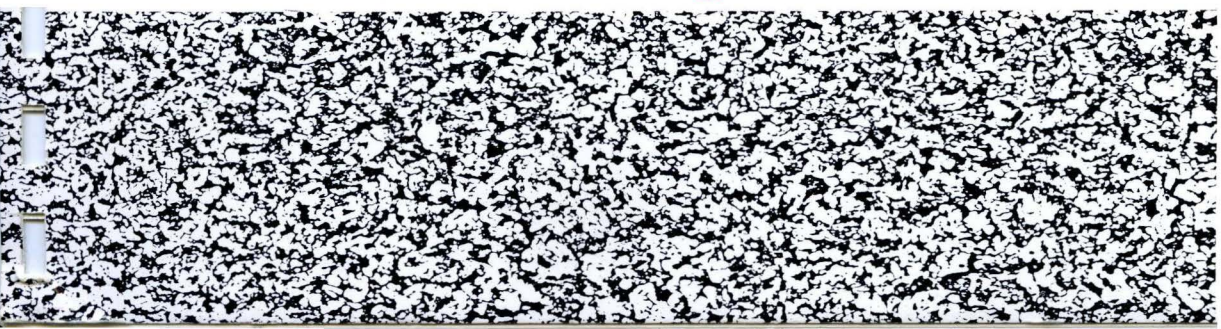
UNIX Compiler Manual: C, Pascal, FORTRAN 77

An NBI Company

Integrated Solutions



UNIX Compiler Manual:
C, Pascal, FORTRAN 77

Integrated Solutions
1140 Ringwood Court
San Jose, CA 95131
(408) 943-1902

490292 Rev. A

April 1989

Preface

Integrated Solutions distributes compilers for C, Pascal, and FORTRAN. This manual describes these three compilers. Information in this manual is applicable to all 68000 series targets; that is, 68000, 68010, 68020, and 68030.

Effective with ISI Release 5.1 (4.3BSD), we offer a second FORTRAN compiler. This compiler is called **ftn**(1). Information on **ftn** is not included in this manual. Refer to the online man page for a description of the compiler. To get additional documentation on NKR FORTRAN, order part number 720110 from Sales or Customer Support.

Section 1 - Introduction
This section explains the general characteristics of the compilers.

Section 2 - Optimization
The Optimization section gives detailed information about the optimizations used by the three ISI compilers (ISI C, ISI Pascal, and ISI FORTRAN) to improve program performance. It also gives you general ideas as to how to get the best performance out of your program.

Section 3 - Porting Programs to ISI Compilers
This section discusses difficulties that you may encounter in moving a program developed with another compiler to the ISI compiler. It gives specific examples of problems that may appear and how to resolve them. Although many examples are shown for ISI C, this section is applicable to ISI C, ISI Pascal, and ISI FORTRAN.

Section 4 - Using the 68000 Series Compilers
This section describes the target processor and operating system environment in which your program will operate. It describes calling conventions, register allocation and memory allocation strategies. Restrictions imposed on the compiler by the MC68000 are included. It also tells how to modify the output of the compiler to be compatible with different target environments. This section is applicable to all ISI compilers (ISI C, ISI Pascal, and ISI FORTRAN).

Section 5 - Pascal
The section contains a discussion of the implementation of the Pascal programming language. The Pascal compiler is referred to as ISI Pascal in this document. Additional information can be found in the **pc**(1) man page.

Section 6 - FORTRAN 77
The section contains a discussion of the implementation of the FORTRAN programming language. The FORTRAN compiler is referred to as ISI F77 in this document. Additional information can be found in the **f77**(1) man page.

Section 7 - C Language
This section contains a discussion of the implementation of the C programming

language. The C compiler is referred to as ISI C in this document. Additional information can be found in the **cc**(1) man page.

Section 8 - Compile Time Options
This section describes how to adjust the output of the ISI compilers to accommodate your needs by using the many variations that have been implemented.

Appendix A - Man Pages
The man pages for the three compilers (**cc**(1), **pc**(1), **f77**(1)) are included here.

You will see these uses of **boldface** in this manual:

- Section headings.
- References to UNIX commands (**tar** (1) refers to the command **tar;** the "(1)" shows that you can find a description of **tar** in "Section 1: Commands" of the UNIX 4.3BSD *User's Reference Manual* (URM)).
- Commands that you type to UNIX exactly as printed (for example, "Enter **fsck** and press RETURN").
- Messages that UNIX prints on your screen (for example, **login:**).
- User account names (for example, the **root** login account, the group **operator**).

Table of Contents

Section 1: Introduction

1.1 Memory Requirements

The Integrated Solutions compilers are advanced optimizing compilers. They are a major improvement over the current generation of microprocessor compilers. In accordance with their complexity they require a lot of memory. These compilers require up to 200K for the program (after using a compiler to compile them). They are designed to work best when they have at least 400K of memory available.

The compilers' primary use of memory is for the program and static data structures, global declarations, parse trees, and generated machine code. The program and static data structures consume approximately 250K. Global declarations consist of the global constant, type, variable, and procedure declarations. This is a major use of memory when large numbers of declarations are included in a program, since even unused global declarations must be stored throughout the compilation.

The compilers are one pass, reading the source program only once. Each procedure is converted into a parse tree as it is read. When the end of the procedure is reached the optimizer is called with the parse tree as input. The optimizer modifies the parse tree and passes it on to the code generator. The code generator produces an internal representation of the machine code to be output for the procedure. After another optimizer phase is called to modify this machine code, the final machine code for the procedure 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 procedure.

The memory usage for parse trees and machine code is determined by the size of the largest procedure in the program. If memory size problems exist, try to reduce the size of the include files by including just the declarations that are needed. Simple procedures 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.

1.2 Memory Allocation

Memory is allocated by the ISI compilers in a different way than by other compilers. The ISI compilers perform a number of optimizations which other compilers do not. This can lead to problems in porting some programs from other compilers to the ISI compilers if the programs depend on memory or register allocation peculiarities of other compilers.

The Pascal compiler **pc**(1) allocates variables based on their size, frequency of use, and other factors. No assumptions regarding contiguous allocation of variables can be made safely (except where specified by the language standard). Programs that otherwise rely on the ordering of variables within memory may not work. Variables may be reordered, some may be in registers, others may be eliminated altogether.

1.2.1 Local Variables

Unlike other compilers, local variables (automatic variables in C) may be put into registers by the compiler even if a register declaration is not used. In addition, the lifetime of each local variable is studied, and if possible several variables may share the same register in one routine. By default there is no upper limit on the number of variables allocated to a register. This is determined by their lifetimes.

Scalar, pointer, or floating point variables generally qualify for allocation to a register unless they are passed by reference or by its address. A floating point variable generally doesn't qualify for allocation to a register unless an architecture directly supports floating point arithmetic (i.e., the MC68881 is used). The compiler, by default, tries to allocate all eligible local variables to registers.

When a variable is allocated to a register it always resides in that register. However, since other variables may share the register, the value of the register may not always contain the value of the variable.

Variables declared in register declarations are allocated to registers before any non-register variables are allocated to registers. This allows you to specify the most important variables.

1.2.2 Static and Global Variables

A static variable may be allocated to a register if the compiler can determine that the variable can never hold a value across invocations of the routine in which it is declared; i.e., the static variable is always assigned in the subroutine before it is referenced. This optimization is particularly important to FORTRAN 77, which uses primarily static variables. A static variable may have no memory allocated if it is never used.

1.3 Converting Programs Developed with Other Compilers

There are a number of general problems involved with transporting programs from other compilers. The most common problems to watch out for are explained in the following subsections. Refer to Section 3 for more detailed information.

1.3.1 Byte Ordering

Most machines today are byte oriented. That is, they address 8 bit bytes. They have operations which operate on 8, 16, 32, 64 and/or 128 bit quantities. Most older machines are word oriented, in that they address words of a standard size varying from 16 to 64 bits.

The word size of a machine has several effects on the transportability of programs. The standard integer and floating point data types of C, Pascal, and FORTRAN are generally defined by the word size. In particular, the word size affects the range of numbers implemented by the integer data type and the precision and range of the standard floating point datatypes. Most programs expect the size of the integer and single precision floating point data type to be 32 bits and the double precision floating point data type to be 64 bits.

The most common word size problems are (often undetected) integer overflows and floating point underflows and overflows. The layout of bit aligned data structures in C and Pascal will vary with the word size. Overlaying structures in memory (with C union types or Pascal unchecked variant records) makes programs difficult to transport. Doing pointer arithmetic in

integer variables is generally not transportable between machines with different word sizes. C provides portable pointer arithmetic if it is used correctly.

Programs designed for maximum portability should only assume 16 bit integers and 32 bit floating point.

1.3.2 Byte Machine Portability

Compilers developed on 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 generally not portable.

Section 2: Optimization

2.1 Introduction

ISI C, ISI Pascal, and ISI FORTRAN do many optimizations which are not available in other compilers. These optimizations can reduce the size of a program by as much as 30% and increase its speed by up to four times. The ISI compilers also perform all of the optimizations done by most other compilers. They fold constant expressions, convert multiplications into shifts and divides into multiplications when advantageous, and eliminate redundant jumps and unreachable code.

2.2 General Optimizations

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

2.2.1 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 is used later on. Often, all local variables are kept in registers and none in memory.

By default, any integer, pointer, enum, float, or double automatic (or register) variable in C is a candidate for allocation to a register, unless its address is taken with the "&" operator. The same is true for any integer, real, or logical local variable of the main program or function for FORTRAN, unless it is passed to a function, and any integer, pointer, or real local variable of a function for Pascal, unless it is passed by reference.

By default, all register candidates are allocated to the available registers so as to give either the fastest or densest code possible (as controlled by the **-OL** compile time option). Most C compilers allocate one register variable to each available register and then allocate all other register variables and all automatic variables in the stack frame. Most Pascal compilers will allocate all local variables in memory. The ISI compilers allocate as many of the register variables to registers as possible. Then they allocate any automatic variables to registers if possible. Overall, the ISI compilers allocate registers more effectively than most other compilers.

In the following example, ISI C 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();
}
```

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

The savings by ISI C 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

The same results can be seen using ISI Pascal, as follows:

```
procedure proc;
var
    i, j: integer;
begin
    for i := 1 to 9 do
        f;
    for j := 1 to 9 do
        g;
end;
```

The same results can be seen using ISI FORTRAN, as follows:

```
      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
```

2.2.2 Memory Allocation

The ISI compilers allocate 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 small address offsets. Some variables which other compilers would allocate in memory are allocated in registers as explained in Section 2.2.1.

2.2.3 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 back 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, the ISI compilers do not set up a frame pointer in each function. It generates a frame pointer if the code is the same size or smaller with a frame pointer; otherwise, it does not create a frame pointer and accesses 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 guarantees that there will always be a frame pointer, but it usually increases the size of the program.

When only one or two registers need to be saved in a function, the ISI compilers 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 Section 2.2.1 for an example of improvements to the entry and exit code.

2.2.4 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. 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 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 in next section for use of Stack Adjustment Coalescing.

2.2.5 Static Address Elimination

A valuable optimization performed by the compilers is to store frequently used static addresses in registers. Since static addresses are 4 bytes long, it a static address is used just twice in a function, it is faster and smaller to load the address into a register just once at the beginning of the function and always use "register indirect" addressing to access it. In this way, most static references are reduced to one-third the space and a shorter execution time. The following example was done using ISI C.

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

ISI C		UNIX PCC	
p:		p:	
		link	a6,#0
movl	a2,sp@-	moveml	#0,a6@(0)
movl	#f,a2		
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		
movl	sp@+,a2	moveml	a6@(0),#0
		unlk	a6
rts		rts	
40 bytes		76 bytes	

The savings by ISI C 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

2.2.6 Loop Rotation

In the ISI Compilers, 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 executes two branch instructions on each iteration of the loop.

A better way to generate code for loops, called loop rotation, is to place the test at the bottom of the loop. If it can be determined at compile time that the loop always executes at least once, the loop is entered from the top. If it cannot be determined that the loop will be executed at least once, 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.

2.2.7 Peephole Optimizations

Peephole optimizations are local improvements to the code which are certain to be correct without further analysis of the surrounding code. An example would be two machine instructions where the first moves the contents of register A to register B, and the second instruction moves the contents of register B to register A. If the program code never branches to the second instruction (i.e. both instructions are always executed together), and subsequent instructions do not use the condition codes set by the second instruction, the second instruction can be safely eliminated. The ISI compilers keep the full lifetime 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.

2.3 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 the compiler that most computation is performed in inner loops. When this compile time option is specified, the compiler 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.

2.3.1 In Line Multiplication

The MC68000 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 a MC68000, 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 MC68000, under most compilers, the ratio is greater than 100 to 1. Under the **-O**

compile time option, the compiler 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, the compiler uses 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 assumes 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 uses 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.

2.3.2 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 are accessed with "register mode" and all invariant addresses are accessed with "register indirect modes." This optimization usually eliminates all computations of invariant expressions and addresses in loops.

2.3.3 Strength Reduction

Strength reduction is found only in the most advanced compilers. It applies to loops which have an index variable which is incremented by a constant on each iteration of the loop. When a loop index variable is used as the subscript for an array, most compilers multiply the loop index by the size of the array elements and add this offset to the base of the array. Each such reference typically requires 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.

Section 3: Porting Programs to the ISI Compilers

3.1 Introduction

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

3.2 Compatibility with Other Compilers

The ISI languages (C, Pascal, and FORTRAN 77) use the same calling conventions for all subroutines, routines, procedures, and functions. Therefore, code from other languages can be freely used in your ISI programs.

3.3 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 in that 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 (ISI C), INTEGER data type (ISI FORTRAN), or integer data type (ISI Pascal). The word size also affects the precision and range of the float and double data types (ISI C), real data types (ISI Pascal), and READ and DOUBLE PRECISION data types (ISI FORTRAN).

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 in ISI C; with unchecked variant records or pointers in ISI Pascal) makes programs difficult to port to another compile. Address arithmetic done in integer variables is often not portable. C provides portable pointer arithmetic if it is used correctly.

3.4 Byte Order Problems

There is one major portability problem between byte machines. Some machines place the most significant byte of a multiple byte integer value at the lowest address. Many byte machines such as the MC68000 have followed this convention. Other machines place the least significant byte of a multiple byte integer value at the lowest address. 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.

3.5 Alignment Requirements

The ISI compilers always align 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 the compilers are defined in Section 4. It is possible for the compiler to guarantee that there will be no illegal references if the programmer follows simple rules.

When using the ISI compilers, the size of all compound data types are rounded up to a multiple of the maximum alignment required for any component data type. The compiler always aligns parameters and local variables within the stack at an allowable offset from the beginning of the frame. The compiler always rounds up the size of the frame to the minimum alignment of the MC68000/10/20/30. If the initial stack pointer is aligned to the maximum optimal alignment of the MC68000/10/20/30 and if the program involves no explicit manipulation of the stack pointer, all stack references are optimal and legal.

All variables within the global frame are allocated at a legal offset from the base of the global frame. If the assembler and/or linker allocates the global frame with the minimum legal alignment of the MC68000/10/20/30, all global data references are optimal and legal.

The ISI compilers alway ensure that components of a data structure requiring alignment appear only at a legal offset from the beginning of the data structure. If all allocation routines always return pointers which are aligned to the minimum legal alignment of the MC68000/10/20/30 and the program does not use integer arithmetic for pointer computations, all references to dynamically allocated memory are optimal and legal.

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

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

3.6 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.

The compilers use the ASCII character set and the ASCII collating sequence. Some implementations of compilers 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 dependence 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.

3.7 Floating Point Range and Accuracy

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

3.8 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 the ISI compilers. Refer to your target operating system documentation for a description of legal file names for your environment.

3.9 Assembly Language Interfaces

Programs which use embedded assembly code or interface to external assembly code may require assembly code to be rewritten when the program is transported to a new machine.

3.10 Expression Evaluation Order

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

Some implementations of the languages evaluate the arguments to a function from right to left, others from left to right. See Section 4 for details of the ISI compiler calling conventions.

Expressions with side effects, such as function calls and the operators "++", "--", "+=", etc., may be executed in a different order by the ISI compilers than by other 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.

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

A particular case of evaluation order dependency is the use of the "?:" operator in an expression which is an argument to a function call. ISI C evaluates the question-mark operator before any other arguments, and keeps the result in a temporary variable. ISI C evaluates the "?:" operator at its position in the argument list. The call"foo(b?i:i+i, i++)" will usually evaluate differently under PCC than under ISI C.

3.11 C Preprocessor Incompatibilities

The C preprocessor that is provided with PCC has many undocumented features. Most of these undocumented features are implemented in ISI C. One little known feature of the C Preprocessor allows the results of two macro expansions to be concatenated into a single token. For instance:

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

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

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

Due to the one pass nature of ISI C it is not possible for its builtin preprocessor to manufacture a token such as "/*". In order to compile a program with such constructs it is necessary to run ISI C in two passes. First compile the program with the **-E** compile time option to produce the preprocessed source. Then compile the preprocessed source as you would normally.

However as a special case the compiler can construct an identifier as:

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

which becomes (in both PCC and ISI C):

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

3.12 Illegal Assumptions about Compiler Optimizations

Some programs illegally depend on the exact code that a particular compiler generates. Such programs are particularly difficult to port to an advanced optimizing compiler, such as ISI C, 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 Section 2 before reading this section.

3.12.1 Problems with Setjmp and Longjmp

*** NOTE ***

Under the default configuration of ISI C, 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. 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 UNIX programs depend on whether a variable's value will be restored by longjmp. Since the compiler may allocate automatic variables to registers and may allocate "register" variables in memory, it is not predictable as to whether any modifications to a variable which take place after a setjmp will be retained or lost after a call to longjmp on the same "label" variable.

The **-X18** switch causes all programmer defined variables which are not declared "register" to be allocated in memory as in the portable C compiler. The **-X18** switch generates worse code than the default configuration, but in the few cases in which the (undocumented) subtleties of setjmp and longjmp are depended upon, it will operate consistently with the portable C compiler. The compile time option **-X125** automatically activates the **-X18** compile time option for any function in which there is a call to a function called "setjmp". This option is turned on by default.

3.12.2 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.

3.12.3 Memory Allocation Assumptions

Memory is allocated by the ISI compilers in a different way than with most other 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. The compilers 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. The ISI compilers may not allocate unused variables. Some programs depend on knowing that certain variables will be allocated in memory. It will allocate certain variables to registers that other compilers would always allocate to memory. Programs compiled with ISI compilers must not make assumptions regarding the order of allocation of variables in memory (except where the C language standard specifies it).

3.12.4 -OM Restrictions

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

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

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

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

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

3.12.5 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 written, the compiler may have temporarily assigned that register to some other purpose. Always check results after they are written, 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.

3.13 Problems with Compiler Memory Size

The ISI compilers are advanced optimizing compilers. They are much better that the current generation of "optimizing" microprocessor compilers. In accordance with the greater capability they require more memory. Each compiler requires 300 Kbytes just for the program. They are designed to work best when it has at 1 Mbyte or more of memory available. They 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.

Each compiler 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 68000/10/20/30 code generator. The code generator produces an internal representation of the 68000/10/20/30 machine code to be output for the function. Another optimization phase is then called to modify this machine code. Finally the optimized machine code for the function is output. After the machine code is output, the memory being used for the parse tree and machine code is reclaimed for use in compiling the next function.

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

3.14 Detection of Portability Problems

Many of the problems associated with porting programs to ISI C from other compilers can be detected with the UNIX utility program **lint**(1). 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.

Section 4: Using the 68000 Series Compilers

4.1 68010 Target Environment

The ISI compilers generate code for the 68000 by default. The compile time option **-X12** can be used to generate code for the 68010 instead. The major difference relates to how condition codes are saved and restored. This code is rarely used. It should normally only occur in very complex routines.

4.2 68020/68030 Target Environments

NOTE:

In all cases, the 68030 is treated identically to the 68020.

The compile time option **-X98** enables 68020 code generation. You should also have the **-X12** compile time option on when you use it.

The **-X75** compile time option causes a COFF "-o" file to be generated directly, eliminating the need for an assembly pass. This must be used with the **-X74** (System V) switch.

The **-X122** compile time option generates 68020 instructions as binary words in the assembly code so that the code can beassembled 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 ISI compilers).

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.

4.3 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.

For ISI C, the alignment is as follows:

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

For ISI Pascal, the alignment is as follows:

Data Type	Size	Alignment	
		68000/10	68020 (-X140)
integer	32	word	long
real (default)	64	word	long
real (-X155)	32	word	long

Constant	Value
maxint	2**31-1
minint	-2**31

For ISI FORTRAN, the alignment is as follows:

Data Type	Size	Alignment	
		68000/10	68020 (-X140)
INTEGER	32	word	long
LOGICAL	32	word	long
REAL*4	32	word	long

REAL*8	64	word	long
CHARACTER*1	8	byte	byte
INTEGER*1	8	byte	byte
INTEGER*2	16	word	word
INTEGER*4	32	word	long
LOGICAL*1	8	byte	byte
LOGICAL*2	16	word	word
LOGICAL*4	32	word	long
COMPLEX*8	64	word	long
COMPLEX*16	128	word	long

4.4 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. For instance, in ISI Pascal, a char return value has only the low 8 bits of d0 valid, the high order 24 bits should be considered undefined. 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 compile time option **-X19**.

When software floating point conventions are used, single precision floating point values are returned in d0, double precision in d0/d1.

When 68881 floating point conventions are used (**-X99, -X130**), single and double precision values are returned in fp0.

When Weitek 1167 floating point conventions are used (**-X143, -X130**), single precision values are returned in fp2, double precision in fp2/fp3.

If the **-X130** (full-68020) switch is not used in Weitek or 68881 mode, results are returned in d0/d1 for 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 on the processor, fp0 and fp1 on the 68881, and fp1 through fp13 on the Weitek 1167 are assumed to be destroyed by a call to 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 an 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 (using a6 as the frame pointer) 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 8 bytes must be removed from the stack, a ''addql #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 68000/10 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.

4.4.1 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.

4.5 Local Variables

The ISI compilers may allocate 8 bit and 16 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.

4.6 Assembler Format

As a default, the ISI compilers generate UNIX (MIT) format assembler code. There are several options for the different nuances of the UNIX (MIT) assembler.

Floating point is IEEE (32/64) format by default. The MIT VAX-like format is no longer supported, but can be obtained with **-Z22**.

4.6.1 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'. Both BSD 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. If you want to change it, refer to Section 8 for instructions.

4.7 Motorola Assembler Version

The compile time option (**-X35**) causes the compiler 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 the compiler to convert all identifiers to upper case when they are output in the assembler code.

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

Floating point is IEEE format by default, most significant byte at the lowest address. This obsolete MIT format can be obtained with **-Z22**

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

In FORTRAN, the **-X93** (dbnamedsections) compile time option must be used in order to use common variables. If your assembler does not support named sections then you will be unable to use common variables.

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** 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 inline, 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.

4.8 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.

4.8.1 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. **-X23** if you have a very small stack.

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.

4.8.2 Motorola/AT&T System V

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 sed script.

4.9 Runtime Libraries

The compiler will generate out of line calls for certain operations that are not supported by the 68000/10, such as 32 bit divides and floating point. The source for this part of the runtime library is provided.

4.9.1 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)
ffIti	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
madiv0sub			Floating divide by zero

* No actual code is provided for these routines.

4.9.2 Floating Point, IEEE Support

Floating point constants will be in IEEE format by default. The obsolete MIT format constants can be obtained with **-Z22**.

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.

Section 5: Pascal

5.1 Pascal Standards

There are currently two competing standards for Pascal. The first is the international effort of the British Standards Institute (BSI) and the International Standards Organization (ISO); this standard will be referred to as the BSI/ISO standard. The BSI/ISO standard has two levels, Level 1 is a superset of Level 0. The second is the U.S. effort of the American National Standards Institute (ANSI) and the Institute of Electrical and Electronic Engineers (IEEE); this standard, ANSI/IEEE770X3.97-1983, will be referred to as the ANSI/IEEE standard.

The Integrated Solutions Pascal compiler (ISI Pascal) implements the ANSI/IEEE standard and the BSI/ISO standard Level 0. In addition, ISI Pascal implements many of the extensions present in the Berkeley 4.X BSD **pc**(1) Pascal compiler.

5.2 Extensions to the Basic Pascal Language

ISI Pascal implements a number of extensions to the ISO and ANSI/IEEE Pascal Standards. These features are enabled by default. If you want to run in standard only mode, you must specify the **-X56** compile time option. If you want runtime checking as well, specify **-X57**. On UNIX versions of ISI Pascal, the -s compile time option sets both **-X56** and **-X57**.

5.3 Comment Delimiters

The symbol ''(*'' is equivalent to ''{'' and ''*)'' is equivalent to ''}''.

5.4 Argc and Argv

There is an additional built-in function, argc, which takes no arguments and returns an integer. Argc returns the number of command line arguments on a UNIX Target. The number of command line arguments includes the command name, so argc() is always greater than zero. When run on a non-UNIX system it returns 0.

There is an additional built-in procedure, argv, which takes two arguments. The first argument is an integer which is an argument number. The second argument is a string variable. Argv reads the command line argument specified by the first argument into the string variable specified by the second argument. If the string variable is longer than the corresponding command line argument, the rightmost characters of the command line argument are truncated. If the string variable is shorter than the corresponding command line argument, the string variable is padded on the right with blanks. It is illegal to attempt to access an argument number greater than argc()-1. Argument number zero is the command name.

In non-UNIX versions of Pascal, argv is illegal.

5.5 Set Implementation

Sets of subranges of char, boolean, and enumeration types are implemented as sets of the base type. This allows sets to be operated upon directly with in line code.

The above rule is inadequate for integer subrange sets, because the base type is integer, and a ''set of integer'' would require an exorbitant amount of memory. Therefore, sets of integer type are implemented as sets of some subrange of integers.

When the **-X56** (ISO compatibility) flag is set, all sets of integers are implemented as ''set of 0..255''. When the **-X56** flag is not set, by default, sets of integers are 0..31. The implementation of ''set of 0..31'' is much more efficient than ''set of 0..255''.

5.6 Separate Compilation

ISI Pascal has been extended to make multiple module program development possible. In every executable program there must be one file which consists of a program declaration and a main begin-end block. This is called the program module. This may be the entire program or it may only be a part of the program. If it is only part of the program, some of the procedures, functions, and variables referenced in the main program must be declared ''external''. These external procedures, functions, and variables must be linked with the main program file to obtain a complete program. These external procedures, functions, and variables may be implemented in Pascal, assembly language, or any other programming language.

To implement external procedures, functions, and variables in Pascal, create a file, called a declarations module, which consists only of a series of declarations. The declarations may include procedures, functions, constants, types, and variables. It must not contain a program statement, a main begin-end block, or a final period. The procedures, functions, and variables declared at the top level of a declarations module and at the outermost level of the program module are declared external to the linker.

If the system linker has a limit to the symbol length, this limit must be observed for all external names. Some versions of Pascal prepend an underscore or period to all external names, so this extra character must also be counted in any symbol length restrictions.

5.6.1 External Directive

A new identifier ''external'' is recognized as an alternative to ''forward'' in procedure and function declarations. It specifies that the named procedure or function exists in a separately compiled module. In addition, the identifier ''external'' followed by a semicolon may appear directly after a variable declaration. This specifies that the declared variable actually exists in a separately compiled module.

In UNIX target environments the same variable may be declared in the outer scope of several declarations modules. Each declaration refers to the same variable. That is, in UNIX target environments, the ''external'' declaration is unnecessary for variables. It is, however, required for external procedures and functions.

It is legal to declare a function or procedure external, and then later in the same module to give a declaration of the function or procedure body. If this is done, the function or procedure body must not include a parameter list or return type. The external declaration works exactly like the forward declaration, except that it may only appear in the outer scope, and it is legal for the body of an externally declared function or procedure not to appear in the module.

It is NOT legal to declare a variable external and then to declare it again.

5.6.2 Static Directive

If you want a variable declared in the outer scope of a declarations module or the program module to not be exported to other modules, specify the identifier "static" followed by a semicolon after the declaration of the variable.

5.6.3 Relaxed Declaration Order

Standard Pascal requires all constants to be declared before any types, all types to be declared before any variables, and all variables to be declared before any procedures or functions. To make Pascal easier to use, ISI Pascal allows declarations to be in any order, provided that the declaration appears before any reference to the symbol defined by the declaration (except as allowed by standard Pascal).

5.6.4 #include

If a "#include" appears at the start of a line followed by a file name enclosed in single quotes (apostrophes), the named file is read as input to the program. At the end of the named file input is taken starting at the beginning of the line following the #include. This feature works exactly like #include in ISI C. It is intended to be used to include declarations common to several modules.

5.6.5 Example of a Multiple Module Program

The following program consist of the program module "file1.p", the declarations module "file2.p", and the include file "file3.h". The main program calls the procedure "p" in another module. Notice that the variable "y" in each module is the same, but the variable "x" is different. Notice that an include file is used to hold the declaration of the procedure "p". Keeping external procedure and function declarations in include files makes sure that all calls to the procedure or function are made with the correct arguments. This allows type checking between modules. Notice also that the relaxed declaration order permits procedure and variable declarations to be intermixed for convenience.

file1.p:

```
    program prog;
    #include 'file3.h'
    var
      x: integer; static;
      y: integer;
    begin
      y := 2;
      x := 4;
p(16);
      if x + y <> 20 then
        writeln('ERROR');
    end.
```

file2.p:

```
#include 'file3.h'
var
   x: integer; static;
   y: integer; external;
procedure p{(z: integer)};
begin
   x := 8;
   y := z;
end;
```

file3.h:

```
procedure p(z: integer); external;
```

5.7 C Extensions

Several features of C have been added to ISI Pascal to make it more useful for systems programming.

5.7.1 Hexadecimal Constants

The C syntax of 0x<hex digits> is accepted for hexadecimal integer constants.

5.7.2 Case Sensitivity

For compatibility with C, by default, the compiler is case sensitive. For example, the identifiers ''ABC'' and ''abc'' are distinct. Keywords are recognized only in lower case. For example, the identifier ''BEGIN'' is not a keyword. This default is not compatible with either Pascal standard, both of which specify that case is to be ignored. The compile time option **-X59**9 The compile time option **-X56** sets full standard compatibility and so also sets **-X59**.

5.7.3 Additional Operators

The most commonly used C operators have been added to ISI Pascal. Note that ''/'' and ''/='' represent floating point division for integer operands, unlike in C.

&	The unary address of operator takes one variable operand and returns a pointer to its operand.
~	The one's complement operator takes one integer operand.
&	The bitwise logical and operator takes two integer operands. Same precedence as ''*''.
%	The C modulo function takes two integer operands. This modulo function is the natural remainder from integer divide supplied by the Target. Generally the sign of the result is the sign of the second operand.
>>	The right shift operator takes two integer operands. The first operand is shifted right by the number of bits specified by the second operand. Same precedence as ''*''.

<< The left shift operator takes two integer operands. The first operand is shifted left by the number of bits specified by the second operand. Same precedence as "*".

| The bitwise logical or operator takes two integer operands. Same precedence as "+".

5.8 Input and Output

The predefined file "input" is initially set to the standard input device. The predefined file "output" is initially set to the standard output device. The exact meaning of the standard input and standard output device are defined by the Target system, but they are usually the user's terminal.

5.8.1 Interactive I/O

All input files are organized for interactive I/O. In many implementations of Pascal the program will wait for input to become available on a file when it is reset. Since "input" is reset by default before the program starts executing, many implementations will wait for a line to be typed before they will begin executing the user's program. Under the Pascal library, reseting a file will not cause the program to wait for input to become available. The program will only wait if an access is made to the file buffer variable or information is requested about the file, such as eoln or eof.

5.8.2 Default Field Widths

The default field width for the standard data types are given in the following table.

char	1
integer	12
Boolean	length of true or false
real	25

5.8.3 Second Argument to Reset and Rewrite

The built-in procedures reset and rewrite can have an optional second argument. The second argument must be a string. The string is interpreted as name of the file to be opened. If the second argument is not a string constant, it must contain a character with an ordinal value of 0, this character marks the end of the file name. The string is interpreted by the host operating system and therefore programs using this feature may not be transportable to systems with different file naming conventions.

5.9 Predefined Constants and Types

The following symbols are defined as if the following declarations occurred just before the beginning of each source file.

```
const
        maxint =        2147483648;
type
```

```
integer =       -2147483647..maxint;
char =          chr(0)..chr(127);
```

The type ''real'' is 64-bits. The **-X155** compile time option specifies that ''real'' is 32 bits.

5.10 Float and Double Types

There are two additional types: float and double. Float is a 32 bit real data type, and double is a 64 bit real data type.

5.11 New and Dispose

If initial values for tag fields are specified to ''new'', the record will be allocated to have the minimum possible size for the tag fields specified. It is illegal to change tag fields set in this way, but no checking is performed. If the tag field is changed illegally, serious problems may occur because insufficient memory may have been allocated for the variant designated by the new tag field value. Attempts to store into these new fields may store into adjacent memory allocated to an entirely unrelated variable.

5.12 Record Comparison

Record comparison is implemented. However, it is illegal if either of the operands is not of the maximum variant size. It is wise to limit record assignment and comparison only to records which do not have variants.

5.13 Compile Time Options

Refer to Section 8 of this manual and to the **pc**(1) man page for a complete list of options. Additional options are supported by **ld**(1).

5.14 Interface to the C Library

By default, the names of Pascal external variables, procedures, and functions are accessible from C functions linked with the Pascal program. Names in Pascal can be accessed with the same name in C. When compiling with the **-s** or **-X56** compile time options (ANSI standard features only) or the **-X174** compile time option (Append Underscore), the names of Pascal external procedures and functions are changed by adding an additional underscore ("_"). To access the Pascal function "func" one must use the name "func_" in C. This change in names causes all of the C library functions provided with ISI Pascal to become inaccessible.

If a Pascal program redefines the built in procedure "WRITE" or "READ" it must be compiled with the **-X174**, **-X56**, or **-s** option. The ISI Pascal Runtime Library and the UNIX C library use the names "write" and "read" (to which "WRITE" and "READ" are by default translated) for the basic I/O primitives. If the program redefines these names very strange results (often infinite loops) will occur. The **-X174**, **-X56**, and **-s** compile time options will translate these names to "write_" and "read_" instead, so no redefinition will occur. However, under these options communication between Pascal and C or the C Library becomes rather cumbersome.

Section 6: FORTRAN 77

6.1 FORTRAN Standard

The Integrated Solutions FORTRAN compiler (ISI F77) implements the ANSI FORTRAN-77 (Full Language) standard. It also implements all of the extensions to FORTRAN-77 documented in the 4.XBSD F77 documentation and many of the undocumented extensions in the 4.XBSD F77 implementation. The documentation for ISI F77 consists of the ANSI FORTRAN-77 standard plus the 4.XBSD F77 documentation in addition to this document.

6.2 Extensions to 4.XBSD F77 Documentation

The 4.XBSD F77 documentation is suitable for ISI F77. The same language, switches, and arguments are accepted, the code generated is compatible, and the libraries are identical.

All of the enhancements to the FORTRAN-77 standard included in the BSD F77 implementation have been included in ISI F77.

None of the violations of the FORTRAN-77 standard documented in Section 3 of the BSD F77 manual exist in Integrated Solutions FORTRAN implementation.

The **-II66** command line argument generates proper printer carriage control operation on all files and terminals (but not on pipes).

The **T** and **TL** formats work on terminals using the termcap features of UNIX. **T** and **TL** do not work on pipes (a concept foreign to FORTRAN).

The BSD F77 restrictions on double precision alignment don't apply to ISI F77.

The Dummy Procedure Arguments restriction in BSD F77 don't apply to ISI F77.

The switches **-O** and **-g** are compatible in ISI F77.

The 4.2BSD documentation never gives the calling sequence for the following additional built-in functions:

```
integer*4 function iargc()

subroutine getarg(arg_number, arg_value)
integer*4 arg_number
character*20 arg_value

subroutine getenv(env_name, env_value)
character*(*) env_name
character*20 env_value

integer function or(i1, i2)
```

integer function and(i1, i2)

integer function xor(i1, i2)

integer function rshift(*value, bits*)

integer function lshift(*value, bits*)

integer function not(i1)

6.3 Illegal Programs

Floating point computations are done at compile time to simulate floating point constant computations. Illegal constant computations may cause unrecoverable floating point errors at compile time causing the compiler to crash.

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 unexpected 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. Excessive subroutine size will cause the compiler to grow very large.

Complex multiply, add and subtract calculations are done in line for maximum speed. Each complex multiply generates at least four floating point multiplies, two floating point adds, and two temporary variables. Routines containing in excess of 100 complex operations may require more than a megabyte of memory.

TL (tab left) and **T** (tab) to a column left of the current output column does not work on pipes.

The **-w66** switch does nothing. There are no FORTRAN-66 warnings.

Some intrinsics (as allowed by the standard) cannot be passed as arguments to other procedures. The compiler will not detect this error and an undefined reference will occur at link time.

Many intrinsics are implemented as generics even though the standard specifies that they are not generic.

No check is made for recursive statement functions.

Error messages follow the style of F77. There is no indication of the position within the line of an error.

I/O system error messages are quite verbose but often unhelpful.

6.4 Compile Time Options

Refer to Section 8 of this manual and to the **f77**(1) man page for a list of options. Additional options are supported by **ld**(1).

Section 7: C Language

7.1 Introduction

The ISI C compiler 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 application programs. Several of the most important of these extensions are contained in this section, but this is by no means a complete list.

7.2 Additions to the Basic C Language

7.2.1 Preprocessor

In addition to the standard UNIX preprocessor, ISI C includes a preprocessor which is functionally identical to the UNIX C preprocessor. Unlike PCC which depends on an initial text processing pass by a preprocessor program, ISI C 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.

7.2.2 Backslash v

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

7.2.3 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.

7.2.4 __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.

7.2.5 __FILE__

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

7.2.6 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 have no holes in the memory representation. The alternative is to explicitly initialize each such variable 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. Since the structure or union is copied when it is passed, 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 returned and used 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. If the function call in the interrupt routine then modifies the static return variable that the interrupted routine is using, the interrupted routine, when it continues, 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.

7.2.7 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-expression 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.

7.2.8 The VARARGS(3) Facility

ISI C supports the **varargs**(3) facility. This allows a function to access its parameters in left to right order even if the number and/or types of the parameters are not known until run time. To use the **varargs** facility:

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);
}
```

7.3 Bit Fields

ISI C supports signed and unsigned bit fields. All versions of the MC68000 PCC that we have

examined support only 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 BSD 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.

7.4 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.

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

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

7.5 Unsigned Char and Unsigned Short Int

The data types ''unsigned char'' and ''unsigned short int'' are not defined in the Kernighan & Ritchie C manual, although they are supported by ISI C 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. An attempt has been made to follow the BSD compiler whenever possible.

7.6 asm Statement

The asm statement (for in line assembly code) in ISI C is somewhat different than the asm construct in PCC. In ISI C 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 ISI C is substantially different than the code generated by other compilers it is usually necessary to modify asm statements.

7.7 Compile Time Options

Refer to Section 8 of this manual and to the **cc**(1) man page for a list of options. Additional options are supported by **ld**(1).

Section 8: Compile Time Options

The ISI compilers are configured to enable some of the compile time options described in this section and to disable the rest. Most of the options are documented in this section. Refer to the **cc**(1), **pc**(1), and **f77**(1) man pages for a complete list of options.

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. 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 unpredictable 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. 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.

Integrated Solutions does not guarantee that the compilers will act as you expect when using these options. We retain the right to change any options 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 (ISI C) If this option is given, comments are output in the preprocessor output. The default is to strip comments from the output.

(ISI Pascal and FORTRAN) Turn on runtime checking of subranges and array bounds. The code will be much slower under this option.

-Dname (ISI C)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 (ISI FORTRAN) (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 (ISI C) Define ''name'' to the preprocessor with the value ''string''. This is equivalent to putting ''#define name string'' at the top of the source file.

-Dname=string (ISI FORTRAN)(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.

-DCODESEG=xx (ISI C) (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 (ISI C) 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 ''cpp'', so use ''cpp'' for big jobs.

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

-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.

- F (ISI FORTRAN) (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 debugger information 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 in some programs. This makes program debugging difficult. When debugging a program this option should be used. This option does not imply ''**-g**''.

-i2 (ISI FORTRAN) Make the type INTEGER be INTEGER*2.

-Istring Include file names which are not absolute (do not start with ''/'') 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 (ISI FORTRAN) (UNIX Host only) Preprocess files whose names begin with ''.r'' with ''m4'' before running the ratfor preprocessor.

-O The **-O** option activates the optimizers. These are safe for use on all programs, except the loop optimizer.

-OL Optimize the program to be as fast as possible even if it is necessary to make the program larger. 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 and 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 bot not necessarily faster. After experimenting with a program, it is possible to discover which modules benefit from **-OL** 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.

-OM 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.

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

-OML This option is equivalent to **-OLM**.

-onetrip (ISI FORTRAN) 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.

-p (UNIX Host and Target only) Generate calls for execution profiling and links the code with routines which support *prof*(1).

-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**" "**-F**" (FORTRAN) is 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.

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

-s (ISI Pascal) (UNIX Host only) Compile the program in the ANSI compatible mode. Generate errors for the use of extensions to the ANSI Pascal Standard. This also changes the default supported range for set of subrange of integer from 0..31 to 0..255. The code for sets of subranges of integers under this option is much worse than under the default.

-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 (ISI FORTRAN) Make the default data type for undeclared variables be "undefined". As if "IMPLICIT UNDEFINED(A-Z)" was placed at the top of the file.

-U (ISI FORTRAN) Do not convert upper case 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 (ISI C) 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.

-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 Use a ''skip'' pseudo-op to allocate blank spaces in the assembly code. This option formerly indicated prepending an underscore to all programmer defined names, which is now the default. See also **-X58**.

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

-X22 Use the obsolete VAX format for floating point constants rather than the default IEEE format.

-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 filenames 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 (ISI C), "case" statements (ISI Pascal), and computed goto statements (ISI FORTRAN) 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, but is not recommended.

-X37 Emit a warning when dead code is eliminated.

-X38 When using the obsolete 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.

-X55 Make fields of type int, short, and char be signed. The default is for all fields to be unsigned.

-X56 (ISI Pascal) Allow only ANSI/IEEE features. Generate error messages for any non-standard constructs which are used.

-X58 Do not put an underscore in front of the names of global variables and procedures.

-X59 (ISI Pascal) Turn off case sensitivity.

-X74 The target system is UNIX System V.

-X75 (System V only) Generate object files instead of assembly files.

-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 ISI C.

-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 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 i 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**, **-X129**, and **-X130**.

-X100 to -X105 Customer defined options.

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

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

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

-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.

-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.

-X138 Do not use the ''lcomm'' (local common) directive.

-X140 Use MC68020 alignment rules. If you plan to upgrade to a 68020 system, it is a 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. This option is a good idea for ISI Pascal and ISI FORTRAN.

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

-X155 (ISI Pascal) Make the type ''real'' be a 32 bit real. The default is for ''real'' to be 64 bits

-X156 (ISI Pascal) Export the names of variables in the outer most scope of a Pascal main program for use with other modules of a multiple module program. The default is for the variables in the outer most scope of a Pascal main program to be static variables inaccessible to external modules.

-X157 (ISI Pascal) Subrange types in unpacked records are packed into bytes or 16 bit fields if possible. The default is for all subranges to occupy 32 bits in unpacked records.

-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. If this option is used, the compiler may crash or otherwise terminate abnormally.

-X167 Unsupported option, Evaluate expressions involving only float operands as float (not double). Do not expand float arguments to double. Do not expand float return values to double.

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

-X174 (ISI Pascal) Append an underscore to the names of all external procedures and functions to avoid name conflicts with unix library routines. Set by default if -s specified.

-X187 Suppress output from #ident.

-X188 Use FORTRAN mixed mode expression evaluation rules. In particular, does float*float computation in single precision; does not convert to double precision before performing operation.

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

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

-X216 Invoke pipeline reordering.

-X217 Invoke local optimizations, e.g., inline divide.

-X218 Generate code which takes into account the early 68881 ''FSQRT'' bug.

-X219 Suppress elimination of jumps to jumps.

-X220 (ISI Pascal) Generate code to check that correct tag field value is present when a field of a variant is referenced.

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

-X233 Functions which return the type ''float'' return a single precision value, not a double precision value.

-X237 Apply associative rules in common subexpression elimination.

-X241 (ISI Pascal) Sets of unknown size will be allocated 256 bits rather than 32.

-X243 The ''fsglmul'' and ''fsgldiv'' instructions will be given a size of ''.s'' rather than ''.x'' in UNIX assembly output.

-X255 Print a brief description of enable -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 peephole phase until code no longer improves.

-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.

-X300 Pad 68881 instructions with "FPNOP" instructions so interrupts are taken at the correct position.

-X304 Truncate names to eight characters on input.

-X306 C "asm" inline directive not recognized.

-X308 Perform tail recursion optimizations.

-X311 Don't make multiple copies of blocks in merge blocks phase.

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

-X316 Enable all ANSI C extensions which are sensible in a UNIX environment.

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

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

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

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

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

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

-X334 The usual arithmetic rules apply to operator assignments, as ANSI requires, rather than the Berkeley "left side prevails" rule. For example, "charvar *=0.5" will be performed using floating arithmetic.

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

-X350 In ACSI C, allow /**/ to be a concatenation operator in a macro, as it in the portable C compiler.

-X352 Don't extend float arguments to double in order to pass them to functions.

-X353 Perform common subexpression analysis twice. Rarely useful.

-X370 Output line numbers in the assembly file.

-X380 Parentheses behave as they are said to in (some versions of) the proposed ANSI C standard; that is, the compiler may not associate over them.

Appendix A: Man Pages

The man pages for the Integrated Solutions' compilers are included in this section.

NAME

cc – C compiler

SYNOPSIS

cc [*options*] *files*

DESCRIPTION

cc is an optimizing C compiler. It accepts several types of arguments.

Arguments whose names end with ".c" are C source programs. They are compiled and left in a ".o" file in the working directory.

Arguments whose names end with ".s" are assembly language source programs. They are assembled and left in a ".o" file in the working directory.

The ".o" file is deleted if a single source file is compiled and linked.

cc creates ".s" files for each module only when the user compiles with the –S option.

OPTIONS

cc accepts the options listed below. Additional options are supported by **ld**(1).

–ansi Provides basic compiler support for ANSI C but does not achieve full conformance due to rounding incompatibilities when allocating floating point variables to machine registers.

–ansiconform
Provides preliminary conformance to the proposed ANSI C draft. However, this option supports fewer floating point optimizations than does the **–ansi** flag.

–c Compiles to the ".o" level only. Does not link.

–C Does not strip comments from the preprocessor output. Must be used with the **–E** option.

–D*name=def*

–D*name* Defines the *name* to the preprocessor, as if by "define". If no definition is given, the name is defined as "1".

–E Places the output of the preprocessor in the standard output file. Does not compile the program. This is useful for debugging preprocessor macros. Use */lib/cpp* for big jobs because the integrated preprocessor cannot generate output as fast as /f2/lib/cpp/f1.

–f Generates code for a 68881 coprocessor. By default many of the functions supported by the 68881 will be inline as well; use -Z129 if you want a transcendental call to go to the routine instead. The **–f** option does not support profiling.

–g Generates BSD style debugger information in the assembly file for use with a debugger such as **dbx**(2).

–ga Generates a stack frame for every routine, regardless of need.

–k Prevents the compiler from optimizing an "and" with a single bit into a BTST instruction. Some I/O devices require word access to their registers while BTST is a byte access instruction.

–I*dir* Looks for "#include" files whose names do not begin with "/" first in the directory of the *file* argument, then in directories named in **–I** options, then in /f2/usr/include/f1.

–o *output* Names the final output file *output*. If this option is used, the file a.out is left undisturbed.

−O Performs various speed optimizations, such as moving constant expressions out of loops. Generally this makes your programs somewhat larger. If their performance is not loop bound, they can become slower as well.

−O2 This flag is an older version of the **−OM** flag. It allows backwards compatibility.

−OL Optimizes the program to be as fast as possible even if it makes the program larger. In particular, most of the available resources are allocated to optimizations of the innermost loops. It is counter-productive to specify **−OL** on code that contains no loops or that is rarely executed as it makes the whole program larger but not necessarily faster.

−OM Specifies that memory locations do not change except by explicit stores. 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 that can change them asynchronously with respect to the correct process.

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

−OML This option is equivalent to **-OL** and **-OM**.

−p Generates profiling code and links the code with routines that support **prof**(1). However, **−p** does not support profiling for programs compiled with the **−f** (floating point) option. (Not implemented.)

−pg Generates profiling code similar to **−p** but links with a more comprehensive profiling mechanism that supports **gprof**(1). Like the **−p** option, **−pg** does not support profiling for programs compiled with the **−f** (floating point) option. (Not implemented.)

−R Makes initialized variables part of the text segment and passes them on to **as**.

−S Compiles the named C programs and leaves the assembler-language output on corresponding files suffixed ".s".

−U*name* Removes any initial definition of *name*.

−v Turns on verbose mode so **cc** prints out the arguments to each phase of compilation and linking.

−w Suppresses warning messages.

−X*n* Turns on option *n* (an integer constant). Numerous options are available for such things as signed bit fields, short return types, etc. Section 8 of the *UNIX Compiler Guide: C, Pascal, FORTRAN 77* describes these options.

−Z*n* Turns off option number *n* that is on by default or was turned on with the **X** option.

FILES

file.[cs]	input file
file.o	object file
a.out	loaded output
/bin/as	assembler
/lib/cpp	C preprocessor
/lib/crt0.o	startup code
/usr/lib/ccom	C compiler
/usr/include	standard directory for "#include" header files
/usr/lib/libc.a	UNIX standard I/O library
/usr/lib/libc_p.a	profiling UNIX standard I/O library
/usr/lib/gcrt0.o*	profiling startup code for **gprof**(1)
/lib/mcrt0.o*	profiling startup code for **prof**(1)
/usr/lib/libm.a	transcendental floating point math library
/usr/lib/libm_p.a	profiling transcendental floating point library

SEE ALSO

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978
B. W. Kernighan, *Programming in C—a tutorial*
D. M. Ritchie, *C Reference Manual*
UNIX Compiler Guide: C, Pascal, FORTRAN 77
as(1), **prof**(1), **gprof**(1), **adb**(1), **dbx**(1), **ld**(1), **f77**(1), **pc**(1)

DIAGNOSTICS

The diagnostics produced by the C compiler are self-explanatory and similar to those produced by the BSD compiler. Occasional messages can be produced by the assembler or loader.

NAME

f77 – Fortran 77 compiler

SYNOPSIS

f77 [*options*] *files*

DESCRIPTION

f77 is an optimizing Fortran 77 compiler. **f77** accepts several types of arguments:

Arguments whose names end with ''.f'' are Fortran 77 source programs. They are compiled and left in a ''.o'' file in the working directory.

Arguments whose names end with ''.F'' are preprocessed with the C preprocessor before they are compiled by **f77**.

Arguments whose names end with ''.r'' are Ratfor programs. They are processed by **/usr/bin/ratfor** before they are compiled by **f77.**

Arguments whose names end with ''.s'' are assembly language source programs. They are assembled and left in a ''.o'' file in the working directory.

The ''.o'' file is deleted if a single source file is compiled and linked.

f77 creates ''.s'' files for each module if the user compiles with the **–S** option.

OPTIONS

f77 accepts the options listed below. Additional options are supported by **ld**(1).

–20 Generates code for a 68020 CPU. To maintain compatibility with old code, the alignment rules are not changed unless you specify -X134. This alignment forces longwords to 32 bit boundaries.

–c Compiles to the ''.o'' level only. Does not link.

–C Does not strip comments from the preprocessor output. Must be used with the **–E** option.

–D*name*=*def*

–D*name* Defines the *name* to the preprocessor, as if by ''define''. If no definition is given, the name is defined as ''1''.

–E Places the output of the preprocessor in the standard output file. Does not compile the program. This is useful for debugging preprocessor macros. Use */lib/cpp* for big jobs because the integrated preprocessor cannot generate output as fast as */lib/cpp*,

–f Generates code for a 68881 coprocessor. By default many of the functions supported by the 68881 will be inline as well. Use **-Z139** if you want a transcendental call to go to the routine instead. The **–f** option does not support profiling.

–F Preprocess the ''.r'' or ''.F'' files given to ''.f''. Does not compile.

–g Generates BSD style debugger information in the assembly file for use with a debugger such as **dbx**(2).

–ga Generates a stack frame for every routine, regardless of need.

–k Prevents the compiler from optimizing an ''and'' with a single bit into a BTST instruction. Some devices require word access to their registers while BTST is a byte access instruction.

–I*dir* Looks for ''#include'' files whose names do not begin with ''/'' first in the directory of the *file* argument, then in directories named in **–I** options, then in */usr/include*.

–m Preprocesses ratfor programs with M4.

–o *output* Names the final output file *output*. If you use this option, **f77** leaves the file ''a.out'' undisturbed.

–onetrip Ensures that DO loops are executed at least once.

–O Performs various speed optimizations while avoiding loop optimizations that expand code and memory optimizations that are unsuitable for asynchronous memory accesses.

–O2 This option is an older version of the *–OM* flag. It allows backwards compatibility.

–OL Optimizes the program to be as fast as possible even if it makes the program larger. In particular, most of the available resources are allocated to optimizations of the innermost loops. It is counter-productive to specify **–OL** on code that contains no loops or that is rarely executed as it makes the whole program larger but not necessarily faster.

–OM Specifies that memory locations do not change except by explicit stores. 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 correct process.

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

–OML This option is equivalent to -OL and -OM.

–p Generates profiling code in a manner similar to **cc**(1) and links the code with routines that support **prof**(1). However, **–p** does not support programs compiled with the **–f** (floating point) option. (Not implemented.)

–pg Generates profiling code similar to **–p** but links with a more comprehensive profiling mechanism that supports **gprof**(1). Like the **–p** option, **–pg** does not support profiling for programs compiled with the **–f** (floating point) option. (Not implemented.)

–R Makes initialized variables part of the text segment and passes them on to **as**.

–R*string* Passes *string* along to **ratfor** as an option.

–S Compiles the named programs and leaves the assembler-language output on corresponding files suffixed ''.s''.

–u Makes all undeclared variables ''undefined.''

–U Retains character case significance. By default, identifiers are converted to lower case.

–U*name* Removes any initial definition of *name*.

–v Turns on verbose mode so **f77** prints out the arguments to each phase of compilation and linking.

–w Suppresses warning messages.

–X*n* Turns on option number *n* (an integer constant). Numerous options are available for such things as signed bit fields, short return types, etc. Section 8 of the *UNIX Compiler Guide: C, Pascal, FORTRAN 77* describes these options.

–Z*n* Turns off option *n* that is on by default or was turned on with the –Xoption.

FILES

file.[fFresc]	source input file
file.o	object file
a.out	loaded output
/bin/as	assembler
/lib/cpp	C preprocessor
/usr/lib/libc.a	UNIX standard I/O library
/usr/lib/libc_p.a	profiling UNIX standard I/O library
/usr/lib/fcom	Fortran compiler
/usr/lib/gcrt0.o*	profiling startup code - IEEE floating point
/usr/lib/libF77.a	Fortran intrinsic function library
/usr/lib/libF77_p.a	profiling intrinsic function library
/usr/lib/libI66.a	Fortran I/O library for Fortran 66
/usr/lib/libI77.a	Fortran I/O library
/usr/lib/libI77_p.a	profiling Fortran I/O library
/usr/lib/libU77.a	UNIX interface library
/usr/lib/libU77_p.a	profiling UNIX interface library
/usr/lib/libmU77.a	UNIX interface library compiled for the 68020/68881
/usr/lib/libmF77.a	Fortran intrinsic functions for the 68020/68881
/usr/lib/libmI77.a	Fortran I/O library for the 68020/68881
/usr/lib/libm.a	Intrinsic floating point math library
/usr/lib/libm_p.a	profiling intrinsic floating point library
mon.out	file produced for analysis by **prof**(1)
gmon.out	file produced for analysis by **gprof**(1)

SEE ALSO

UNIX Compiler Guide: C, Pascal, FORTRAN 77

as(1), **cc**(1), **pc**(1), **prof**(1), **gprof**(1), **adb**(1), **dbx**(1), **ld**(1), **ratfor**(1), **m4**(1)

DIAGNOSTICS

The diagnostics produced by the **f77** compiler are self-explanatory and similar to those produced by the BSD f77 compiler. Occasional messages can be produced by the assembler or loader.

NAME

pc – Pascal compiler

SYNOPSIS

pc [*options*] *files*

DESCRIPTION

pc is an optimizing ISO and/or ANSI/IEEE Pascal compiler. **pc** accepts several types of arguments:

Arguments whose names end with ".p" are Pascal source programs. They are compiled and left in a ".o" file in the working directory.

Arguments whose names end with ".f" are Fortran 77 source programs. They are compiled and left in a ".o" file in the working directory.

The **f77**(1) and **cc**(1) commands are normally used for Fortran and C programs. When multi-lingual programs need to be compiled, you can use any one of the three commands (**cc**, **pc**, or **f77**).

Arguments whose names end with ".s" are assembly language source programs. They are assembled and left in a ".o" file in the working directory.

If you compile and link a single source file, **pc** deletes the ".o" file.

pc creates ".s" files for each module only if the user compiles with the **–S** option.

OPTIONS

pc accepts the following options. Additional options are supported by **ld**(1).

–20 Generates code for a 68020 CPU. To maintain compatibility with old code, the alignment rules are not changed unless you specify -X134. This alignment forces longwords to 32 bit boundaries,.

–c Compiles to the ".o" level only. Does not link.

–C Compiles code to perform runtime checks, verifies **assert** calls, and initializes all variables to zero as in **pi**.

–f Generates code for a 68881 coprocessor. By default many of the functions supported by the 68881 will be inline as well. Use -Z129 if you want a transcendental call to go to the routine instead. The **–f** option does not support profiling.

–g Generates BSD style debugger information in the assembly file for use with a debugger such as **dbx**(2).

–ga Generates a stack frame for every routine, regardless of need.

–I*dir* Looks for ".include" files whose names do not begin with "/" first in the directory of the *file* argument, then in directories named in **–I** options, then in /f2/usr/include/f1.

–o *output* Names the final output file *output*. With this option, **pc** leaves the file "a.out" undisturbed. This does not apply to assembly output.

–O Performs various speed optimizations while avoiding loop optimizations that expand code and memory optimizations that are unsuitable for asynchronous memory accesses.

–O2 This option is an older version of the *–OM* flag. It allows backwards compatibility.

–OL Optimizes the program to be as fast as possible even if it makes the program larger. In particular, most of the available resources are allocated to optimizations of the innermost loops. It is counter-productive to specify **–OL** on code that contains no loops or that is rarely executed as it makes the whole program larger but not necessarily faster.

–OM Specifies that memory locations do not change except by explicit stores. 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 that can change them asynchronously with respect to the correct process.

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

–OML This option is equivalent to -OL and -OM.

–p Generates profiling code in a manner similar to **cc**(1) and links the code with routines that support **prof**(1). However, **–p** does not support programs compiled with the **–f** (floating point) option. (Not implemented.)

–pg Generates profiling code similar to **–p** but links with a more involved profiling mechanism that supports **gprof**(1). Like the **–p** option, **–pg** does not support profiling for programs compiled with the **–f** (floating point) option. (Not implemented.)

–R Makes initialized variables part of the text segment and passes them on to **as**.

–S Compiles the named C programs and leaves the assembler-language output on corresponding files suffixed ".s".

–w Suppresses warning messages.

–X*n* Turns on option *n* (an integer constant). Numerous options are available for such things as signed bit fields, short return types, etc. Section 8 of the *UNIX Compiler Guide: C, Pascal, FORTRAN 77* describes these options.

–Z*n* Turns off option *n* (on by default or turned on with the **X** option).

FILES

file.p	Pascal source file
/bin/as	assembler
/usr/lib/pcom	Pascal compiler
/usr/lib/libc.a	UNIX standard I/O library
/usr/lib/libc_p.a	profiling UNIX standard I/O library
/usr/lib/libmc.a	UNIX standard I/O library for the 68020/68881
/usr/lib/libpc.a	Pascal I/O library with math intrinsics
/usr/lib/libpc_p.a	profiling Pascal I/O library with math intrinsics
/usr/lib/libm.a	intrinsic floating point math library
/usr/lib/libm_p.a	profiling intrinsic floating point library
/usr/lib/libmm.a	68020/68881 intrinsic floating point math library
/usr/lib/gcrt0.o*	profiling startup code
/lib/mcrt0.o*	profiling startup code
mon.out	file produced for analysis by **prof**(1)
gmon.out	file produced for analysis by **gprof**(1)

SEE ALSO

Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, Springer-Verlag, 1978
American National Standard Pascal Computer Programming Language,
IEEE/John Wiley-Interscience, 1983
UNIX Compiler Guide: C, Pascal, FORTRAN 77
as(1), **prof**(1), **gprof**(1), **adb**(1), **dbx**(1), **ld**(1), **cc**(1), **f77**(1), **cc**(1)

DIAGNOSTICS

The diagnostics produced by the Pascal compiler are self-explanatory and similar to those produced by the BSD **pc** compiler. Occasional messages can be produced by the assembler or loader.

Integrated Solutions

DOCUMENTATION COMMENTS

Please take a minute to comment on the accuracy and completeness of this manual. Your assistance will help us to better identify and respond to specific documentation issues. If necessary, you may attach an additional page with comments. Thank you in advance for your cooperation.

Manual Title: *UNIX Compiler Manual*	Part Number: *490292 Rev. A*

Name: _____ Title: _____

Company: _____ Phone: ( ) _____

Address: _____

City: _____ State: _____ Zip Code: _____

1. Please rate this manual for the following:

	Poor	Fair	Good	Excellent
Clarity	☐	☐	☐	☐
Completeness	☐	☐	☐	☐
Organization	☐	☐	☐	☐
Technical Content/Accuracy	☐	☐	☐	☐
Readability	☐	☐	☐	☐

Please comment: _____

2. Does this manual contain enough examples and figures?

Yes ☐ No ☐

Please comment: _____

3. Is any information missing from this manual?

Yes ☐ No ☐

Please comment: _____

4. Is this manual adequate for your purposes?

Yes ☐ No ☐

Please comment on how this manual can be improved: _____

Fold Down First

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

First-Class Mail Permit No. 7628 San Jose, California 95131

Postage will be paid by addressee

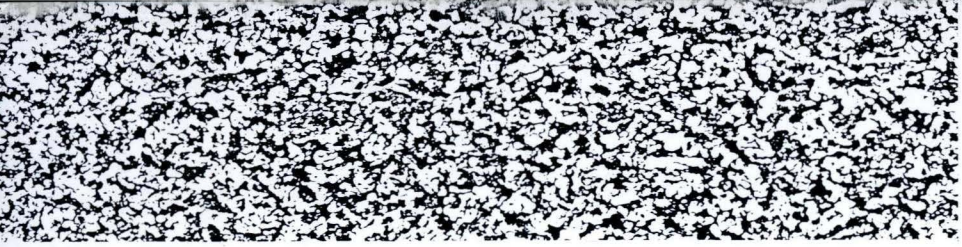
Integrated Solutions

An NBI Company

ATTN: Technical Publications Manager
1140 Ringwood Court
San Jose, CA 95131

Fold Up Second

Staple Here



Integrated Solutions
1140 Ringwood Court
San Jose, CA 95131
408 943-1902
Telex 499 6929