl #1,a6@(-4)
	jra .L15
.L7:	.L20001:
jbsr f	jsr f
addql #1,d2	addql #1,a6@(-4)
	.L15:
moveq #10,d0	cmpl #10,a6@(-4)
cmpl d2,d0	jlt .L20001
bgt .L7	movl #1,a6@(-8)
moveq #1,d2	.L20003:
.L4:	jsr g
jbsr g	addql #1,a6@(-8)
addql #1,d2	cmpl #10,a6@(-8)
moveq #10,d0	jlt .L20003
cmpl d2,d0	moveml a6@(-12),#128
bgt .L4	unlk a6
movl sp@+,d2	rts
rts	
.data	
allocations for f	
d2 i	
d2 j	
-----	-----
38 bytes	76 bytes

The savings by C-68000 can be summarized as:

Put i and j in d2	16 bytes
Replace moveml by movl	8 bytes
Delete link/unlk	6 bytes
Use moveq	6 bytes
rotate loop	2 bytes

4.2.1.1. C Local Static Variables

Since C-68000 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-68000 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-68000 does not set up a frame pointer in each function. C-68000 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.

When only one or two registers need to be saved in a function, C-68000 may generate code to save registers one at a time instead of with a general save under mask instruction. This can result in considerable savings in function entry and exit code.

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. Stack Adjustment Coalescing

In many programs, function calls require a substantial amount of code. Each time a function is called the arguments are pushed on the stack, the function is called, and when the function returns the arguments are removed from the stack by adding to the stack pointer. In many programs a substantial savings in code size is realized by the Green Hills compiler by not adding to the stack pointer after each call. Instead, the total amount of space that needs to be removed is accumulated until some occurrence such as a branch forces the compiler to adjust the stack.

This optimization may cause a program to use more stack space than it otherwise would have. The -X23 (dbpopalot) compile time option forces an adjustment of the stack after each call. This stops stack frames from growing too large at the expense of generating more code.

See the example under Static Address Elimination for a use of Stack Adjustment Coalescing.

4.2.5. Static Address Elimination

A valuable optimization performed by C-68000 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-68000	UNIX PCC
-----	-----
p:	p:
movl a2,sp@-	link a6,#0
movl #f,a2	moveml #0,a6@(0)
pea 1:w	movl #1,sp@-
jbsr a2@	jsr f
	addql #4,sp
pea 2:w	movl #2,sp@-
jbsr a2@	jsr f
	addql #4,sp
pea 3:w	movl #3,sp@-
jbsr a2@	jsr f
	addql #4,sp
pea 4:w	movl #4,sp@-
jbsr a2@	jsr f
	addql #4,sp
lea sp@(16),sp	moveml a6@(0),#0
movl sp@+,a2	unlk a6
rts	rts
-----	-----
40 bytes	76 bytes

The savings by C-68000 can be summarized as:

Static Address Elimination	10 bytes
Use movl instead of moveml	8 bytes
Use pea instead of movl #	8 bytes
Eliminate link/unlk	6 bytes
Stack Adjustment Coalescing	4 bytes

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, where the following instruction did not use the condition codes. C-68000 keeps the full life-time information available at this stage, so many optimizations which would be unsafe for a peephole optimizer can be performed. For this reason we refer to them as “local” optimizations.

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-68000 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-68000. 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-68000. 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. In Line Multiplication

The 68000 has no 32 bit by 32 bit multiply instruction, so most compilers call a library routine to do a 32 bit multiply. When transporting code, from larger machines to the 68000 a severe performance degradation can occur on programs which do a lot of 32 bit arithmetic. On most mainframes a 32 bit multiply can be expected to take from 3 to 20 times as long as a 32 bit add. But on a 68000, under most compilers, the ratio is greater than 100 to 1. Under the -O compile time option, C-68000 expands 32 bit multiplications into in line code. This reduces the multiply to add ratio to about 20 to 1. The size of the in line multiplications is somewhat larger than a call to a library routine, and the compiler will run more slowly and take more memory if a function has a very large number of 32 bit in line multiplications, but the in line multiply is about 5 times faster than the typical library routine.

To further avoid 32 bit multiplications, C-68000 will use 16 bit arithmetic for array index calculations if the compiler can determine that the computation will not overflow. Sometimes the declaration of an array is only a dummy declaration. The array might be entirely different than the size specified. To avoid any problems, the compiler will assume that the programmer is lying about the dimension of an array if it is specified to be of length 0 or 1. In these cases, the compiler will use 32 bit arithmetic. If the dimension is specified to be anything greater than 1, then it is assumed to be legitimate.

Multiplications by constants are also dealt with to reduce the use of 32 bit multiplications. If the constant is a 16 bit value then an in line 16 by 32 bit multiply is done instead of a 32 by 32 bit multiply. If the constant is of the form $2^n + 2^m + 1$ (or simpler) it is expanded in line as shifts and adds.

4.3.2. 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.

4.3.3. 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.

4.3.4. 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-68000

Some programs which appear to compile and operate correctly when compiled with other C compilers, may not operate correctly when compiled with C-68000. The C Language specifications define legal programs in such a way that legal programs will always work with all C compilers, including C-68000. 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-68000.

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-68000 program.

The implementation of each Green Hills C Compiler is the same for each Green Hills Target. Therefore, legal programs written in C-68000 can be moved to any other Green Hills C Compiler.

C-68000 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 descendants 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-68000 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-68000 are defined in the 68000/10/20 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 68000/10/20. If the initial stack pointer is aligned to the maximum optimal alignment of the 68000/10/20 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 68000/10/20, all global data references will be optimal.

C-68000 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 68000/10/20 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-68000 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 68000/10/20 Target chapter for details of the C-68000 calling conventions.

Expressions with side effects, such as function calls and the operators “++”, “--”, “+=”, etc., may be executed in a different order by C-68000 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-68000 may allocate some pointer variables not declared “register” to registers. This may allow C-68000 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-68000.

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-68000 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++)” will usually evaluate differently under PCC than under C-68000.

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-68000.

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 va1;
main()
{
    va/**/O = 1;
}
```

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

```
/* A comment */
int va1;
main()
{
    va1 = 1;
}
```

Due to the one pass nature of C-68000 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-68000 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-68000, 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-68000, 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. The compile time option `-X125` automatically activates the `-X18` compile time option for any function in which there is a call to a function called “`setjmp`”.

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-68000 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-68000 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-68000 may not allocate unused variables. Some programs depend on knowing that certain variables will be allocated in memory. C-68000 will allocate certain variables to registers that PCC and other compilers would always allocate to memory. Programs compiled with C-68000 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-68000 is an advanced optimizing compiler. It is much better than the current generation of “optimizing” microprocessor C compilers. In accordance with its greater capability it requires more memory. C-68000 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-68000 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 68000/10/20 code generator. The code generator produces an internal representation of the 68000/10/20 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-68000 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-68000 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-68000.

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. The term “Motorola Assembler” refers to the MC68000 assembler defined by Motorola. This assembler is quite different from the assembly language available on most MC68000 based UNIX systems.

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.

- 20 (UNIX Host only) Produce MC68020 code.
- 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.
- DCODESEG=xx (Motorola Assembler only) Place the code in section “xx”. This may also be set with a #define statement. “xx” is passed directly through to the assembler; by default it is 9.
- DDATASEG=xx (Motorola Assembler only) Place data in SECTION “xx” rather than SECTION 14.
- DINITSEG=xx (Motorola Assembler only) If the -R option is enabled, place initialized data in SECTION xx rather than 13.

- 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.
- f (UNIX Host only) Generate code for the MC68881 floating point coprocessor. The default is for all functions supported by the MC68881 to be performed in line. The option -Z129 will disable this feature, e.g. if you want a call to “sin(3M)” to go to the library.
- fsky Generate code for Sky floating point board. If this option is used for any module, it must also be used when linking.
- 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-68000, 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-68000 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.

In addition, all scalar multiplies are performed inline, and larger block moves are performed with sequential moves rather than an inline loop.
- 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.
(Motorola Assembler only) Place explicitly initialized variables in a different segment from uninitialized variables so that they can be placed in ROM.
- 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 Use a “mov #0,x” rather than a “clr x” to non-stack addresses.
Also use the output suffix “.a68” rather than “.s”.
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.
- X11 Generate a stack probe at the beginning of each routine which uses more than 40 bytes of stack space, or which calls a routine.
- X12 Generate 68010 code rather than 68000 code.
- X13 Suppress code generation. An output file of zero length will be created.
- X14 Prepend an underscore to all programmer defined names.
- X16 (UNIX Target only) Use “.=.n” to allocate n bytes of blank space. The default is “.blkb”.
- X18 Do not allocate programmer-defined local variables to a register unless they are declared register.
- X19 Functions which return less than a full word (e.g. char x()) will not set the high order bytes.
- X20 (Motorola Assembler only) Replace an underscore as the first character of an identifier with “.” (period), for assemblers which require this.
- X21 (Motorola Assembler only) Map all identifiers to upper case, for strict Motorola compatibility.
- X22 Use IEEE format floating point constants, and generate calls when numbers change size. By default, use VAX format floating point constants.
- X23 Always adjust the stack after every function call.
- X25 Output routine sizes on the error output as the compilation proceeds. Useful for monitoring compilation progress.
- X26 (MIT version only) I&D space separation; dispatch tables will be located in data space.
- X29 Suffix the output file with “.asm” rather than “.s”.
- X30 Generate inline calls to the Sky floating point unit. Uses hard coded constants for the device address.
- 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.
- X35 Generate Motorola Assembler output code; this should be used with most cross development assemblers.
- X36 Presume that all “switch” statements will take a defined label. No check is made that the index is in the range of the smallest to largest value specified. If the index is out of range the results will almost certainly be disastrous. This generates somewhat faster and smaller code.

- X37 Emit a warning when dead code is eliminated.
- X38 When using the MIT/VAX floating point format, generate calls to a rounding routine when changing from double to single precision.
- X39 Do not move frequently used procedure and data addresses to registers.
- X41 For use with -X25, print out more statistics.
- X42 Do not generate calls to single precision routines, except for conversions to and from double. Use this if you do not have single precision floating point arithmetic support.
- 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.
- 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-68000.
- 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.
- X89 (Definicon/Dave Rand only) Pack structures with no space between members (even if it makes them impossible to access!)
- X90 Turn off code hoister. Same as -X80.
- X92 (UNIX Target only) Generate AT&T format assembly code. This is the format used in AT&T System V as well as the AT&T Software Generation System.
- X93 (Motorola Assembler only) This option specifies that the Motorola compatible assembler being used supports Named Sections for the definition of C global variables and Fortran COMMON variables. Many Motorola compatible assemblers do not support this feature.
- X94 Remove all garbage from the top of the stack at the end of each basic block (before each possible transfer of control).
- X95 (UNIX Target only) Use names of the form “.Lnnn” for temporaries not “Lnnn”.
- X96 (UNIX Target only) Generate “space” instead of “blkb” to allocate space for data.
- X97 (UNIX Target only) Do not generate the “.ascii” directive, use a sequence of bytes instead.
- X98 (Motorola Assembler only) Generate 68020 code. See also -X122.
- X99 (Motorola Assembler only) Generate 68881 code. You must use -X22 with this option. See also -X122 and -X129.
- X100 (CRDS only) “handle” flag for CRDS; use c.ret to exit routine rather than unlk/rts.
- X100 to -X109 Customer defined options.
- X101 (CRDS only) Charles River Data Systems Target.
- X102 (CRDS only) Activate CRDS mods in c68u.
- X103 (CRDS only) Whitesmith calling conventions
- 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

- X120 (Motorola Assembler only) Perform the Hunter and Ready code translations on the assembly language code.
- X121 Disable the machine operand specific portions of the local (peephole) optimizer. This option, or -X9, should be tried if you have reason to believe the compiler is incorrectly compiling a program.
- X122 Pre-assemble 68020 instructions, so that 68020 instructions can be processed by a 68000 assembler and linker.
- X123 (Edge only) Target is an Edge Vanguard.
- X124 (Edge only) Generate Edge Vanguard floating point.
- X125 Activate -X18 if there is a call to setjmp.
- X129 Use the 68881 instructions for evaluation of sin, cos, etc. This is done by intercepting the calls to these routines; you can still define “sin” etc., however the code will not call it.
- X130 (MC68020 only) Full 68020 mode. This is reserved for indicating that the code will only run on a 68020 system. Presently it implies that floating point values come back in fp0.
- X133 (Motorola Assembler only) Use the non-filling declaration “DS.B” rather than “DCB.B”. This prevents large arrays from taking up large amounts of output module size (and download time). However, they will not be initialized.
- X136 (UNIX Target only) Target the output for an Integrated Solutions Inc. assembler.
- X137 (Edge only) Peephole optimization for Edge Vanguard.
- X138 (UNIX Target only) Do not use the “lcomm” (local common) directive.
- X140 Use 68020 alignment rules. If you plan to upgrade to a 68020 system, it is good idea to use this option for ALL compilations. If this option is used on any library or any module of your program it must be used on ALL modules of your program.
- X142 (Edge Only) Target Edge assembler without invoking other Edge modifications. (Heurikon only) Use in conjunction with -X120 to prevent a4 and d7 from being used in fixup.
- 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.

Table of Contents

1. Overview	3
1.1. How to Use This Manual	3
1.2. Related Documentation	3
2. The C Language	5
2.1. Introduction	5
2.2. Additions to the basic C Language	5
2.2.1. Preprocessor	5
2.2.2. Backslash v	5
2.2.3. void type	5
2.2.4. <code>__LINE__</code>	5
2.2.5. <code>__FILE__</code>	6
2.2.6. Structure and Union Extensions	6
2.2.7. Enumeration Type	6
2.2.8. The VARARGS(3) Facility	7
2.3. Bit Fields	7
2.4. Extern and Common	8
2.5. Unsigned Char and Unsigned Short Int	8
2.6. asm Statement	8
2.7. C Runtime Libraries	8
3. 68000/10/20/30 Target	10
3.1. Introduction	10
3.2. 68010 Target Environment	10
3.3. 68020 Target Environment	10
3.4. Memory Allocation	10
3.5. Calling Conventions	11
3.5.1. Stack Probes	12
3.6. Local variables	12
3.7. UNIX Version	12
3.7.1. Symbolic Debugger Support	12
3.8. Motorola Assembler Version	12
3.9. Common Compile Time Option Combinations	13
3.9.1. 68000 Cross Development	13
3.9.2. 68020/68881 Cross Development	13
3.9.3. 68010 Unisoft System V Release 1	13
3.9.4. Sun Microsystems Series 2	13
3.9.5. Motorola/AT&T System V (e.g. AT&T PC)	13
3.10. Runtime Libraries	13
3.10.1. Motorola and MIT library Entry Points	14
3.10.2. Floating Point, IEEE support	14
4. Optimization	15

4.1. Introduction	15
4.2. General Optimizations	15
4.2.1. Register Allocation by Coloring	15
4.2.1.1. C Local Static Variables	16
4.2.2. Memory Allocation	16
4.2.3. Entry and Exit Code Optimization	17
4.2.4. Stack Adjustment Coalescing	17
4.2.5. Static Address Elimination	17
4.2.6. Peephole Optimizations	18
4.3. Speed Optimizations	18
4.3.1. In Line Multiplication	19
4.3.2. Loop Rotation	19
4.3.3. Loop Invariant Analysis	19
4.3.4. Strength Reduction	20
5. Porting Programs to C-68000	21
5.1. Compatibility with other Green Hills Compilers	21
5.2. Word Size Problems	21
5.3. Byte Order Problems	21
5.4. Alignment Requirements	22
5.5. Floating Point Range and Accuracy	22
5.6. Assembly Language Interfaces	22
5.7. Expression Evaluation Order	22
5.8. C Preprocessor Incompatibilities	23
5.9. Illegal Assumptions about Compiler Optimizations	23
5.9.1. Problems with Setjmp and Longjmp	23
5.9.2. Implied register usage	24
5.9.3. Memory Allocation Assumptions	24
5.9.4. -O2 Bugs	24
5.9.5. Problems with Source Level Debuggers	25
5.10. Problems with Compiler Memory Size	25
5.11. Detection of Portability Problems	25
6. Compile Time Options	26