intel

i860™ Microprocessor Fortran Compiler User's Guide

Version 1

January 1990

240730-001

Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

DISCLAIMER

GREEN HILLS SOFTWARE, INC. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MECHANTABILITY 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 or Green Hills Software, Inc. to notify any person of such revision or changes.

Intel retains the right to make changes to this document at any time, without notice.

Contact your local sales office to obtain the latest document before placing your order.

The following are trademarks of Intel Corporation and may only be used to identify Intel products.

376, 386, 387, 486, 4-SITE, Above, ACE51, ACE 96, ACE186, ACE196, ACE960, BITBUS, COMMputer, CREDIT, Data Pipeline, ETOX, Genius, i, i486, i860, ICE, iCEL, ICEVIEW, iCS, iDBP, iDIS, ^{2}CE, iLBX, iMDDX, iMMX, Inboard, Insite, Intel, intel, Intel386, intelBOS, Intel Certified, Intelevision, inteligent Identifier, inteligent Programming, Intellec, Intellink, iOSP, iPDS, iPSC, iRMK, iRMX, iSBC, iSBX, iSDM, iSXM, Library Manager, MAPNET, MCS, Megachassis, MICROMAINFRAME, MULTIBUS, MULTICHANNEL, MULTIMODULE, MultiSERVER, ONCE, OpenNET, OTP, PROMPT, Promware, QUEST, QueX, Quick-Erase, Quick-Pulse Programming, Ripplemode, RMX/80, RUPI, Seamless, SLD, SugarCube, UPI, and VLSiCEL, and the combination of ICE, iCS, iRMX, iSBC, iSBX, iSXM, MCS, or UPI and a numerical suffix.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

MULTIBUS is a patented Intel bus.

CHMOS and HMOS are patented processes of Intel Corp.

Intel Corporation and Intel's FASTPATH are not affiliated with Kinetics, a division of Excelan, Inc. or its FASTPATH trademark or products.

UNIX is a trademark of AT&T Bell Labs.

MS-DOS and XENIX are a trademark of Microsoft Corporation.

OS/2 is a trademark of International Business Machines Corporation.

Additional copies of this manual or other Intel literature may be obtained from:

Intel Corporation
Literature Sales
P.O. Box 7641
Mt. Prospect, IL 60056-7641

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

Oasys Division (Sales & Support)
230 Second Ave.
Waltham, Ma. 02154
(617) 890-7889
890-4644

Table of Contents

Chapter 1
Overview

1.1. The Green Hills Fortran Documentation Set

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

1.2. User's Guide Structure

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

Overview

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

Language Features

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

Target

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

Optimization

The Optimization chapter gives detailed information about the optimizations used by Green Hills Fortran 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 Fortran

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

Interfacing Fortran and C

This chapter describes how to call C subroutines from a Fortran program, and vice versa.

Interfacing Pascal and C

This chapter describes how to call C subroutines from a Pascal program, and vice versa.

Fortran Runtime Library

This section describes the Fortran-860 runtime library.

Compile Time Options

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

Runtime Errors

This table lists the Fortran runtime errors.

Compile Time Errors

This table lists the Fortran compile-time errors.

Chapter 2
Language Features

2.1. F77 Compatibility Mode

Green Hills Fortran provides F77 compatibility via the -novms compile time option. On most systems, the default is VMS compatibility mode. Some of the compiler features and functions may differ, or not be available, in F77 compatibility mode. In this case, the information, and any F77 equivalents, will be listed in a notes section following the feature or function description.

2.2. VAX/VMS Fortran Language Extensions

The following list summarizes the VAX/VMS Fortran Language Extensions implemented in Green Hills Fortran. Note that on most systems, VMS compatibility mode (-vms) is the default. Some of the features listed may not be available when the compiler is executed in F77 compatibility mode (using -novms). Check with your system supplier to determine your compiler's default mode.

2.2.1. Line Formatting Extensions

Comments may be indicated by 'C', '!', or '*' in column 1
Comments may be indicated by '!' in the statement field
D in column 1 is a debug statement indicator
Continuation convention extension: 'tab' character followed by any of the characters '1' ... '9'
Allow up to 99 continuation lines
INCLUDE statement syntax is INCLUDE "pathname"
    Include files in text libraries are not supported
    The /LIST and /NOLIST options are not supported

2.2.2 Lexical Extensions

'$', '_' in identifiers
Identifiers up to 31 characters long
'nnn'O and 'nnn'X octal and hex integer constants
Radix 50 constants
Hollerith typeless constants

2.2.3. Declaration Extensions

BYTE data type
DATA statement in any place in program unit
IMPLICIT NONE statement
VOLATILE statement
VIRTUAL statement
Extended PARAMETER statement with additional operators and functions
"*n" size qualifier on function names and identifier declarations
Declare multifield records with STRUCTURE statement (no initialization allowed)
Single subscript in EQUIVALENCE of multidimensional array

2.2.4. Initialization Extensions

Initialization in type declaration statements
Initialize CHARACTER variables with integer values
Initialize static and COMMON storage to 0 if not specified (Unix only)

2.2.5. Expression Extensions

%VAL(), %REF(), %LOC(), accept and ignore %DESC
Logical type allowed in integer context in expressions
Logical operators apply (bitwise) to integers
Use non-integer type expression in integer context: convert to integer

2.2.6. Builtin Subroutine and Function Extensions

Degree-style trig functions
System subroutines: DATE,IDATE,ERRSNS,EXIT,SECNDS,TIME,RAN,MVBITS

2.2.7. Statement Extensions

DO WHILE statement
END DO statement
Extended range DO loops
Ampersand (&) or asterisk (*) for alternate return
OPTIONS statement with full VAX syntax. Ignore all options except /I4, /NOI4, /D_LINES

2.2.8. Input/Output Extensions

NAMELIST
ACCEPT statement
TYPE statement
ENCODE/DECODE statements
Accept full VAX/VMS syntax for OPEN, INQUIRE and CLOSE statements, but ignore most options
Allow 99 files open at the same time
IBM ('n) form of relative record specification

2.2.9. Format Extensions

O (octal) and Z (hex) Edit descriptors, including the form 'On.n'
H edit descriptor on input
Q edit descriptor
$ edit descriptor
Default field widths for IOZLFEDGA edit descriptors match data type
'$' and '\0' carriage control — mapped to Unix carriage control
Short field terminators to separate numeric data on input

2.2.10. Compiler complexity equal to or better than VAX/VMS Fortran

20	DO and IF statement nesting
255	Arguments in a CALL or function reference
250	Named COMMON blocks
8	Format group nesting
500	Labels in computed GOTO
40	Parentheses in nested expressions
10	Include file nesting
99	Continuation lines (with compiler option)
132	Source line length in characters
31	Identifier length
2000	Constant length, character and Hollerith
12	Constant length, radix-50
7	Array dimensions
250	Number of names in a NAMELIST group

2.2.11. Unimplemented VAX/VMS Fortran Extensions

REAL*16, COMPLEX*32
Indexed Files
Expressions in Format Statements (F<i+j>.<k–l>)
The DELETE statement for relative files
The REWRITE statement for relative files
Initialization of STRUCTUREs
The DEFINE FILE statement
The FIND statement

Chapter 3
Intel i860 CPU Target

3.1. Introduction

This chapter describes the Intel i860 CPU target environment for Fortran-860.

3.2. i860 Characteristics

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

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

Character encoding is ASCII.

The stack is always sixteen byte aligned.

Data Type	Size	Alignment
INTEGER (default)	32	32
INTEGER (-i2)	16	16
INTEGER*1	8	8
INTEGER*2	16	16
INTEGER*4	32	32
LOGICAL	32	32
LOGICAL*1	8	8
LOGICAL*2	16	16
LOGICAL*4	32	32
REAL	32	32
REAL*4	32	32
REAL*8	64	64
DOUBLE PRECISION	64	64
CHARACTER*1	8	8
CHARACTER*n	8*n	8*n
COMPLEX	64	32
COMPLEX*8	64	32
COMPLEX*16	128	64
DOUBLE COMPLEX	128	64

3.3. Compiler Output Format

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

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

3.4. Register Usage

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

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

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

3.5. Calling Conventions

The compiler supports two calling conventions (selectable by a compile time switch). The first is the calling convention specified in the 1989 i860 PRM. The second is that specified in the 1990 i860 PRM and are Standard Binary Specification (SBS) compliant.

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

3.5.1. 1989 i860 PRM Calling Convention

The following is the calling convention set forth in the subroutine or function Programmer's Reference Manual.

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

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

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

3.5.2 SBS

The following is the calling convention set forth in the i860 1990 PRM.

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

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

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

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

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

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

Accesses to parameters are made relative to r28 (or a copy of it) in the 1990 i860 PRM and relative to either the frame pointer or the stack pointer in the 1989 i860 PRM. Accesses to local stack storage are made relative to either the stack pointer or the frame pointer. A stack frame pointer is set up if the stack frame is large enough for it to be useful or if source level debugging is required.

Chapter 4
Optimization

4.1. Introduction

Fortran-860 does many optimizations which are not available in other Fortran compilers. These optimizations can reduce the size of a program by up to 30% and increase its speed by up to four times. Fortran-860 performs all of the optimizations performed by most other Fortran compilers. It folds constant expressions, converts multiplications into shifts and divides into multiplications when it is advantageous, and eliminates redundant jumps and unreachable code.

4.2. General Optimizations

General Optimizations always make programs smaller and faster.

4.2.1. Register Allocation by Coloring

Register allocation by coloring is used to keep the most commonly used values in registers at all times. The entire subroutine or 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 subroutine or 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, real, or logical local variable of the main program or a subroutine or function is a candidate for allocation to a register, unless it is passed to a subroutine or function.

By default, all register candidates will be allocated to the available registers so as to give either the fastest or densest code possible (as controlled by the -OL compile time option). Most Fortran compilers will allocate all local variables in memory. Fortran-860 will allocate as many local variables to registers as it can. Fortran-860 is much better than most Fortran compilers in its register allocation.

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

```
        subroutine proc
        integer i, j
        i = 1
    10  call f
        i = i + 1
        if (i .lt.  10) goto 10
        j = 1
    20  call g
        j = j + 1
        if (j .lt.  10) goto 20
        end

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

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

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

4.2.1.1. Fortran Local Variables

Many Fortran programmers have made use of the (undocumented and non-standard) assumption that local variables in a subroutine or function will retain their values between calls to the subroutine or function. This was true of the IBM Fortran compilers and has become a de facto standard. Any implementation of Fortran which does not retain the values of local variables between calls will not be able to run many important Fortran programs. Therefore, most implementations of Fortran allocate all local variables in memory. The code generated by these compilers (such as UNIX f77) is very bad because all variables are in memory. Even simple counter and index variables must be loaded from memory every time they are used.

Fortran-860 makes a very important optimization. Since it knows the complete data flow within a subroutine or function, it checks each Fortran local 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 subroutine or function, and the variable must be allocated in memory. However, most variables are initialized before they are referenced, and these variables are treated as temporary variables and are candidates for allocation to registers. In Fortran-860 most local variables are allocated in registers instead of memory.

4.2.2. Memory Allocation

Fortran-860 allocates variables based on their size, frequency of use, and other factors. Variables which are never used are usually not allocated. Variables are usually sorted to allocate the smaller and more frequently used variables first, and the larger and less frequently used variables later. This allows the use of small address offsets to access commonly used variables. If the compiler allocated some very large variable first, small address offsets might not be able to access variables allocated after it. By putting the smallest and most frequently used variables first, the compiler makes the greatest possible use of small address offsets. Some variables which other compilers would allocate in memory are allocated in registers as explained in the section "Register Allocation by Coloring".

4.2.3. Entry and Exit Code Optimization

Most compilers use a frame pointer register in each subroutine or 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, Fortran-860 does not set up a frame pointer in each subroutine or function. Fortran-860 will generate a frame pointer if the code is the same size or smaller with a frame pointer, but otherwise it will not create a frame pointer and it will access all local variables by using the stack pointer instead.

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

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

If a subroutine or 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 subroutine or function. If all of the parameters and local variables of a subroutine or function are allocated in registers (usually for a subroutine or 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 Fortran-860 is to maintain frequently used static addresses in registers. Since static addresses are 4 bytes long, if a static address is used just twice in a subroutine or function, it is faster and smaller to load the address into a register just once at the beginning of the subroutine or 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.

```
          subroutine p
          common /x/x
          integer x
          do 10 i=1,9
      10  x = x+i
          return
          end
_p:
          adds     -16,sp,sp
          st.l     r4,12(sp)
          orh      h%_x,r0,r16
          or       l%_x,r16,r16
          or       8,r0,r18
          adds     -1,r0,r19
          bla      r19,r18,.L24
           or      1,r0,r17
.L24:
.L5:
          ld.l     0(r16),r28
          adds     r17,r28,r28
          adds     1,r17,r17
          bla      r19,r18,.L5
           st.l    r28,0(r16)
          ld.l     12(sp),r4
          bri      r1
           adds    16,sp,sp
//_i      r17      local
//_x      _x       import
```

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

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

4.2.5. Register Coalescings

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

For example the Fortran-860 compiler will compile the statement "i = i*100+j" as follows

(i is in r16, j in r17)

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

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

4.2.6. Passing Parameters in Registers

In Fortran-860, most parameters are passed to a subroutine or function in registers rather than by pushing them on the stack. This avoids the memory accesses involved in pushing parameters onto the stack and in accessing the parameters from within the subroutine or function. Further improvement comes from organizing the computation of parameter values so that the value ends up in the register in which the value is to be passed to the subroutine or function. Finally, the necessity of removing the parameters from the stack after the call returns is eliminated with register parameters. This optimization reduces the code size of most programs by twenty percent.

4.2.7. Loop Rotation

In Fortran, the DO loop specifies the loop termination conditions at the top of the loop. Therefore, many Fortran compilers generate a termination test at the top of the loop and an unconditional branch from the bottom of the loop to the top of the loop. The loop will execute two branch instructions on each iteration of the loop.

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

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

4.2.8. Peephole Optimizations

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

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

4.3. Loop Optimizations

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

The loop optimizer is selected by the -OL compile time option. This compile time options informs Fortran-860 that most computation is performed in inner loops. When this compile time option is specified, Fortran-860 assigns most of the machines resources, registers in particular, to uses in the innermost loop. This can result in significant performance increases in programs which do most of their computation in loops.

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

4.3.1. In Line Multiplication and Division

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

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

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

4.3.2. Loop Invariant Analysis

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

4.3.3. Strength Reduction

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

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

4.3.3.1. Fortran Applications

Strength reduction and loop invariant analysis are particularly important to Fortran programmers. Repetitive array indexing in DO loops is very common in Fortran application programs. Since most mainframe compilers do strength reduction, many Fortran programs have been written to depend on it. Fortran programmers take it for granted that an ordinary dot product or matrix operation will generate strength reduced code. If a Fortran compiler does not do strength reduction many popular application

programs will run many times slower than expected based on a simple performance comparison of two machines. This is especially true of some Fortran applications programs being moved from mainframes or VAX/VMS to 860 UNIX systems. While the apparent performance ratio of the hardware might be 3 to 1, often the performance of Fortran programs is 10 to 1.

The inner loop of a typical numerical application, a matrix multiplication, is shown below.

```
      SUBROUTINE MATMUL(A,B,C)
      REAL A(100,100),B(100,100),C(100,100)
      DO 10 I = 1,100
      DO 10 J = 1,100
      A(I,J) = 0
      DO 10 K = 1,100
10    A(I,J) = A(I,J) + B(I,K) * C(K,J)
      RETURN
      END
```

Matrix Multiply inner loop comparison

//r20: address of A(I,J), r18: address of B(I,K), r19: address of C(K,J)

```
.L6:
        fld.l     0(r18),f26      // Load B(I,K)
        fld.l     0(r19),f27      // Load C(K,J)
        adds      4,r19,r19       // Increment address of C
        fmul.ss   f26,f27,f27     // B(I,K)*C(K,J)
        fld.l     0(r20),f26      // Load A(I,J)
        adds      400,r18,r18     // Increment address of B
        fadd.ss   f27,f26,f26     // A(I,J)+ B(I,K)*C(K,J)
        bla       r21,r16,.L6     // Decr index, compare, branch
        fst.l     f26,0(r20)      // Store result into A(I,J)
```

4.4. Pipeline Instruction Scheduler

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

Fortran-860 simulates the timing of 860 instruction sequences. When Fortran-860 detects that an instruction sequence will result in a pipeline delay, Fortran-860 attempts to reorder the instruction sequence so that the execution results remain the same, but the total execution time is reduced. Pipeline delays tend to happen frequently, since the result of an expression is often used immediately after it is computed. The instruction scheduler can often greatly reduce the total time that a sequence of instructions will take.

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

Chapter 5
Porting Programs to Fortran-860

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

5.1. Compatibility with other Green Hills Compilers

All Green Hills Languages use the same calling conventions for all subroutines, routines, procedures, and functions. Therefore, code from other Green Hills Languages can be freely used within your Fortran-860 program.

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

Fortran-860 can be obtained on any Green Hills Host. It is exactly the same on every Host. Therefore, program development can be done on more than one Host, and moving your development to a new Host system is easy.

5.2. Word Size Problems

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

If two different machines have different word sizes or if one is word addressable and the other is byte addressable, a program which operates on one machine may not operate on the other machine for several reasons. The word size affects the range of numbers implemented by the INTEGER data type. The word size also affects the precision and range of the REAL and DOUBLE PRECISION 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 makes programs difficult to port to another compile. Doing address arithmetic in integer variables is often not portable.

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.

5.4. Alignment Requirements

Fortran-860 always aligns multiple byte data items on appropriate address multiples so that all accesses will be legal and efficient. The alignment conventions for Fortran-860 are defined in the 80860 Target chapter. It is possible for the compiler to guarantee that there will be no illegal or inefficient references if the programmer follows simple rules.

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

All variables within the global frame are allocated at an optimal and legal offset from the base of the global frame. If the assembler and/or linker allocates the global frame with the maximum optimal and legal alignment of the 80860, all global data references will be optimal and legal.

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

In satisfying different alignment requirements, complex data types may be allocated differently on different machines. This will lead to the usual problems with programs which rely on memory overlays. It will also lead to problems with programs which make implicit assumptions about the size and offset of objects.

5.5. Character Set Dependencies

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

Fortran-860 uses the ASCII character set and the ASCII collating sequence. Some implementations of Fortran-860 use a different collating sequence such as EBCDIC.

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

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

You can use the intrinsic functions LGE, LGT, LLE, and LLT, to improve the portability of character-entity comparisons.

5.6. Floating Point Range and Accuracy

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

5.7. Operating System Dependencies

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

If a program contains explicit file names it may be necessary to change the names to names acceptable to the target system in order to get them to operate with Fortran-860. Refer to your target operating system documentation for a description of legal file names for your environment.

5.8. Assembly Language Interfaces

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

5.9. Evaluation Order

The Fortran-860 Language specification does not fully specify the order in which the various components of an expression or statement must be evaluated, but it disallows computations whose results depend on which permitted evaluation order is used. Many illegal programs have gone undetected for years because they have only been compiled with one compiler. Since the Fortran-860 evaluation order is not identical to the evaluation order of other Fortran-860 compilers, some of these illegal programs which operate as expected with another Fortran-860 compiler may not operate the same way when compiled with Fortran-860.

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

Expressions with side effects, such as subroutine or function calls may be executed in a different order by Fortran-860 and other Fortran-860 compilers. When a variable is modified as a side effect of an expression and its value is also used at another point in the expression, it is not defined whether the value used at either point in the expression is the value before or after modification. Different values for the same variable could potentially be used at different places in the expression depending on the order the compiler chose for evaluation.

5.10. 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 Fortran-860, because the optimizer makes major changes in the code in order to make the program smaller and/or faster. Described below are some of the most common illegal assumptions about code generation that some programs depend on to work. Please familiarize yourself with the optimizations described in the "Optimization" chapter before reading this section.

5.10.1. 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.

5.10.2. Memory Allocation Assumptions

Memory is allocated by Fortran-860 in a different way than by f77 and other Fortran-860 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. Fortran-860 will not necessarily allocate variables in the order of declaration. Some programs depend on knowing that the compiler will allocate all variables even if they are not used. Fortran-860 may not allocate unused variables. The -X331 (dbdebugallocateall) compile time option can be used to force all variables to be allocated even if they are never used. Some programs depend on knowing that certain variables will be allocated in memory. Fortran-860 will allocate certain variables to registers that f77 and other compilers would always allocate to memory. Programs compiled with Fortran-860 must not make assumptions regarding the order of allocation of variables in memory (except where the Fortran-860 language standard specifies it).

5.10.3. -OM Restrictions

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

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

5.10.4. Problems with Source Level Debuggers

Once a variable is allocated to a register it will always reside in that register. However, since other variables may share the register, the register may not always contain the value of that variable. This may cause a source level debugger to give incorrect results. If you ask for the value of a variable at a point at which that variable is about to be assigned into, the compiler may have temporarily allocated that register for some other purpose. Always check results just after they are assigned, or when the current value is going to be used later. Near the end of a subroutine or 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.

5.10.5. Dummy Assigned Goto Label List

The list in an assigned goto statement must be correct. The optimizer makes use of the list to determine data flow. Programmers have been known to put in dummy lists because many compilers ignore the list; this could cause erroneous results. If no list is present the optimizer assumes that the goto could branch to any label which appears in any ASSIGN statement in the program unit containing the assigned goto. The assigned goto may not be used to jump from one program unit to another program unit. Jumps from assembly code to assigned labels will only work if extreme care is taken.

5.11. Problems with Compiler Memory Size

Fortran-860 is an advanced optimizing compiler. It is much better that the current generation of "optimizing" microprocessor Fortran-860 compilers. In accordance with its greater capability it requires more memory. Fortran-860 requires 500 Kbytes just for the program. It is designed to work best when it has at 1 Mbyte or more of memory available. It will run in less memory but with some degradation of performance or capability.

The compiler's primary use of memory is for the program, static data structures, global declarations, parse trees, and generated machine code. Global declarations consist of the global constant, type, variable, and subroutine or 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.

Fortran-860 is a one pass compiler. That is, it reads the source program only once. Each subroutine or function is converted into a parse tree as it is read. When the end of the subroutine or function is reached the optimizer is called with the parse tree as input. The optimizer modifies the parse tree and then passes it on to the 80860 code generator. The code generator produces an internal representation of the 80860 machine code to be output for the subroutine or function. Another optimization phase is then called to modify this machine code. Finally the optimized machine code for the subroutine or 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 subroutine or function.

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

COMPLEX multiply, add and subtract are expanded in line for maximum speed. Each COMPLEX multiply generates at least four floating point multiplies and two floating point adds, and uses two temporary variables. Routines containing in excess of one hundred COMPLEX operations may require more than a megabyte of memory to compile.

5.12. Additional Undetected Errors

Many common errors can slip by the compiler and produce erroneous programs. Listed below are some of the problems to watch out for.

It is illegal (according to the Fortran-77 Standard) to pass some intrinsics as arguments to other procedures. Fortran-860 will not detect this error and an undefined reference will occur at link time.

No check is made for recursive statement functions.

Error messages follow the style of the UNIX f77 compiler. There is no indication of the position within the line of an error.

Chapter 6
Interfacing FORTRAN and C

This chapter describes the mechanism for interfacing C and FORTRAN routines within the same program.

6.1. Calling a C routine from Fortran

The Fortran compiler appends a "_" to every external name (function, subroutine and common), while the C compiler does not. To translate a Fortran external name into a C external name an underscore (_) is appended. Therefore, Fortran can only call a C routine whose name ends in "_".

All Fortran arguments are passed by reference. Therefore, each parameter in the called C routine is a pointer of the appropriate type, as follows:

Fortran Passes	C Receives
REAL*4	float *
REAL*8	double *
INTEGER*4	long *
INTEGER*2	short *
INTEGER*1	char *
LOGICAL*4	long *
LOGICAL*2	short *
LOGICAL*1	char *
COMPLEX	struct complex {float realpart, imagpart;} *
COMPLEX*16	struct dcomplex {double realpart, imagpart;} *
CHARACTER	char *

In the case of a passing a CHARACTER argument, Fortran not only passes a pointer to the "char" variable, but also passes the length of the CHARACTER variable, as an "in" (NOT as an "int *") at the end of the argument list. Fortran CHARACTER strings constants are null terminated.

If the C routine being called from Fortran is a function, then the return types correspond as follows: an "int" C function must be declared either as INTEGER and LOGICAL in the calling Fortran routine. A "float" or "double" C function must be declared as DOUBLE PRECISION in the calling Fortran routine. Since C usually promotes "float" return values to "double", REAL return values usually cannot be returned from C. COMPLEX, DOUBLE COMPLEX, and CHARACTER are returned by passing the address of where the return value is to be stored as an extra first parameter to the C function. The length of a CHARACTER return value is passed as an extra second "int" parameter to the C function.

The alternate return statement or Fortran, "RETURN e", is equivalent to "return(e)" in C. The Fortran caller does a computed GOTO statement on the return value to implement the alternate return.

6.2. Calling a Fortran routine from C

The Fortran compiler appends a "_" to every external name, while the C compiler does not. Therefore, a C routine can only call a Fortran routine by referring to it with an explicit "_" on the end of the name. The same naming convention is used for a C routine to reference a Fortran named COMMON block.

All Fortran parameters are passed by reference. Therefore, the corresponding argument in the C call must be a pointer of the appropriate type. For example, to pass a scalar variable X from C to Fortran, pass & X.

C Passes	Fortran Receives
float *	REAL*4
double *	REAL*8
long *	INTEGER*4
short *	INTEGER*2
char *	INTEGER*1
long *	LOGICAL*4
short *	LOGICAL*2
char *	LOGICAL*1
struct complex {float re, im;} *	COMPLEX*8
struct dcomplex {double re, im;} *	COMPLEX*16
char *	CHARACTER

In the case of a passing a CHARACTER argument, C must not only pass a pointer to the "char" variable, but must also pass the length of the "char" variable, as an "int" (NOT as an "int *") at the end of the argument list.

If the Fortran routine being called from C is a FUNCTION, then the return types correspond as follows: an INTEGER or LOGICAL Fortran FUNCTION must be declared as "int" in the calling C routine. A DOUBLE PRECISION Fortran function must be declared as "double" in the calling C routine. Since C usually promotes "float" return values to "double", a REAL return value may not be accessible in C. COMPLEX, DOUBLE COMPLEX, and CHARACTER are returned from the called Fortran routine by passing the address of where the return value is to be stored as an extra first parameter to the C function. The length of a CHARACTER return value is passed as an extra second "int" parameter to the C function. The alternate return statement or Fortran,"RETURN e", has no equivalent in C.

Chapter 7
Fortran Runtime Library

The Green Hills Fortran-860 Runtime Library is supplied with the Fortran-860 compiler. The Fortran-860 Runtime Library requires the functions available in a standard C runtime library. On UNIX systems, the standard C library should be loaded along with the Fortran-860 runtime library (this will normally be done by default). For non-UNIX systems which do not already have a C library, Green Hills supplies the C-860 Runtime Library.

The use of these libraries requires no licensing except the standard Fortran-860 compiler license. Under this license, unlimited distribution of programs incorporating the library code is permitted without further charge. However, distribution of the library, or its components, is not permitted.

The Fortran-860 Runtime Library includes all of the functions required by the ANSI/IEEE standard plus access to many additional functions in the C Library.

7.1. UNIX Fortran Runtime Library

If you are running under the UNIX operating system you may be using the AT&T or Berkeley UNIX Fortran runtime libraries. Fortran-860 can operate with the standard UNIX Fortran runtime libraries as well as the Fortran-860 Runtime Library. The choice of which runtime libraries to use is made by the compiler software system integrator.

Chapter 8
Compile Time Options

Each Fortran-860 compiler is configured to enable some of the compile time options described in this chapter and to disable the rest. Your compiler should have been configured so that it can be used in its intended environment without needing to specify any special compile time options. All normal compile time options are documented in the compiler invocation documentation that you received with Fortran-860. It should also document all options that are supported by your implementation.

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

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

Most options are case sensitive and are only recognized as shown in the documentation. The compiler accepts but ignores any invalid optionspecifications, therefore the -v (Unix only) and/or -X255 options are often useful to verify which options were accepted and are in use by the compiler for the current compilation. The -X prefix options are used to turn on a specific function. To negate the effect of the -X prefixed option, the -Z prefix is used instead. For example, -X308 turns on tail recursion optimizations. If this optionis set by default for your compiler, using -Z308 will turn off tailrecursion optimizations.language.

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 onlyobject 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 Turn on run time checking of subranges and array bounds. The code will be much slower under this option.

-Dname (UNIX Host only) For files named *.F 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
(UNIX Host only) For files named *.F define "name" to the preprocessor with the value "string". This is equivalent to putting "#define name string" at the top of the source file.

-Exxx (UNIX Host only) Pass the string "xxx" to EFL as an option when preprocessing .e files into .f files.

-F (UNIX Host only) Do not produce assembly, object, or executable files, produce only Fortran source files. For each source language file named "*.F" preprocess the source language file with the C preprocessor and leave the preprocessor output on a file whose name ends in ".f". Similarly preprocess files named "*.e" with the EFL preprocessor, and files named "*.r" with ratfor.

-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".

-i2 Make the type INTEGER be INTEGER*2.

-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.

-m (UNIX Host only) Preprocess files whose names begin with ".r" with "m4" before running the ratfor preprocessor.

-p (UNIX Host and Target only) Generate calls for execution profiling. The UNIX profiler must be available; a profiler is not part of the library provided by Green Hills.

-pg (BSD UNIX Host and Target only) Generate more profiling information, and force all routines to have frames.

-o filename (UNIX Host only) Place the executable file output into the file named "filename". If this option is not specified the executable file will be named "a.out". This option is ignored if "-c", "-S", or "-F" is present.

-O The -O option activates the Green Hills optimizers which are safe for use on all programs, except for the loop optimizer. If used in conjunction with -ansi (or -X153), -OM is assumed.

-OM This option is eqivalent to -O except that it also allows the optimizer to assume that memory locations do not change except by explicit stores. That is, the optimizer is guaranteed that no memory locations are I/O device registers that can be changed by external hardware and no memory locations are being shared with other processes which can change them asynchronously with respect to the current process. This compile time option must be used with extreme caution (or not at all) in device drivers, operating systems, shared memory environments, and when interrupts (or UNIX signals) are present.

-OL Optimize the program to be as fast as possible even if it is necessary to make the program bigger. In particular, most of the available resources are allocated to optimizations of the innermost loops. The -OL compile time option will perform optimizations which may make the program faster but larger. It is counter-productive to specify -OL on code which contains no loops or that is rarely executed as it will make the whole program larger but no faster. After experimenting with a program it is possible to discover which modules benefit from -OL and which ones do not. The -X482 option may be used in conjunction with -OL to allow various loop optimizations to be performed without turning on loop unrolling. In addition, -OL causes all scalar multiplies are performed inline, and larger block moves are performed with sequential moves rather than an inline loop.

-OLM This option is equivalent to -OL and -OM.

-OML This option is equivalent to -OLM.

-onetrip Execute at least one iteration of every DO loop. The default is that if the lower bound is greater that the upper bound to execute no iterations of the DO loop (this is the ANSI Fortran-77 standard). This was unspecified under the ANSI Fortran-66 standard and some important implementations (especially IBM) chose to always execute the loop at least once. The use of this option makes the compiler incompatible with the ANSI Fortran-77 standard, but it may be necessary to use it to get certain old Fortran-66 programs to operate correctly.

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

-Rxxx (UNIX Host only) Pass the string "xxx" to ratfor as an option when preprocessing .r files into .f files.

-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".

-U Do not convert upper case user-supplied names in Fortran to lower case. By default, Fortran is not case sensitive and all Fortran names which are externally visible are in the object file in lower case. In case one wishes to gain access to names defined in C as upper case this option can be used. However, use of this option makes the compiler incompatible with the ANSI Fortran-77 standard.

-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.

-X9 Disable local (peephole) optimizer.

-X13 Suppress code generation. An empty output file will be created.

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

-X21 Map all identifiers to upper case, for assemblers which require this.

-X31 (Non-UNIX Host only) Allow arbitrary file names to be specified to compiler.

-X32 Display the names of files as they are opened. Useful for finding out why the compiler cannot find an include file.

-X37 Emit a warning when dead code is eliminated.

-X39 Do not move frequently used procedure and data addresses to registers.

-X57 Generate bounds checking code for subranges and arrays.

-X68 This makes characters unsigned as they are in some implementationsof Fortran. The default is signed characters as in UNIX f77.

-X71 Use the Green Hills single precision math libraries.

-X74 The target system is UNIX System V Generate object files instead of assembly files.

-X77 Turn off compile time checking of FORMAT statements. Use this option if your runtime library supports FORMAT statement features that the Fortran compiler doesn't know about.

-X79 Pad hollerith constants on the right with blanks. The default is compatible with UNIX f77: only the first byte of the hollerith is significant and the constant is zero padded on the left.

-X80 Turn off the branch tail merging optimization. This can speed up compilation in some cases.

-X81 Allow extern variables to be initialized (by turning off extern). This is an error in cc, and by default in Fortran-860.

-X82 Compile lines starting with "x", "X", "d", "D". The default is to treat them as comments. Used for enabling debugging statements.

-X85 (UNIX.i Target only) Generate ".lcomm" (BSD) or ".bss" (UNIX System V) for zero initialized statics, rather than placing uninitialized local data in the initialized data section with initial value 0. This can result in significant reduction in the size of binary files. The -X85 option is normally on by default in a Unix environment.

-X114 (UNIX Target only) Target is UNIX BSD 4.2

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

-X161 Extend source to interpret columns 1 through 132 instead of 1 through 72 only.

-X164 Do not stop in the event of a code generator abort or "Internal Compiler Error" error message. This option is occasionally useful for determining the cause of a compiler failure. If this option is used, the compiler may crash or otherwise terminate abnormally.

-X168 Do not move invariant floating point expressions out of loops.

-X171 Do not create a static base register.

-X181 VMS Fortran compatibility.

-X202 Don't output "." before assembler directives.

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

-X219 Suppress elimination of jumps to jumps.

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

-X237 Apply associative rules in common subexpression elimination.

-X239 The host operating system of the compiler is MS-DOS. Change include file conventions etc. appropriately.

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

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

-X265 Suppress register database phase of peephole.

-X266 Repeat the peephole phase until the code doesn't get any better.

-X268 Don't delete redundant register alignments.

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

-X272 Suppress the realvar code in database phase.

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

-X285 Suppress block merge phase.

-X307 Turn off instruction reordering (don't attempt to fill gaps between instructions). This makes output easier to read, but much slower.

-X308 Perform tail recursion optimizations.

-X312 Suppress recognition of ?: operators as absolute value and min/max.

-X313 Generate code for multiple initializations of common variables. Only works with Avalon assemblers.

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

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

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

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

-X333 Suppress passing of front end information to the peephole optimizer and instruction scheduler.

-X337 Suppress most multiple initialization error messages. Some program units initialize the same variable more than once. This is illegal, but it is supported by some compilers. If a variable is initialized more than once with the same value, this switch may make the program work. If the variable is initialized with different values, this switch may mask the detection of a serious program error.

-X344 Suppress adrconst optimizations. Do not try to undo ineffective allocation of constants to temporaries.

-X353 Perform common subexpression analysis twice. Rarely useful.

-X356 Make sure DO loop variable is always up to date (never cache it).

-X370 Output line numbers in the assembly file.

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

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

-X401 Enable CEXTERNAL declaration.

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

-X405 Type "char" is unsigned type.

-X407 Turn on IEEE machine independent software floating point.

-X410 Modify front end to parse i860 preprocessor commands.

-X412 Force a call to a precise divide routine.

-X414 Use 96-bit objects for long doubles.

-X422 All types larger than 128 bits are forced to that alignment.

-X424 Replace unsigned division by constant with multiply by the pseudo-reciprocal and shift right.

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

-X426 DOD MIL STD 1753 features enabled.

-X428 Turn on CSE register caching.

-X429 Don't try to propagate common subexpressions through loops.

-X434 Put in code to check for nil pointer dereferences.

-X435 Presume 386 MicroSoft asm rules.

-X442 No special tricks for constant multiplies.

-X447 Produce code to generate a runtime error if a switch (case, computed goto) statement has no default (otherwise) case and is entered with a value that is not one of the listed cases.

-X463 Perform loop unrolling.

-X465 Mark errors in cpp listing file.

-X468 Don't propagate constants in their own pass.

-X470 Suppress tail recursion.

-X474 Suppress Common Subexpression Elimination (CSE). .

-X482 Disable loop unrolling. This option is intended for use with -OL to allow various loop optimizations to be performed without turning on loop unrolling.

-X483 Flush assembly output at end of each routine.

-X490 Use following string for .file directive.

-X491 Kanji character support.

-X496 Check to make sure all args/vars are used.

-X500 Don't delete .s file if errors encountered.

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

-X505 Enable new loop optimizer.

-X509 Complain about locals read before written.

-X510 Suppress .stab for enum types.

-X523 Use same technique as -X424 except use for mod operator.

-X524 Equivalent to -X424 and X523, except for signed operands.

-X525 For machines without 33-bit shift capability for which you wish to activate

-X424, -X523 and/or -X524.
Replaces a divide or mod operation, which has a constant divisor, with two multiplies and a right shift.

-X543 Suppress slow divide optimizations. Equivalent to -Z424, -Z523, -Z524 and -Z525.

-X588 Put in code to generate a runtime error when an uninitialized variable is accessed.

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

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

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

-X615 Use the calling conventions specified in the 1989 i860 PRM.

-X616 Use the calling conventions specified in the 1990 i860 PRM.

-X617 Use the assembler syntax specified in the Intel i860 Architecture and System V ABI.

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

Chapter 9
Runtime Errors

The following table shows the Fortran Runtime error numbers and associated error message strings.

Error Number	Message
100	Error in format
101	Off end of record
102	Error in list input
103	Read unexpected character
104	Horrible I/O error
105	Out of memory
106	Buffer too large
107	Blank logical input
108	Illegal record number
109	Too many decimal digits
110	Floating point overflow
111	Expected "=" in NAMELIST input
112	Digit expected in NAMELIST input
113	Expected "*" in NAMELIST input
114	Positive constant expected in NAMELIST input
115	Illegal type conversion in NAMELIST input
116	Too many elements specified in NAMELIST input
117	Expected ")" in NAMELIST input
118	Expected "," in NAMELIST input
119	Syntax error in NAMELIST input
120	Name expected
121	No such name in NAMELIST
122	Substring lower bound must not be zero
123	Expected ":" in NAMELIST input
124	Substring upper bound out of range
125	Error in formatted input
200	No such unit
201	Cannot do direct access on this unit
202	Record too large
203	Illegal modification of attributes in open
204	File already exists
205	Bad unformatted file
206	Sequential access required
207	Formatted I/O to UNFORMATTED file
208	Unformatted I/O to FORMATTED file
209	Indexed I/O not supported
210	Attempt to write to READONLY file
211	Invalid or no RECL = specified for Direct file

9.1. Compiler generated debugging checks

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

-X57 Array index/variable assignment out of bounds
-X447 Case/switch index out of bounds

Chapter 10
Compile Time Errors

The following table lists the Fortran compile-time error messages in alphabetical order.

%FILL expected
A FUNCTION may not return a STRUCTURE
Alternate return not allowed from function
Alternate return specifier only in CALL statement
Array index out of bounds
Array size exceeds implementation limit
Array, variable or Common block name expected
Attempt to read off end of line
Cannot assign to this object
Cannot have REC= in list directed I/O
Cannot open file:
Cannot take the address of this object
Cannot use this type with %VAL
Character constant expression expected
Character expression expected
Character function length undefined
Common block name expected
Concatenate operand must be a character expression
Constant expected
Constant expression expected
Could not disambiguate overloaded procedure name:
Direct or unformatted I/O illegal
Do not know how to do list I/O on this type
Duplicate field:
Duplicate specifier
END Statement expected
END Statement expected at end of file
END statement missing
END= not allowed in WRITE statement
ENTRY Statement illegal
ENTRY Statement illegal in main program
EQUIVALENCE entity extends before common block
EQUIVALENCE forces misalignment of item
Empty Statement illegal in Logical IF
Empty Statement with Label is illegal
End of line found in Radix-50 constant
End of line found in hollerith data
End of line found in string
Entry parameter declared after use:

Error in complex constant
Error on line
Fatal error in reading library file:
File name too long
Format identifier expected
Format statement must have a label
I/O error on read
I/O list not permitted for NAMELIST I/O
INQUIRE may not specify both UNIT and FILE
INTEGER*4 variable expected
INTEGER/REAL/DOUBLE PRECISION value required
INTEGER/REAL/DOUBLE PRECISION variable expected
Illegal END DO statement
Illegal EQUIVALENCE item
Illegal EQUIVALENCE or DATA item
Illegal Format Statement
Illegal Label
Illegal OPTIONS statement
Illegal Radix-50 character
Illegal argument to %LOC
Illegal built-in function name
Illegal character
Illegal concatenation of indeterminate length
Illegal continuation card
Illegal data-implied-do-list
Illegal digit in hex or octal constant
Illegal goto into loop to label:
Illegal initialization
Illegal length specification
Illegal operation
Illegal option:
Illegal placement of OPTIONS statement
Illegal placement of PROGRAM statement
Illegal specifier
Illegal statement in DO range
Illegal statement in logical IF
Illegal statement number
Illegal substring
Illegal substring operation
Illegal terminal statement of DO-loop
Illegal to equivalence two common entities
Illegal to initialize a Non-constant address
Illegal type for field
Illegal type for:
Illegal type of object in NAMELIST

Illegal variable or expression
Implicit statement illegal
Include nested too deeply
Incompatible definition of function type:
Incorrect label on END DO statement
Inliner Fatal Error: Variable not on decllist.
Inliner: Can not save nested routines.
Inliner: Cannot process ENTRY statements.
Integer constant expression expected
Integer or string constant expected
Integer variable expected
Integer*4 variable expected
Internal Compiler Error
It is illegal to jump to a FORMAT statement
Label expected
Label not defined
Label used inconsistently
Length of COMPLEX must be 8 or 16
Length of INTEGER must be 1, 2, or 4
Length of LOGICAL must be 1, 2, or 4
Length of REAL must be 4 or 8
Letter expected
Line longer than 13200 characters
Logical*4 variable expected
More than 7 dimensions not allowed
Multiple definition of FORMAT statement
Multiple definition of name:
Multiple definition of statement label
Multiple initialization
Must have logical unit number
NAMELIST group-name expected
NAMELIST name must be a variable name or array name
NAMELIST name not a constant size:
No matching if statement above
No such field in STRUCTURE:
No such intrinsic:
Non-digit in statement number field
Not constant size:
Not enough constants specified
Only 5 digits allowed
Only one equivalence item specified
Only the last dimension can be an asterisk
Parameter name already defined
Procedure name expected
REAL or COMPLEX operand not allowed here

Radix-50 constant too long (limited to 12 characters)
Ran out of string space
Repetition count must be greater than zero
Same control variable used for inner and outer DO loop
Scalar reference or array reference expected
Specification Statement Illegal
Statement function parameter must be constant size
Statement function type must be constant size
String initializer of numeric
Structure-name required at outermost level
Substring of parameter not allowed
There is an illegal GOTO to the end of this DO loop
This compiler must be used on a licensed system.
This error message reserved for the inliner.
This warning message reserved for the inliner.
Too few subscripts
Too many -I options
Too many constants specified
Too many subscripts
Too many subscripts specified
Type conflict for:
Type error in intrinsic
Type error in statement function
Type expected
Type mismatch
UNIT must be specified
USEROPEN value must be EXTERNAL SUBROUTINE
Undefined STRUCTURE name:
Undefined name:
Unit identifier expected
Unmatched left parenthesis
Unmatched right parenthesis
Unrecognized Statement
Variable expected
Warning: value out of range
gap left in common block due to alignment rules
warning: Arithmetic constant too large for type/array index
warning: Variable read before written: