April 1995
Order Number: 312491-003

Paragon™ System
Fortran Compiler User's Guide

Intel® Corporation

WARNING

Some of the circuitry inside this system operates at hazardous energy and electric shock voltage levels. To avoid the risk of personal injury due to contact with an energy hazard, or risk of electric shock, do not enter any portion of this system unless it is intended to be accessible without the use of a tool. The areas that are considered accessible are the outer enclosure and the area just inside the front door when all of the front panels are installed, and the front of the diagnostic station. There are no user serviceable areas inside the system. Refer any need for such access only to technical personnel that have been qualified by Intel Corporation.

CAUTION

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

LIMITED RIGHTS

Preface

This manual describes the Paragon™ system Fortran compiler and driver. This manual assumes that you are an application programmer proficient in the Fortran language and the UNIX operating system.

Organization

Chapter 1	Introduces the software development environment and shows how to create executable files from Fortran source code. This chapter contains enough information to get you started creating executable files for the Paragon system.
Chapter 2	Describes **if77**, the command for compiling, assembling, and linking Fortran source code for execution on the Paragon system.
Chapter 3	Gives you a strategy for using the compiler's optimization features to help maximize the single-node performance of your programs.
Chapter 4	Tells how to use the compiler's function inliner.
Chapter 5	Tells how to write Fortran routines that are callable from C and how to call C functions from Fortran.
Chapter 6	Describes the language that the Fortran compiler accepts (ANSI Fortran 77) and extensions to the standard language (i.e., features and capabilities not defined for the ANSI standard language, such as VAX and Cray extensions).
Appendix A	Lists the error messages generated by the compiler, indicating each message's severity and, where appropriate, the probable cause of the error and how to correct it.
Appendix B	Lists the error messages generated by the Fortran runtime system, indicating, where appropriate, the probable cause of the error and how to correct it.

Appendix C Describes the internal structure of the compiler, with special emphasis on the vectorizer and optimizer.

Appendix D Contains reference manual pages for the Paragon software development commands and compiler-related system calls.

Notational Conventions

This manual uses the following notational conventions:

Bold Identifies command names and switches, system call names, reserved words, and other items that must be entered exactly as shown.

Italic Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

`Bold-Italic-Monospace`
Identifies user input (what you enter in response to some prompt).

`Bold-Monospace`
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

```
<Break>     <s>          <Ctrl-Alt-Del>
```

`[ ]` Surround optional items.

... Indicate that the preceding item may be repeated.

| Separates two or more items of which you may select only one.

`{ }` Surround two or more items of which you must select one.

Applicable Documents

For more information, refer to the *Paragon™ System Technical Documentation Guide.*

Comments and Assistance

Intel Scalable Systems Division is eager to hear of your experiences with our new software product. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
Phone: 800-421-2823
Internet: support@ssd.intel.com

Intel Corporation Italia s.p.a.
Milanofiori Palazzo
20090 Assago
Milano
Italy
1678 77203 (toll free)

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

Intel Japan K.K.
Scalable Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

United Kingdom Intel Corporation (UK) Ltd.
Scalable Systems Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056 (*answered in French*)
(44) 793 431062 (*answered in Italian*)
(44) 793 480874 (*answered in German*)
(44) 793 495108 (*answered in English*)

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
85622 Feldkirchen bei Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Scalable Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 677-7600 (Monday through Friday, 8 AM to 5 PM Pacific Time)
Fax: (503) 677-9147

Table of Contents

Chapter 1
Getting Started

Chapter 2
The if77 Driver

Chapter 4
Using the Inliner

Chapter 5
Interfacing Fortran and C

Chapter 6
Extensions to ANSI Fortran

Appendix A
Compiler Error Messages

Appendix B
Runtime Error Messages

Appendix C
Compiler Internal Structure

Appendix D
Manual Pages

List of Illustrations

List of Tables

Getting Started 1

This chapter introduces the software development environment and shows how to create executable files from Fortran source code.

This chapter contains enough information to get you started using the compiler driver to create executable files from Fortran source code that conforms to the ANSI Fortran 77 standard. For information on extensions to the standard language, refer to Chapter 6.

The Software Development Environment

The software development environment consists of a Paragon™ system and its supporting software.

System Hardware

A Paragon™ system consists of an ensemble of *nodes* connected by a high-speed internal network. Each node contains one or more i860™ processors and 16M bytes or more of memory. Each node's memory is directly accessible only to that node; nodes share information with other nodes by passing *messages* over the network. All nodes run the operating system. Multiple processes can run on each node, and each process can have multiple *threads* (also known as *lightweight processes*).

The nodes appear to the programmer and user to be a single system. For example, every process has a different process ID from any other process running anywhere in the system, no matter what node the processes are running on. In addition, all nodes share a single file system and have equal access to the system's I/O facilities.

The nodes of the system are divided into a *service partition* and a *compute partition*. The compute partition may be subdivided into smaller partitions.

- Nodes in the service partition run a variety of system services, such as user shells, editors, and compilers. Programs run in the service partition consist of single, independent processes.
- Nodes in the compute partition run *parallel applications*—user-written programs that consist of groups of cooperating processes. All the processes in a single application run in the same compute partition; they may or may not use all the processors in the partition.

See the *Paragon™ System User's Guide* for more information about partitions and applications.

System Software

The system software is a complete implementation of the OSF/1 operating system. It includes all the calls and commands of OSF/1, plus extensions for parallel programming.

- For information on the standard OSF/1 calls and commands, see the *OSF/1 User's Guide*, *OSF/1 Command Reference*, and *OSF/1 Programmer's Reference*.
- For information on the parallel extensions, see the *Paragon™ System User's Guide*, *Paragon™ System Commands Reference Manual*, and *Paragon™ System Fortran Calls Reference Manual*.

Software Development Environments

The operating system includes a complete set of commands for compiling, linking, executing, and debugging parallel applications. These commands are available in two different software development environments:

- The *cross-development environment* runs both on the Paragon system and on supported workstations.
- The *native development environment* runs only on the Paragon system itself.

Table 1-1 lists the commands in the two software development environments.

Table 1-1. Software Development Commands

Name in Cross-Development Environment	Name in Native Environment	Description
ar860	**ar**	Manages object code libraries
as860	**as**	Assembles i860™ source code
dump860	**dump860**	Dumps object files
if77	**f77**	Compiles Fortran programs
ifixlib	**ifixlib**	Updates inliner library directories.
ld860	**ld**	Links object files
mac860	**mac**	Preprocesses assembly-language programs
nm860	**nm**	Displays symbol table (name list) information
size860	**size**	Displays section sizes of object files
strip860	**strip**	Strips symbol information from object files

With minor exceptions, these commands work the same in both environments and on all supported hardware platforms. The biggest difference between the two environments is the names of the commands, as shown in Table 1-1 For convenience, the cross development name is also supported in the native environment. Where other differences exist, they are noted in Appendix D.

NOTE

This manual uses the cross-development names for these commands. However, except where noted, all discussions of the cross-development command names apply equally to the corresponding native command names.

This manual gives complete information on the compiler and provides manual pages for the other commands shown in Table 1-1 The Paragon system also provides a symbolic debugger, parallel performance analyzer, and other software tools; for information on these tools, see the *Paragon™ System Application Tools User's Guide.*

Compiler Driver

The Fortran driver provides an interface to the compiler, assembler, and linker that makes it easy to produce executable files from Fortran source code. For example:

- It automatically sets appropriate compiler, assembler, and linker switches.
- It lets you pass switches directly to the assembler and linker. All functionality of the **as860** assembler and **ld860** linker is available through the driver.
- It lets you stop after the preprocessor, compiler, assembler, or linker steps.
- It lets you retain intermediate files.

The driver creates an executable file for execution on a Paragon system node.

The **if77** command invokes the Fortran driver. For example, the following command line compiles, assembles, and links the Fortran source code in the file *myprog.f* (using the default driver switches) and leaves an executable version of the program in the file *a.out*:

```
% if77 myprog.f
```

Chapter 2 describes the **if77** driver in detail, and Appendix D contains a manual page for **if77**.

NOTE

You can invoke the assembler and linker directly (as indicated in the next two sections). However, if you do so, you must explicitly specify switches, libraries, and other information that is provided automatically by the driver. Therefore, such usage is recommended for advanced users only.

i860™ Assembler

The **as860** command invokes the i860 assembler to assemble the output of the compiler. For example, the following command line assembles the file *myprog.s* and leaves the resulting object code in the file *myprog.o*:

```
% as860 myprog.s
```

For more information on using the i860 assembler, refer to the **as860** manual page in Appendix D.

i860™ Linker

The **ld860** command invokes the i860 linker to link the output of the **as860** assembler. For example, the following command line links the file *myprog.o* with the library *mylib.a* and leaves the resulting executable in the file *a.out*:

```
% ld860 myprog.o mylib.a
```

For more information on using the i860 linker, refer to the **ld860** manual page in Appendix D.

Execution Environments

The software tools can create executable files for execution on one node or multiple nodes.

Running on a Single Node

By default, the **if77** driver creates a file for execution on a single node. For example, the following command line compiles *myprog.f* to the executable *a.out*:

```
% if77 myprog.f
```

When you run the resulting executable by typing **a.out** on the Paragon system, it runs on one node in the service partition.

Running on Multiple Nodes

To run a program on multiple nodes, you must use calls from the library *libnx.a*. This library contains the calls that you use to start processes on multiple nodes and communicate with processes running on other nodes. (All of the calls in *libnx.a* are described in the *Paragon™ System Fortran Calls Reference Manual*.)

The **if77** driver does not automatically search *libnx.a*. To search *libnx.a*, you can use either the **-nx** or **-lnx** switch when linking:

- The **-nx** switch links in *libnx.a*, *libmach.a*, and *options/autoinit.o*, and creates an executable that automatically starts itself on multiple nodes when invoked. For example, the following command line compiles *myprog.f* to the executable *a.out*:

  ```
  % if77 -nx myprog.f
  ```

When you run the resulting executable by typing **a.out** on the Paragon system, it runs on all the nodes in your default partition. You can use the command line switches and environment variables described in the *Paragon™ System User's Guide* to control its execution characteristics.

For compatibility with the iPSC system, the **-node** switch is equivalent to **-nx**. For example, the following command is equivalent to the previous command:

```
% if77 -node myprog.f
```

However, continued support for this switch is not guaranteed.

- The **-lnx** switch links in *libnx.a*, but you should use the **-nx** switch if your program is going to run on multiple nodes. For example, the following command line compiles *myprog.f* to the executable *a.out*:

```
% if77 myprog.f -lnx
```

Note that **-lnx** must appear *after* the filenames of any source or object files that use calls from *libnx.a*.

Debugging

To debug programs, use the Interactive Parallel Debugger (IPD). IPD can debug any program that runs on the Paragon system.

To compile an application for debugging, use the **-g** compile-time switch. The **-g** switch is equivalent to the following switches:

-O0	Do not optimize code.
-Mdebug	Include symbol table and line table information.
-Mframe	Include stack frame traceback information.

If you do not use the **-g** switch you can still debug the program, but debugging will be limited. For example, at optimization levels higher than 0, access to individual source lines will be decreased, and display or modification of variables and registers will probably have unpredictable results. In addition, without stack frame traceback information turned on, the information displayed by the debugger for a stack traceback will be incomplete.

For more information on using the Interactive Parallel Debugger, refer to the *Paragon™ System Interactive Parallel Debugger Reference Manual* and the *Paragon™ System Application Tools User's Guide*.

Example Driver Command Lines

The following example command lines show how to use the **if77** driver to perform typical tasks:

- Compile and link for a single node, leaving the executable in a file called *x*:

```
% if77 -o x x.f
```

- Compile and link for multiple nodes with automatic start-up:

```
% if77 -nx -o x x.f
```

- Compile and link in *libnx.a*.

```
% if77 -o x x.f -lnx
```

- Compile, but skip assemble and link steps (**-S**); leaves assembly language output in file *x.s*:

```
% if77 -S x.f
```

- Compile and assemble, but skip link step (**-c**); leaves object output in file *x.o*:

```
% if77 -c x.f
```

- Compile and assemble with optimizations:

```
% if77 -c -O2 x.f          (level 2 - global optimizations only)
% if77 -c -O3 x.f          (level 3 - adds software pipelining)
% if77 -c -O3 -Mvect x.f   (level 3 optimizations plus vectorization)
```

See Chapter 3 for more information on optimization.

The if77 Driver 2

This chapter describes **if77**, the driver for compiling, assembling, and linking Fortran source code for execution on the Paragon™ system.

The following sections tell how to invoke **if77** and how to control its inputs, processing, and outputs.

Invoking the Driver

The **if77** driver is invoked by the following command line:

```
if77 [switches] source_file...
```

where:

switches Is zero or more of the command line switches listed in Table 2-1. Note that case is significant in switch names.

source_file Is the name of the file that you want to process. **if77** bases its processing on the suffixes of the files it is passed:

file.F is considered to be a Fortran program with preprocessor directives. It is preprocessed, compiled and assembled. The resulting object file is placed in the current directory.

The Fortran preprocessor is similar to the standard UNIX preprocessor **cpp**. See *The C Programming Language* by Kernighan and Ritchie for information on the preprocessor control directives used by this preprocessor.

file.f	is considered to be a Fortran program. It is compiled and assembled. The resulting object file is placed in the current directory.
file.s	is considered to be an i860 assembly language file. It is assembled and the resulting object file is placed in the current directory.
file.o	is considered to be an object file. It is passed directly to the linker if linking is requested.
file.a	is considered to be an **ar** library. It is passed directly to the linker if linking is requested.
file.c	is considered to be a C program. It is passed to the C compiler.

All other files are taken as object files and passed to the linker (if linking is requested) with a warning message. If a file's suffix does not match its actual contents, unexpected results may occur.

Table 2-1. Summary of if77 Driver Switches (1 of 2)

Switch	Description
-c	Skip link step; compile and assemble only (to *file*.o for each *file*.f).
-D*name*[=*def*]	Define preprocessor symbol *name* to be *def*.
-E	Preprocess each *file*.F to *stdout*.
-F	Preprocess each *file*.F to *file*.f.
-g	Synonymous with **-Mdebug -O0 -Mframe**.
-I*directory*	Add *directory* to include file search path.
-K*option*	Request special mathematical semantics (**ieee**, **ieee=enable**, **ieee=strict**, **noieee**, **trap=fp**, **trap=align**).
-l*library*	Load **lib***library***.a** from library search path (passed to the linker).
-L*directory*	Add *directory* to library search path (passed to the linker).
-m	Generate a link map (passed to the linker).

Table 2-1. Summary of if77 Driver Switches (2 of 2)

Switch	Description
-M*option*	Request special compiler actions (**alpha, anno, beta, [no]bounds, clr_reg, cray, concur, cncall, cpp860, [no]dclchk, [no]debug, [no]depchk, [no]dlines, dollar, extend, extract, [no]frame, [no]i4, info, inline, keepasm, [no]list, [no]longbranch, neginfo, noansi, nostartup, nostdinc, nostdlib, onetrip, [no]perfmon, [no]quad, [no]r8, [no]r8intrinsics, [no]recursive, [no]reentrant, reloc_libs, safealloc, [no]save, [no]signextend, [no]split_loop_ops, [no]split_loop_refs, standard, [no]streamall, [no]stride0, [no]unixlogical, [no]unroll, [no]upcase, vect, [no]vintr, [no]xp**).
-nostdinc	Remove the default include directory from the include files search path.
-nx	Create executable application for multiple nodes.
-o*file*	Use *file* as name of output file.
-O[*level*]	Set optimization *level* (**0, 1, 2, 3, 4**).
-r	Generate a relinkable object file (passed to the linker).
-S	Skip assemble and link step; compile only (to *file*.s for each *file*.f or *file*.F).
-s	Strip symbol table information (passed to the linker).
-U*name*	Remove initial definition of *name* in preprocessor.
-v	Print the entire command line for assembler, linker, etc. as each is invoked in verbose mode.
-V	Print the version banner for assembler, linker, etc. as each is invoked.
-VV	Displays the driver version number and the location of the online release notes, but performs no compilation.
-W*pass*,*option*[,*option*...]	Pass *option*s to *pass* (**0, a, l**).
-Y*pass*,*directory*	Look in *directory* for *pass* (**0, a, l, S, I, L, U, P**).

The rest of this chapter discusses these switches in more detail.

NOTE

The switches that discuss loop parallelization are available only with the Paragon System MP product.

Controlling the Driver

The following switches let you control how the driver processes its inputs:

-W	Pass specified options to specified tool.
-Y	Look in specified directory for specified tool.
-E	Skip compile, assemble, and link step; preprocess only (output to *stdout*).
-F	Skip compile, assemble, and link step; preprocess only (output to *file*.f).
-S	Skip assemble and link step; compile only (output to *file*.s).
-c	Skip link step; compile and assemble only (output to *file*.o).
-D	Define (create) preprocessor macro.
-U	Undefine (remove) preprocessor macro.

Specific Passes and Options

The following switch lets you pass options to specific passes (tools):

```
-Wpass,option[,option...]
```

where:

pass Is one of the following:

0 (zero)	Compiler.
a	Assembler.
l	Linker.

option Is a comma-delimited string that is passed as a separate argument.

The following switch lets you tell the driver where to look for a specific pass:

```
-Ypass,directory
```

where *pass* is one of the following:

0 (zero)	Search for the compiler executable in *directory*.
a	Search for the assembler executable in *directory*.
l	Search for the linker executable in *directory*.
S	Search for the start-up object files in *directory*.
I	Set the compiler's standard include directory to *directory*.
L	Set the first directory in the linker's library search path to *directory* (passes **-YL***directory* to the linker).
U	Set the second directory in the linker's library search path to *directory* (passes **-YU***directory* to the linker).
P	Set the linker's entire library search path to *directory* (passes **-YP***directory* to the linker).

See the **if77** manual page in Appendix D for the defaults for these directories; see the **ld860** manual page in Appendix D for more information on the **-YL**, **-YU**, and **-YP** switches.

Preprocess Only

By default, the driver preprocesses, compiles, assembles, and links each *file*.F. However, the following switches suppress the compile, assemble, and link steps:

-E	After preprocessing each *file*.F, send the result to standard output (*stdout*).
-F	After preprocessing each *file*.F, send the result to a file named *file*.f.

Note that these switches have meaning only for files with the uppercase ".F" suffix.

Preprocess and Compile Only

By default, the driver preprocesses, compiles, assembles, and links each *file*.F and compiles, assembles, and links each *file*.f. However, the following switch tells the driver to suppress the assemble and link steps and produce an assembler source file:

```
-S
```

After compiling each *file*.F or *file*.f, the assembler source file is sent to a file named *file*.s.

Preprocess, Compile, and Assemble Only

By default, the driver preprocesses, compiles, assembles, and links each *file*.F and compiles, assembles, and links each *file*.f. However, the following switch tells the driver to suppress the link step:

```
-c
```

After assembling each *file*.F or *file*.f, the output is sent to a file named *file*.o. If you are compiling a single source file, you can specify a different output file name with the **-o** switch.

Add and Remove Preprocessor Macros

The following command line switches let you predefine preprocessor macros and undefine predefined preprocessor macros:

-D*name*[=*def*] Define *name* to be *def* in the preprocessor. If *def* is missing, it is assumed to be empty. If the "=" sign is missing, then *name* is defined to be the string `1` (one).

-U*name* Remove any initial definition of *name* in the preprocessor.

Because all **-D** switches are processed before all **-U** switches, the **-U** switch overrides the **-D** switch. The **-U** switch affects only preprocessor macros defined with the **-D** switch, not macros defined in source files. The only macro predefined by the preprocessor itself is **__LINE__**, whose value is the current source file line number, and it cannot be undefined with **-U**.

Note that these switches have meaning only for files with the uppercase ".F" suffix.

Controlling the Compilation Step

The following switches let you control the compilation step:

-M*option* Request special compiler actions.

-I Add a directory to include file search path.

-O Set the optimization level.

-g Include symbolic debug information in the output file (synonymous with **-Mdebug -O0 -Mframe**).

Specific Actions

The following command line switch lets you request specific actions from the compiler:

```
-Moption
```

where *option* is one of the following (an unrecognized **-M** *option* is passed directly to the compiler, which often removes the need for the **-W0** switch):

alpha Activate alpha-release compiler features.

anno Produce annotated assembly files, where source code is intermixed with assembly language. **-Mkeepasm** or **-S** should be used as well.

beta Activate beta-release compiler features.

[no]bounds [Don't] enable array bounds checking (default **-Mnobounds**). With **-Mbounds** enabled, bounds checking is not applied to subscripted pointers or to externally-declared arrays whose dimensions are zero (**extern arr[ ]**). Bounds checking is not applied to an argument even if it is declared as an array. If an array bounds checking violation occurs when a program is executed, an error message describing where the error occurred is printed and the program terminates. The text of the error message includes the name of the array, where the error occurred (the source file and line number in the source), and the value, upper bound, and dimension of the out-of-bounds subscript. The name of the array is not included if the subscripting is applied to a pointer.

clr_reg Clear the internal registers after every procedure invocation. This option is used for diagnostic purposes.

concur=[*option*[,*option*...]]

Make loops parallel as defined by the specified *options*. *option* can be any of the following:

altcode:*count* Make innermost loops without reduction parallel only if their iteration count exceeds *count*. Without this switch, the compiler assumes a default *count* of 100.

altcode_reduction:*count* Make innermost loops with reduction parallel only if their iteration count exceeds *count*. Without this switch, the compiler assumes a default *count* of 200.

dist:block Make the outermost valid loop parallel. This is the default option.

dist:cyclic Make the outermost valid loop in any loop nest parallel. If an innermost loop is made parallel, its iterations are allocated to processors cyclically. That is, processor 0 performs iterations 0, 3, 6, ...; processor 1 performs iterations 1, 4, 7, ...; and processor 2 performs iterations 2, 5, 8, and so on.

global_vcache Directs the vectorizer to locate the cache within the area of an external array when generating codes for parallel loops. By default, the cache is located on the stack for parallel loops.

noassoc Do not make loops with reductions parallel. This is the same as **-Mvect=noassoc**.

cpp860 Direct the internal preprocessor to not compress white space.

cncall Make loops with calls parallel. By default, the compiler does not make loops with calls parallel since there is no way for the compiler to verify that the called routines are safe to execute in parallel. When you specify **-Mcncall** on the command line, the compiler also automatically specifies **-Mreentrant**.

-Mcncall also allows several other types of loops to be made parallel:

- loops with I/O statements
- loops with conditional statements
- loops with low loop counts
- non-vectorizable loops

If the compiler can detect a cross-iteration dependency in a loop, it will not make the loop parallel, even if **-Mcncall** is specified.

cray Enable Cray compatibility mode for various options.

[no]dclchk [Don't] require that all variables be declared (default **-Mnodclchk**).

[no]debug [Don't] generate symbolic debug information (default **-Mnodebug**). If -Mdebug is specified with an optimization level greater than zero, line numbers will not be generated for all program statements. **-Mdebug** increases the object file size.

[no]depchk [Don't] check for potential data dependencies (default **-Mdepchk**). **-Mnodepchk** is especially useful in disambiguating unknown data dependencies arising from use of array subscripts that cannot be derived at compile time. For example, if an array is referenced in a loop using the induction variable plus some other unknown non-induction-based variable as a subscript, the compiler must assume that the array conflicts with a similar array reference based on the induction variable alone. If it is known that the two array references do not conflict, then this switch may result in better code. Do not use this switch if such data dependencies do exist, because incorrect code may result.

[no]dlines [Don't] treat lines beginning with D in column 1 as executable statements, ignoring the D (default **-Mnodlines**).

dollar,*char* Specify *char* as the character to which the compiler maps the dollar sign. The compiler allows the dollar sign in names.

extend Allow 132-column source lines (normally only 72 columns are allowed).

extract=[*option*[,*option*...]]
Pass options to the subprogram extractor (see the **inline** option for more information). The *options* are:

[**name:**]*subprogram*
Extract the specified subprogram. **name:** must be used if the subprogram name contains a period.

[**size:**]*number* Extract subprograms containing less than approximately *number* statements.

If both *number*(s) and *subprogram*(s) are specified, then subprograms matching the given name(s) *or* meeting the size requirements are extracted.

The **-o***file* switch must be used with **-Mextract** to tell the compiler where to place the extracted subprograms. The name of the specified *file* must contain a period.

There are some restrictions on the types of subprograms that can be extracted. See Chapter 4 for these restrictions and other information on using the compiler's subprogram extractor.

[no]frame [Don't] include the frame pointer (default **-Mnoframe**). Using **-Mnoframe** can improve execution time and decrease code, but makes it impossible to get a call stack traceback when using a debugger.

[no]i4 [Don't] treat **integer** as **integer*4** (default **-Mi4**). **-Mnoi4** treats **integer** as **integer*2**.

info=[*option*[**,***option*...]]
Produce useful information on the standard error output. The *option*s are:

- **time** or **stat** Output compilation statistics.
- **loop** Output information about loops. This includes information about vectorization, software pipelining, and parallelization.
- **concur** Same as **-Minfo=loop**.
- **inline** Output information about subprograms extracted and inlined.
- **cycles** or **block** or **size** Output block size in cycles. Useful for comparing various optimization levels against each other. The cycle count produced is the compiler's static estimate of freeze-free cycles for the block.
- **ili** Output intermediate language as comments in assembly file.
- **all** All of the above.

inline=[*option*[**,***option*...]]
Pass options to the subprogram inliner. The *options* are:

- [**lib:**]*library* Inline subprograms in the specified inliner library (produced by **-Mextract**). If **lib:** is not used, the library name must contain a period. If no library is specified, subprograms are extracted from a temporary library created during an extract prepass.
- [**name:**]*subprogram* Inline the specified subprogram. If **name:** is not used, the subprogram name must not contain a period.
- [**size:**]*number* Inline subprograms containing less than approximately *number* statements.
- **levels:***number* Perform *number* levels of inlining (default 1).

If both *number*(s) and *subprogram*(s) are specified, then subprograms matching the given name(s) *or* meeting the size requirements are inlined.

There are some restrictions on the types of subprograms that can be inlined. See Chapter 4 for these restrictions and other information on using the compiler's subprogram extractor.

iomutex Place critical sections around I/O statements.

keepasm Keep the assembly file for each Fortran source file, but continue to assemble and link the program. This is mainly used in compiler performance analysis and debugging.

list[=*name*] Create a source listing in the file *name*. If *name* is not specified, the listing file has the same name as the source file except that the ".f" suffix is replaced by a ".lst" suffix. If *name* is specified, the listing file has that name; no extension is appended.

nolist Don't create a listing file (this is the default).

[no]longbranch [Don't] allow compiler to generate **bte** and **btne** instructions (default **-Mlongbranch**). **-Mnolongbranch** should be used only if an assembly error occurs.

neginfo=concur

Print information for each countable loop that is not made parallel stating why the loop was not made parallel.

noansi Allow multiple implicit statements.

nostartup Don't link the usual start-up routines (*crt0.o* and *ifmain.o*), which contain the entry point for the program.

nostdinc Remove the default include directory (*/usr/include* for **f77**, *$(PARAGON_XDEV)/paragon/include* for **if77**) from the include files search path (the list of directories searched for files referenced by **include** statements, such as *fnx.h*).

nostdlib Don't link the standard libraries (*libpm.o*, *guard.o*, *libf.a*, *libm.a*, *libc.a*, *iclib.a*, and *libmach3.a*) when linking a program.

onetrip Force each **do** loop to be iterated at least once (for compatibility with Fortran 66).

[no]perfmon [Don't] link the performance monitoring module (*libpm.o*) (default **-Mperfmon**). See the *Paragon™ System Application Tools User's Guide* for information on performance monitoring.

[no]quad [Don't] force top-level objects (such as local arrays) of size greater than or equal to 16 bytes to be quad-aligned (default **-Mquad**). Note that **-Mquad** does not affect items within a top-level object; such items are quad-aligned only if appropriate padding is inserted. Common blocks are always quad-aligned.

[no]r8 [Don't] treat **real** as **double precision** and **real** constants as **double precision** constants (default **-Mnor8**).

[no]r8intrinsics [Don't] treat intrinsics as follows (default **-Mnor8intrinsics**):

cmplx as **dcmplx**
real as **dble**
alog as **dlog**
alog10 as **dlog10**
amax1 as **dmax1**
amin1 as **dmin1**
amod as **dmod**
csqrt as **cdsqrt**
clog as **cdlog**
cexp as **cdexp**
csin as **cdsin**
ccos as **cdcos**

[no]recursive [Don't] allocate local variables on the stack, thus allowing recursion (default **-Mnorecursive**). **SAVE**d, data-initialized, or namelist members are always allocated statically, regardless of the setting of this switch.

[no]reentrant [Don't] generate reentrant code (default **-Mnoreentrant**). **-Mreentrant** disables certain optimizations that can improve performance but may result in code that is not reentrant. Even with **-Mreentrant**, the code may still not be reentrant if it is improperly written (for example, if it declares static variables).

reloc_libs Causes **-l** switches that appear before source or object file names on the compiler command line to appear after these file names on the **ld** command line.

[no]save [Don't] allocate all local data in static locations instead of on the stack (default **-Msave**). The effect is similar to using the **save** statement for all local variables. Recursion is not allowed with this switch in effect. **-Msave** may allow some older Fortran programs to run, but may decrease performance.

[no]signextend [Don't] sign-extend the result of a conversion of a signed integer to a smaller signed type (default **-Mnosignextend**). For example, if **-Msignextend** is in effect and an **integer*4** containing the value 65535 is converted to an **integer*2**, the value of the **integer*2** will be -1. This option is provided for compatibility with other compilers. **-Msignextend** will decrease performance.

split_loop_ops=*n*
Set a threshold of *n* floating-point operations within a loop. Innermost loops whose number of floating-point operations exceeds *n* are split. Each floating-point operation counts as two. The default for *n* is 40 when **-Mvect** is used.

nosplit_loop_ops
Do not split loops when the floating-point operation threshold is exceeded. When **-Mvect** is specified, innermost loops whose number of floating point operations exceed 40 are split by default. This switch turns the default off.

split_loop_refs=*n*
Set a threshold of *n* array element loads and stores within a loop. Innermost loops whose number of loads and stores exceeds *n* are split. The default for *n* is 20 when **-Mvect** is used

nosplit_loop_refs
Do not split loops when the array element loads and stores threshold is exceeded. When **-Mvect** is specified, innermost loops whose number of array element loads and stores exceeds 20 are split by default. This switch turns the default off.

standard Flag non-ANSI-Fortran77 usage.

[no]streamall [Don't] stream all vectors to and from cache in a vector loop (default **-Mstreamall**). When **-Mnostreamall** is in effect, the compiler chooses one vector to come directly from or go directly to main memory, without being streamed into or out of cache.

[no]stride0 [Don't] inhibit certain optimizations and allow for stride 0 array references. **-Mstride0** may degrade performance, and should only be used if zero stride induction variables are possible. (default **-Mnostride0**).

unixlogical Set the value of a logical expression to one if the result is **.TRUE.**.

unroll[=*option* [,*option* ...]]
Invoke the loop unroller and set the optimization level to 2 if it is set to less than 2. *option* is one of the following:

c:*m* Completely unroll loops with a constant loop count less than or equal to *m*. If *m* is not supplied, the default value is 4.

n:*u* Unroll loops that are not completely unrolled or have a non-constant loop count *u* times. If *u* is not supplied, the unroller computes the number of times a loop is unrolled.

nounroll Do not unroll loops.

[no]upcase [Don't] preserve case in names (default **-Mnoupcase**). **-Mnoupcase** causes all names to be converted to lower case. Note that, if **-Mupcase** is used, then variable name *Q* is different than variable name *q*, and keywords must be in lower case.

vect[=*option*[,*option*...]]
Perform vectorization (also enables **-Mvintr**). If no *option*s are specified, then all vector optimizations are enabled. Note that **-Mvect** causes **-O0**, **-O1** optimization levels to be prevented; **-O2** is the default while **-O3** and **-O4** are supported. The available *option*s are:

altcode[:*number*]
Produce non-vectorized code to be executed if the loop count is less than or equal to *number*. Otherwise execute vectorized code. The default value for *number* is 10.

noaltcode Generate no non-vectorized alternate code.

cachesize:*number*
This sets the size of the portion of the cache used by the vectorizer to *number* bytes. *Number* must be a multiple of 16, and less than the cache size of the microprocessor (16384 for the i860 XP, 8192 for the i860 XR). In most cases the best results occur when *number* is set to 4096, which is the default (for both microprocessors).

noassoc When scalar reductions are present (for example, dot product), and loop unrolling is turned on, the compiler may change the order of operations so that it can

generate better code. This transformation can change the result of the computation due to round-off error. The use of **noassoc** prevents this transformation.

recog Recognize certain loops as simple vector loops and call a special routine.

smallvect[*:number*]
This option allows the vectorizer to assume that the maximum vector length is no greater than *number*. *number* must be a multiple of 10. If *number* is not specified, the value 100 is used. This option allows the vectorizer to avoid stripmining in cases where it cannot determine the maximum vector length. In doubly-nested, non-perfectly nested loops this option can allow invariant vector motion that would not otherwise have been possible. Incorrect code may result if this option is used, if a vector takes on a length greater than specified.

streamlim:*n* This sets a limit for application of the vectorizer data streaming optimization. If data streaming requires cache vectors of length less than *n*, the optimization is not performed. Other vectorizer optimizations are still performed. The data streaming optimization has a high overhead compared to other loop optimizations, and can be counter-productive when used for short vectors. The *n* specifier is not optional. The default limit is 32 elements if **streamlim** is not used.

transform Perform high-level transformations such as loop splitting and loop interchanging. This is normally not useful without **-Mvect=recog**.

-Mvect with no options means the following:
-Mvect=recog,transform,cachesize:4096,altcode:10.

[no]vintr [Don't] perform recognition of vector intrinsics (default **-Mnovintr**, unless **-Mvect** is used).

[no]xp [Don't] use i860 XP microprocessor features (default **-Mxp**). See the *i860™ 64-Bit Microprocessor Family Programmer's Reference Manual* for information on the differences between the i860 XP microprocessor and the original i860 XR microprocessor.

Location of Include Files

The following command line switch lets you add a specified directory to the compiler's search path for include files:

```
-Idirectory
```

where *directory* is the pathname of the directory to be added. If you use more than one **-I** switch, the specified directories are searched in the order they were specified (left to right).

The **INCLUDE** statement directs the compiler to begin reading from another file. The compiler uses two rules to locate the specified file. Note that the Fortran **INCLUDE** statement is different from the **#include** statement, whcih uses the C preprocessor.

1. If the filename specified in the **INCLUDE** statement includes a pathname, the compiler begins reading from the file it specifies.

2. If no pathname is provided in the **INCLUDE** statement, the compiler searches for the file in the following order:

   - any directories specified with **-I**
   - the directory containing the source file
   - the current directory

Optimization Level

The following command line switch lets you set the optimization level explicitly:

```
-O[level]
```

where *level* is one of the following:

0	A basic block is generated for each Fortran statement. No scheduling is done between statements. No global optimizations are performed.
1	Scheduling within extended basic blocks is performed. Some register allocation is performed. No global optimizations are performed.
2	All level 1 optimizations are performed. In addition, traditional scalar optimizations such as induction recognition and loop invariant motion are performed by the global optimizer.

3	All level 2 optimizations are performed. In addition, software pipelining is performed.
4	All level 3 optimizations are performed, but with more aggressive register allocation for software pipelined loops. In addition, code for pipelined loops is scheduled several ways, with the best way selected for the assembly file.

If **-O** is used without a *level*, the optimization level is set to 2. If you do not use the **-O** switch, the default optimization level is 1.

NOTE

When compiling an application for debugging, you will get the best results using **-O0**.

If you prefer optimized code to "debuggability," use **-O2**. See Chapter 3 for information on additional compiler optimization features.

Generating Debug Information

To compile for debugging you should use the **-g** compiler switch. The **-g** switch is equivalent to **-Mdebug -Mframe -O0**. These switches have the following effects:

-Mdebug	Generate symbol and line number information.
-Mframe	Generate stack frames on function calls. (Default **-Mnoframe**.) Debugging code without stack frames generated on function calls will result in stack tracebacks that have missing calls when you use the **frame** command.
-O0	Optimization off. If you do not turn optimization off, access to individual source lines will be decreased, and display or modification of variables and registers will probably have unpredictable results.

You can debug programs not compiled for debugging, but your ability to debug will be very limited. The debugging information generated by **-g** increases the object file size.

Note that **-Mvect** causes the compiler to ignore optimization levels less than 2. For example, **-g -Mvect** is the same as **-g -Mvect -O2**. Optimization cannot be turned off when+ **-Mvect** is used.

Controlling the Link Step

The following switches let you control the link step (they are all passed directly to the linker):

-s	Strip symbol table information.
-r	Generate a relinkable object file.
-m	Produce a link map.
-L	Change the default library search path.
-l	Load a specific library.

Stripping Symbols

The following command line switch strips all symbols from the output object file:

```
-s
```

This results in a smaller object file, but makes it more difficult to debug.

Generating a Relinkable Object File

The following command line switch generates a relinkable object file:

```
-r
```

When you use the **-r** switch, the linker keeps internal symbol information in the resulting object file. This lets you link the object file together with other object files later.

Producing a Link Map

The following command line switch produces a link map on the standard output:

```
-m
```

The link map lists the start address of each section in the object file. To get more information about the object file, use the **dump860** command.

Linker Libraries

The following switch adds a directory to the head of the linker's library search path:

```
-Ldirectory
```

where *directory* is the pathname of a directory that the linker searches for libraries. The linker searches *directory* first (before the default path and before any previously specified **-L** paths).

The following switch tells the linker to use a specific linker library:

```
-llibrary
```

The linker loads the library **lib***library*.**a** from the first library directory in the library search path in which a file of that name is encountered.

See the **ld860** manual page in Appendix D for more information on the linker's library search path.

Controlling Mathematical Semantics

The following command line switch lets you request special mathematical semantics from the compiler and linker:

```
-Koption
```

where *option* is one of the following:

ieee If used while linking, links in a math library that conforms with the IEEE 754 standard.

If used while compiling, tells the compiler to perform **real** and **double precision** divides in conformance with the IEEE 754 standard.

ieee=enable If used while linking, has the same effects as **-Kieee**, and also enables floating point traps and underflow traps. If used while compiling, has the same effects as **-Kieee**.

ieee=strict If used while linking, has the same effects as **-Kieee=enable**, and also enables inexact traps. If used while compiling, has the same effects as **-Kieee**.

noieee If used while linking, produces a program that flushes denormals to 0 on creation (which reduces underflow traps) and links in a math library that is not as accurate as the standard library, but offers greater performance. This library offers little or no support for exceptional data types such as **INF** and **NaN**, and will trap on such values when encountered.

If used while compiling, tells the compiler to perform **real** and **double precision** divides using an inline divide algorithm that offers greater performance than the standard algorithm. This algorithm produces results that differ from the results specified by the IEEE standard by no more than three units in the last place.

trap=fp If used while compiling, disables kernel handling of floating point traps. Has no effect if used while linking.

trap=align If used while compiling, disables kernel handling of alignment traps. Has no effect if used while linking.

-Kieee is the default. See "Non-IEEE Math (-Knoieee)" on page 3-12 for more information on the **-K** switch.

Controlling the Driver Output

The following switches let you control the driver's outputs:

-nx Create an executable application for multiple nodes.

-o Specify the name of the output file.

-V Print the version banner for each tool (assembler, linker, etc.) as it is invoked.

-VV Display the driver version number and the location of the online release notes. but do not perform any compilation.

-v Print the entire command line for each tool as it is invoked, and invoke each tool in verbose mode (if it has one).

Executable for Multiple Nodes

By default, the **if77** driver creates an executable for a single node. The following command line switch creates an executable for multiple nodes:

```
-nx
```

The **-nx** switch has no effect if used while compiling. If used while linking, it has two effects:

- It links in *libnx.a*, the library that contains all the calls in the *Paragon™ System Fortran Calls Reference Manual*. It also links in *libmach.a* and *options/autoinit.o*.

- It links in a special start-up routine that automatically copies the program onto multiple nodes, as specified by standard command line switches and environment variables. See the *Paragon™ System User's Guide* for information on these command line switches and environment variables.

For compatibility with the iPSC® system, the **if77** driver currently accepts the following command line switch, which is synonymous with **-nx**:

```
-node
```

However, support for this switch may be dropped in a future release.

Name of Executable File

By default, the executable file is named *a.out* (or *file*.o if you use the **-c** switch). However, the following command line switch lets you name the file anything you like:

```
-ofile
```

where *file* is the desired name.

Verbose Mode

By default, the driver does its work silently. However, the following command line switch causes the driver to display the version banner of each tool (assembler, linker, etc.) as it is invoked:

```
-V
```

The following command line switch causes the driver to identify itself in more detail than the **-V** switch and display the location of the online compiler release notes. No compilation is performed:

```
-VV
```

The following command line switch causes the driver to display the entire command line that invokes each tool, and to turn on verbose mode (if available) for each tool:

```
-v
```

Overriding Compiler Defaults

You can override the default switch settings for the Paragon Fortran compiler by creating a compiler default file in your home directory, your current working directory, or the directory where the compiler driver resides. This file must be named *.icfrc*. The default file contains compiler switches as they would appear on the command line, delimited by spaces, tabs, or new lines. The file can contain any number of lines.

The following is an example of the contents of a default file:

```
-O3 -Mvect
-Knoieee -Mframe -Mnoperfmon
```

The compiler searches the following directories in the order listed for the *.icfrc* file.

1. your current working directory
2. your home directory
3. The directory where the compiler driver resides. If you place a *.icfrc* file in *usr/ccs/bin* on a Paragon system, you should also have the system administrator create a link to that directory in *usr/bin*.

If you have default files in more than one of these directories, the compiler uses the first one found.

NOTE

The *.icfrc* file is used by both the Paragon C compiler and the Paragon Fortran compiler. It is suggested that *.icfrc* files that reside in your home directory or the directory where the compiler driver resides contain only switches that are common to both compilers.

When you invoke the compiler, the compiler driver reads the default file, if it exists, and constructs a new command line. The command line consists of the switches in the *.icfrc* file first, then the switches in the command line you used to invoke the compiler. Because of this order, you should not put arguments in the default file if they must go at the end of the command line. An example would be directives to link to libraries. The following is the order of precedence for compiler switches:

1. specific entries on the command line
2. entries in the .icfrc file
3. default switch settings

For example, suppose you have the following entries in your *.icfrc* file:

```
-O3 -Mvect
```

If you use the following command line to invoke the compiler:

```
icc -O4 example.c
```

The compiler will generate the following command line:

```
icc -O3 -Mvect -O4 example.c
```

Because the **-O4** switch from the compiler invocation comes after the **-O3** switch from the default file, the explicit command line switch overrides the default file switch, and the optimization level is set to 4.

NOTE

Although you can include file names and switches such as **-c** in the default file, this is not advisable because all arguments in the default file will appear on all compiler command lines. Arguments other than those needed to override default settings of switches should go in a make file.

Control Directives

Control directives alter the effects of certain command line switches or the default behavior of the compiler. While a command line switch effects the entire source file being compiled, control directives affect only selected subprograms or loops in the source file. Control directives allow you to fine tune selected routines or loops.

Directives have the following syntax:

cdir$[*scope*] *directive_body*

The **c** in the directive syntax must be in column 1. For compatibility with other compilers, you can substitute **cvd** for **cdir** in a directive.

scope can be **l** (loop), **r** (routine), or **g** (global)

For directives that allow **loop**, **routine**, and **global** scope, the following rules apply:

l(loop) Indicates the directive applies to the next lexical loop. The directive does not apply to any loops that are enclosed by the next loop. Loop-scoped directives are only applied to **DO** loops.

r(routine) Indicates the directive applies to the code that follows the directive until the end of the routine.

g(global) Indicates the directive applies to the code that follows the directive until the end of the file.

For directives where **loop** scope is not allowed, the scope rules fall into two groups.

The following rules apply to directives **func32**, **frame**, and **opt**:

r(routine) Indicates the directive applies to the current routine, if it is in a routine. If it is not in a routine, it applies to the next routine.

g(global) Indicates the directive applies to all routines that follow it.

The following rules apply to directive **bounds**:

r(routine) Indicates the directive applies to the code that follows the directive until the end of the routine.

g(global) Indicates the directive applies to the code that follows the directive until the end of the file.

If *scope* is not specified, the default scope for each individual directive is applied. Table 1-1 lists these defaults. Additional scope rules are described in the following section.

directive_body can include any of the directives listed in Table 2-2.

The body of the directive can immediately follow scope, or any number of blanks can separate *scope* from the body of the directive. Case is not significant in a directive name, so the names can include upper or lowercase characters. Case is significant for any variable names that appear in the body of the directive if the **-Mupcase** switch has been specified on the command line.

Table 2-2 provides a summary of the supported directives. The default column specifies the default condition for each directive. The scope column lists the permitted scopes for each directive, with the default scope in parentheses. The name of a directive can be preceded by a **-M**. For example, **-Mnoassoc** is equivalent to **noassoc**.

Table 2-2. Directive Summary (1 of 2)

DIRECTIVE	DESCRIPTION	DEFAULT	SCOPE
altcode[*n*]concur	Execute inner loops without reductions in parallel only if their iteration count exceeds *n*.	*n*=100	(l)rg
altcode[*n*]concurreduction	Execute inner loops with reductions in parallel only if their iteration count exceeds *n*.	*n*=200	(l)rg
[no]assoc	[Don't] perform associative transformations	**assoc**	(l)rg
[no]bounds	[Don't] perform array bounds checking	**nobounds**	(r)g
[no]concur	[Don't] consider loops for parallelization	**noconcur**	(l)rg
[no]cncall	[Don't] consider loops for parallelization even if they contain calls or conditionals, their loop counts do not exceed thresholds, or they contain inner non-vectorizable loops	**nocncall**	(l)rg
dist=block	Change concurrency characteristics to block	**N/A**	(l)rg
dist=cyclic	Change concurrency characteristics to cyclic	**N/A**	(l)rg
[no]depchk	[Don't] check for potential data dependencies	**depchk**	(l)rg
[no]eqvchk	[Don't] check EQUIVALENCE statements for data dependencies	**eqvchk**	(l)rg
[no]func32	[Don't] align functions on 32-byte boundaries	**nofunc32**	(r)g
ivdep	Ignore potential data dependencies	**depchk**	(l)rg
[no]lstval	[Don't] compute last values	**lstval**	(l)rg

Table 2-2. Directive Summary (2 of 2)

DIRECTIVE	DESCRIPTION	DEFAULT	SCOPE
opt	Select optimization level	**N/A**	(r)g
[**no**]**recog**	[Don't] recognize vector idioms	**recog**	(l)rg
[**no**]**smallvect**	[Don't] assume short loop count	**nosmallvect**	(l)rg
[**no**]**shortloop**	[Don't] assume short loop count	**noshortloop**	(l)rg
[**no**]**swpipe**	[Don't] perform software pipelining transformations	**swpipe**	(l)rg
[**no**]**transform**	[Don't] perform vector transformations	**transform**	(l)rg
[**no**]**vector**	[Don't] perform vectorizations	**vector**	(l)rg
[**no**]**vintr**	[Don't] recognize vector intrinsics	**vintr**	(l)rg

NOTE

The Cray directive **cdir$** [**no**]**vector** has routine scope instead of loop. The default scope for [**no**]**vector** when any other prefix is used, such as **cvd$**, is loop.

Directive Descriptions

The following sections provide descriptions of each control directive.

altcode[*n*]concur

This directive alters the effect of the **-Mconcur=altcode:n** command line switch. The directive makes innermost loops without reduction parallel only if their iteration count exceeds *n*. Without this directive, the compiler assumes a default of 100.

altcode[*n*]concurreduction

This directive alters the effect of the **-Mconcur=altcode_reduction:n** command line switch. The directive makes innermost loops with reduction parallel only if their iteration count exceeds *n*. Without this directive, the compiler assumes a default of 200.

[no]assoc

This directive alters the effects of the **-Mvect=noassoc** or **-Mconcur=noassoc** command line switches. By default, when scalar reductions are present the vectorizer may change the order of operations to generate better code and allow parallelization of loops. Such transformations change the result of the computation due to roundoff error. The **noassoc** directive disables these transformations.

[no]bounds

This directive alters the effects of the **-Mbounds** command line switch. The **bounds** directive enables the checking of array bounds when subscripted array references are performed. By default, array bounds checking is not performed.

[no]cncall

This directive alters the effects of the **-Mcncall** command line switch. The **cncall** directive causes the compiler to consider loops within the specified scope for parallelization, even if they contain calls to user-defined routines, they contain conditional statements, their loop counts do not exceed the usual thresholds, or they contain inner non-vectorizable loops. If you use the **cncall** directive, you must specify **-Mconcur** on the compiler command line.

[no]concur

This directive alters the effects of the **-Mconcur** command line switch. The **concur** directive causes the compiler to consider loops within the specified scope for parallelization. If you use the **concur** directive, you must specify **-Mconcur** on the compiler command line.

[no]depchk

This directive alters the effects of the **-Mdepchk** command line switch. When potential data dependencies exist, the compiler, by default, assumes that a data dependency exists which may inhibit certain optimizations or vectorizations. The **nodepchk** directive directs the compiler to ignore these potential data dependencies.

[no]eqvchk

The **noeqvchk** directive causes the compiler to ignore any dependencies between variables appearing in **EQUIVALENCE** statements. By default, the compiler checks for dependencies.

[no]func32

This directive alters the effects of the **-Mfunc32** command line switch. The **func32** directive causes the compiler to align functions on a 32-byte boundary. By default, functions are aligned on an 8-byte boundary.

ivdep

The **ivdep** directive is equivalent to the **nodepchk** directive.

[no]lstval

The compiler determines whether or not the last values for loop iteration control variables and promoted scalars must be computed. When the compiler determines it is necessary, it computes the last values. The **nolstval** directive causes the compiler to not compute last values.

There is no command line switch that corresponds to this directive.

opt

This directive overrides the value specified by the **-O** command line switch. The syntax for the **opt** directive is as follows:

cdir$[<*scope*>] **opt=**<*level*>

scope can be either **r** or **g**, and *level* is an integer constant representing the optimization level desired for the subprogram (routine scope) or all subprograms in a file (global scope).

[no]recog

This directive alters the effects of the **-Mvect** command line switch. If the **-Mvect=transform** switch is included on the command line, vector recognition is disabled for the entire compilation. The **norecog** directive allows selective disabling of vector recognition when the **-Mvect** switch is selected. The **recog** directive toggles a previous **norecog**.

The **recog** directive only affects the compiler when **-Mvect** is included on the command line.

[no]smallvect

This directive alters the effects of the **-Mvect=smallvect** command line switch. The **smallvect** directive has the following syntax:

cdir$[*scope*] **smallvect**[=*count*]

scope can be **g**, **l**, or **r**. *count* is an integer constant that specifies the maximum iteration count for a loop whose count is not a constant. If *count* is not specified, the default value is 100.

The default condition is **nosmallvect**, where the vectorizer does not make assumptions about the maximum iteration count for loops whose counts are not constants.

[no]shortloop

This directive is identical to the **[no]smallvect** directive.

[no]swpipe

The **noswpipe** directive causes the compiler to suppress software pipelining transformations that normally occur at optimization levels greater than 2.

There is no command line switch that corresponds to this directive.

[no]transform

This directive alters the effects of the **-Mvect=transform** command line switch. The **notransform** directive can be used to inhibit vector transformations when the **-Mvect** switch is in effect. The **transform** directive can be used to toggle a previous **notransform**. The **transform** directive only affects compilation when the **-Mvect** switch is specified on the command line.

[no]vector

The **novector** directive disables vector transformations and vector recognitions. This directive only affects compilation when the **-Mvect** switch is specified on the command line.

[no]vintr

The **novintr** directive disables recognition of vector intrinsics. This directive only affects compilation when the **-Mvect** switch is specified on the command line. If both the **norecog** and **vintr** directives are present, the **norecog** directive takes precedence.

Directive Examples

This section presents several examples that illustrate the effects of directives and the use of the scope specifiers. During compilation, a directive either turns a switch on or off, and the directive only applies to the section of code following the directive and defined by the scope specified. The scope can be the following loop, the current or following routine, or the rest of the program.

The following program is used for the first example:

```
subroutine s1(a,b,x,y,n)
double precision a(n),b(n), x(n,n), y(n,n)
do i=1,n
   a(i) = sin(b(i))
   do j = 1, n
      x(j,i) = cos(y(j,i))
   enddo
enddo
end
```

When this subroutine is compiled using the **-Mvect** command line switch as follows, the sine and cosine functions are both recognized as operations on vectors, and the compiler produces code using the vector versions of the sine and cosine routines:

```
if77 -Mvect -c -ovect.o subs1.f
```

You can use directives in the source code to alter the compiler behavior as follows:

```
        subroutine s1(a,b,x,y,n)
        double precision a(n), b(n), x(n,n), y(n,n)
cdir$1 novintr
```

```
            do i=1,n
               a(i) = sin(b(i))
cdir$l vintr
               do j = 1, n
                  x(j,i) = cos(y(j,i))
               enddo
            enddo
            end
```

In this version of the program, the compiler does not use the vector intrinsic sine routine, since the first directive turns off vector intrinsic recognition for the loop containing the sine. The second directive toggles the vintr switch before the nested loop, so the compiler uses a vector intrinsic routine for the cosine.

The following example uses the r (routine) directive scope:

```
cdir$r novintr
            subroutine s2(a,b,x,y,n)
            double precision a(n), b(n), c(n), d(n)
cdir$l vintr
            do i=1,n
               a(i) = sin(b(i))
            enddo
            do j = 1, n
               c(j) = cos(d(j))
            enddo
            end
```

When subroutine s2 is compiled using the -Mvect command line switch, the sine intrinsic is recognized as an operation on a vector and the compiler produces code using the vector intrinsic sine routine:

```
if77 -Mvect -c -ovect.o subs2.f
```

Since the scope of the **novintr** directive is for the routine, vector recognition is disabled for the subroutine s2. However, the loop-scoped directive **vintr** appears before the do loop containing the reference to **sin( )**, so vector intrinsic recognition is enabled only for that loop. Since the loop containing the reference to **cos( )** does not have a loop-scoped **vintr** directive in effect, the vector version of **cos( )** is not recognized.

In the following example, the global **novintr** directive turns off vector intrinsic recognition for the entire file:

```
cdir$g novintr
        subroutine s3(a,b,x,y,n)
        double precision a(n), b(n), x(n,n), y(n,n)
        do i=1,n
           a(i) = sin(b(i))
           do j = 1, n
              x(j,i) = cos(y(j,i))
           enddo
        enddo
        end
```

Optimizing Programs 3

Introduction

This chapter gives you a strategy for using the compiler's optimization features to help maximize the single-node performance of your programs. It also explains what the most commonly-used compiler optimization switches do and how they interact with each other. Finally, it gives you a few tips for changes you can make in your code to help the program run faster.

The techniques discussed in this chapter are *single-node optimizations only.* They make the program run faster on each node, but do not improve the program's internode communications. See the *Paragon™ System User's Guide* for information on improving the performance of a multi-node application.

Optimization Procedure

This section presents the recommended procedure for optimizing a new or ported program. The fundamental characteristics of this procedure are *adding optimizations in a controlled manner* and *testing the program after each optimization.*

1. Compile your program with the **-O2** switch for scalar optimizations. The optimizations performed at level 2 are considered "safe"—if your program works at all, it should continue to work (and work faster) with **-O2.**

2. Test the program to be sure it works as you expect.

3. When the program is working, use the performance analysis tools to determine which parts of the code are taking the most time. (See the *Paragon™ System Application Tools User's Guide* for information on performance analysis.)

4. Inspect the time-consuming code to see if will benefit from vectorization. In general, vectorization helps floating-point math on large vectors or in loops. It does not help integer math, string operations, or file operations.

5. Recompile *only those files that will benefit from vectorization* with the **-O4** and **-Mvect** switches.

6. Test the vectorized program to be sure it is still working and has not slowed down. (If the program gives unexpected results or runs more slowly than it did before, try recompiling the vectorized files with **-O3 -Mvect** instead; if loop counts are small, try **-O4** without **-Mvect** instead.)

7. Examine your program to see if it is "numerically stable." A program is said to be numerically stable if it does not depend on the behavior specified by the IEEE standard for floating-point mathematics, such as proper behavior in case a denormal, infinity, or "not-a-number" occurs during a calculation. Recompile and/or link *only those files that are numerically stable* with the **-Knoieee** switch. (The differences between using **-Knoieee** when compiling and using **-Knoieee** when linking are described later in this chapter.) You may get different results with **-Knoieee** on compile and link, and on different source files; try a variety of combinations.

8. If you have MP nodes, compile with **-Mconcur -O4 -Mvect**. Programs with large loop counts can often run faster on two CPUs.

9. Test the program after each attempt to be sure it is still working and has not slowed down.

Further optimizations may be possible at this point. Depending on the program, you may be able to use additional compiler optimization switches (as described under "Compiler Switches for Optimization" on page 3-3) and/or modify your code for greater performance (as described under "Code Changes for Optimization" on page 3-15). Be sure to test the program after each change.

Shortening Turnaround Time

As you can see, optimizing a program can involve many "compile, link, run" cycles. You may be able to reduce the time consumed by each run by using one or more of the following techniques:

- Use a smaller input file.
- Temporarily reduce the count in the outermost loop of the program.
- Add a call to **exit()** after a key subroutine.
- Extract key subroutines into a separate program for testing.

These techniques can help you to optimize your program more quickly by performing more tests per unit time. However, when you use these techniques, be sure that the reduced data or program fragment is representative of the whole program.

Compiler Switches for Optimization

The **if77** command has a number of switches you can use to request compiler optimizations:

-O	Performs general code optimizations.
-Mvect	Performs vectorization.
-Mconcur	Performs loop parallelization.
-Mcncall	Parallelizes loops with calls.
-Munroll	Unrolls loops.
-Knoieee	Uses faster but less accurate floating-point math.
-lkmath	Links to an optimized BLAS library.
-Minline	Replaces subprogram calls with inline code.
-Mnodepchk	Ignores potential data dependencies.
-Mstreamall	Instructs the compiler to stream all vector stores in a loop to the processor cache.Best used with **-Mvect**.

These switches are discussed in the remainder of this section.

General Optimizations (-O)

The **-O** switch performs general code optimization. The **-O** can be followed by a number that specifies the optimization level, from 0 (no optimization) to 4 (all optimizations). Each optimization level performs all the optimizations that the levels below it perform.

If you don't use the **-O** switch, you get optimization level 1. If you use **-O** with no number following it, you get optimization level 2.

Programs optimized at levels above 0 cannot be debugged easily with a symbolic debugger. If you are compiling an application for debugging, you should use the **-O0** switch.

Scalar Optimizations (-O1, -O2)

Optimization levels 1 and 2 perform scalar optimizations. These optimizations do not use the special features of the i860™ microprocessor, but they can improve the performance of most code and are unlikely to break working code.

- Level 1 performs only *local optimizations*: those that affect only a single Fortran statement. These optimizations include algebraic identity removal (removal of subexpressions that do nothing, such as **a=a**), and redundant load and store elimination (elimination of unnecessary memory accesses).

- Level 2 performs *global optimizations*: those that can affect multiple Fortran statements. These optimizations include invariant code motion (moving code that is the same on each iteration of a loop out of the loop) and global register allocation (assigning variables to registers based on how and when they are used).

Software Pipelining (-O3, -O4)

Optimization levels 3 and 4 make the compiled program use the i860 microprocessor's pipelining and dual-instruction mode features. These optimizations are beneficial only for code that performs intensive floating-point mathematics, particularly in loops. Since this type of code is also usually vectorizable, the **-O3** and **-O4** switches are usually used together with **-Mvect**.

Pipelining and dual-instruction mode allow the i860 microprocessor to work on more than one operation at a time.

- *Pipelining* means that the i860 microprocessor's floating-point unit can accept new input while previous inputs continue to move toward the result. For example, a floating-point addition takes three clock cycles, but the adder can accept new input every clock cycle. (The results of each input emerge from the adder three clock cycles after the operands entered.)

  Pipelining means that a sequence of similar operations can be performed in less time. However, it takes a few cycles to prime the pipeline and a few cycles to drain it; this means that a pipeline must have a certain minimum number of operations to be efficient.

  The exposed pipeline of the i860 microprocessor allows floating-point adds and multiplies to occur simultaneously (this is called *dual-operation mode*).

- *Dual-instruction mode* means that the i860 microprocessor's floating-point unit and integer unit can be active at the same time. For example, the floating-point adder can perform an addition at the same time the integer unit is loading the operands for the next addition.

Optimization levels 3 and 4 both attempt to schedule the program's operations to make the most use of pipelining and dual-instruction mode. This procedure is called *software pipelining*. For example, if the program contains an addition and a multiplication that are near each other but do not depend on the other's results, the compiler can schedule the two operations to occur at the same time.

- Level 3 uses a single scheduling algorithm on all candidates for software pipelining.

- Level 4 considers several scheduling algorithms for each candidate, and chooses the one that gives the best performance (or none of them, if the non-pipelined code is faster).

In theory, the code produced by level 4 should always be faster than the code produced by level 3, at the cost of a very small increase in compilation time. You should try **-O4** first, then try **-O3** if the results are not satisfactory.

Keep in mind that optimization levels 3 and 4 benefit code that is floating-point intensive. Code that spends most of its time in string handling, disk operations, or other non-floating-point operations will generally not benefit from optimization levels greater than 2.

Vectorization (-Mvect)

The **-Mvect** switch performs vectorization. Vectorization consists of three processes, which are described in the next section. Vectorization is beneficial only for code that performs floating-point calculations on long vectors, typically in loops of 10 or more iterations.

The difference between **-O3**/**-O4** and **-Mvect** is that optimization levels 3 and 4 (by themselves) perform pipelining on your code *as written*, while **-Mvect** attempts to rearrange your code to make more effective pipelining possible. This is why **-O3**/**-O4** and **-Mvect** are usually used together. **-Mvect** with an optimization level less than 3 will rearrange the code, but no pipelining will be performed; **-O3** or **-O4** without **-Mvect** will perform software pipelining, but will not find as many candidates for pipelining as they would with **-Mvect**. (However, if vector lengths are short, **-O4** alone may work better than **-O4 -Mvect**.)

The vectorization performed by **-Mvect** affects only single nodes. The compiler cannot parallelize vectors by splitting them up among several processors; you must do that yourself.

-Mvect will force an optimization level greater than or equal to 2. **-Mvect -O1** results in the **-O1** being ignored.

How Vectorization Works

Vectorization consists of three processes:

- *Nested loop transformation*—the compiler attempts to rearrange nested loops to increase possibilities for pipelining. For example:

```
            do 100 j = 1, 1000
               do 100 i = 1, 3
                  x(i,j) = x(i,j) * a(i,j)
100         continue
```

  Given this code, the compiler may rearrange the loops so that the loop over *j* becomes the inner loop, resulting in 3 vectors of length 1000 instead of 1000 vectors of length 3.

- *Cache management*—the compiler attempts to perform *streaming* (loading all the operands for a loop into the microprocessor's data cache before beginning the loop) and *stripmining* (breaking a loop into smaller chunks so that the operands for each chunk will fit into the cache).

- *Vector idiom recognition*—the compiler scans the code for certain common vector operations and replaces them with calls to hand-written assembly routines that do the same thing faster. For example, the following source code performs a dot product:

```
do i = 1, 100
   s = s + a(i) * b(i)
enddo
```

The vector idiom recognizer will replace the code produced by these statements with a single call to a hand-coded dot-product routine.

Controlling Vectorization (-Mvect=...)

You can control the vectorizer by specifying options to **-Mvect**. The available options are as follows:

-Mvect=recog	Perform vector idiom recognition and cache management.
-Mvect=transform	Perform nested loop transformation. **transform** is not normally useful without **recog**.
-Mvect=noassoc	Do not rearrange the order of operands in scalar reductions (such as dot product). Rearranging operands can result in faster code, but may give different results due to round-off error.
-Mvect=smallvect[**:***number*]	Assume that no vectorizable **do** loop is iterated more than *number* times. *Number* must be a multiple of 10; if **:***number* is omitted, the value 100 is used. This option improves the performance of doubly-nested, non-perfectly-nested loops, but may result in incorrect code if any vectorizable loop has more iterations than the specified number.
-Mvect=cachesize:*number*	Use at most *number* bytes of the data cache for cache management of vector operations. *Number* must be a multiple of 16, and less than the cache size of the microprocessor (16384 for the i860 XP, 8192 for the i860 XR).
streamlim:*n*	This sets a limit for application of the vectorizer data streaming optimization. If data streaming requires cache vectors of length less than *n*, the optimization is not performed. Other vectorizer optimizations are still performed. The data streaming optimization has a high

overhead compared to other loop optimizations, and can be counter-productive when used for short vectors. The *n* specifier is not optional. The default limit is 32 elements if **streamlim** is not used.

-Mvect=altcode:*number* — Produce non-vectorized code to be executed if the loop count is less than or equal to *number*. Otherwise execute vectorized code. The default value for *number* is 10.

-Mvect with no options means **-Mvect=recog,transform,cachesize:4096,altcode:10**.

You can also control vectorization by using the following switches:

-Msplit_loop_ops=*n* — Set a threshold of *n* floating-point operations within a loop. Innermost loops whose number of floating-point operations exceeds *n* are split. Each floating-point operation counts as two. The default for *n* is 40 when **-Mvect** is used.

-Mnosplit_loop_ops — Do not split loops when the floating-point operation threshold is exceeded. When **-Mvect** is specified, innermost loops whose number of floating point operations exceed 40 are split by default. This switch turns the default off.

-Msplit_loop_refs=*n* — Set a threshold of *n* array element loads and stores within a loop. Innermost loops whose number of loads and stores exceeds *n* are split. The default for *n* is 20 when **-Mvect** is used

-Mnosplit_loop_refs — Do not split loops when the array element loads and stores threshold is exceeded. When **-Mvect** is specified, innermost loops whose number of array element loads and stores exceeds 20 are split by default. This switch turns the default off.

Preventing Associativity Changes (-Mvect=noassoc)

The switch **-Mvect=noassoc** requires a bit more explanation than the others.

In most cases, the rearrangements performed by **-Mvect** do not affect the results of the calculations performed by your program. One exception is that the compiler takes advantage of the associativity of floating-point operations to produce faster code. For example, consider the following dot product.

```
do i = 1, 100
            s = s + a(i) * b(i)
        enddo
```

The order of evaluation of this dot product is as follows:

```
s = ((((s + (a(1)*b(1))) + (a(2)*b(2))) + (a(3)*b(3))) + ... )
```

However, the vector idiom recognizer takes advantage of the associativity of floating-point addition to rearrange it as follows:

```
s = s + (((((a(1)*b(1)) + (a(2)*b(2))) + (a(3)*b(3))) + ... )
```

The rearranged equation is the same algebraically as the original, and runs faster than the original (because it presents a more uniform series of operations for pipelining), but may give slightly different results. You can prevent this type of rearrangement by using the switch **-Mvect=noassoc**.

Getting Information About Vectorization (-Minfo=loop)

You can find out what the vectorizer is doing by using the switch **-Minfo=loop** while compiling with **-Mvect**. This switch sends information about what vectorizations the compiler is performing to the standard error output. For example:

```
% if77 -O4 -Mvect -Knoieee -Minfo=loop -c nas.f
// SW pipelined loop w/ 21 cycles and 2 columns w/ cnt 7 gend for line 27
Vect: streaming data and stripmining loop at line 64. strip size = 1008.
Interchanging loop lines 125, 126
Vect: streaming data and stripmining loop at line 127. strip size = 200.
Vect: loop at line 122 replaced by call to __fill4.
// Software pipelined loop w/ 8 cycles and 3 columns for line 127
// Pipe/Dual-inst 1 column 21 cycle loop gend for line 127
Vect: streaming data for loop at line 164. No stripmine loop required.
// SW pipelined loop w/ 5 cycles and 2 columns w/ cnt 128 gend for line 164
Vect: streaming data and stripmining loop at line 392. strip size = 336.
Vect: loop at line 392 replaced by call to __zxmy4s.
Distributing loop at line 751, 2 new loops
                    •
                    •
                    •
```

Note that optimizations may not be performed in order by line number (for example, the fifth message refers to line 122, while the fourth, sixth, and seventh messages refer to line 127). The meanings of the messages in this example are as follows:

```
// SW pipelined loop w/ 21 cycles and 2 columns w/ cnt 7 gend for line 27
```

This means that the optimizer has performed software pipelining for a loop beginning at line 27 of the source file. Each iteration of this loop takes 21 machine cycles (best-case) to execute. Two "columns" of operations are logically scheduled into the pipelines; that is, there are two sequences of instructions "in the pipeline" at once. The phrase "cnt 7" indicates that the loop has seven iterations, and the word "gend" is an abbreviation for "generated."

```
Vect: streaming data and stripmining loop at line 64. strip size = 1008.
```

This means that the vectorizer has performed cache management by inserting a call to a built-in routine that fills the i860 microprocessor's data cache before the beginning of the loop. Each "strip" (that is, each chunk of data) contains 1008 data values.

The size of the strip is chosen to fill the portion of the cache used by the vectorizer. The larger the amount of data required by each iteration of the loop, the smaller the maximum strip size for that loop. The default for the vectorizer's portion of the cache is 4096 bytes, so in this case each iteration of the loop probably requires four bytes of data. You can change the vectorizer's portion of the cache, and thus the strip size, with the switch **-Mvect=cachesize:***number*.

```
Interchanging loop lines 125, 126
```

This means that the vectorizer has performed nested loop transformation by exchanging two lines of code. This transformation typically gives either more iterations or unit stride in the innermost loop.

```
Vect: streaming data and stripmining loop at line 127. strip size = 200.
```

This message is similar to the previous "streaming data and stripmining loop" message, discussed earlier. This loop has a smaller strip size because it has more data (in this case, about 20 bytes of data are probably required in each loop iteration).

```
Vect: loop at line 122 replaced by call to __fill4.
```

This means that the vectorizer has performed vector idiom recognition by replacing an initialization of an array in a loop with a call to an optimized routine that performs the same function more quickly.

```
// Software pipelined loop w/ 8 cycles and 3 columns for line 127
```

This message is similar to the "SW pipelined loop" message, discussed earlier, except that the number of iterations in the loop could not be determined at compile time (as shown by the lack of a "cnt" phrase in the message). This loop has three columns, so it will be more efficient than the two-column loop shown earlier.

```
// Pipe/Dual-inst 1 column 21 cycle loop gend for line 127
```

This means that the optimizer has made use of the i860 microprocessor's pipelining and dual-instruction mode to optimize a loop.

This message is similar to the previous message, except that a "Software pipelined loop" message means that the vectorizer has inserted loop start-up and shut-down code, while a "Pipe/Dual-inst" message means that the vectorizer is using pipelining and dual-instruction mode within the loop but has not generated any start-up or shut-down code.

```
Vect: streaming data for loop at line 164. No stripmine loop required.
```

This message is similar to the previous "streaming data and stripmining loop" messages, discussed earlier, except that in this case it was not necessary to "stripmine" the loop by gathering data together. For example, this might be an operation on a single array that fits in the cache.

```
// SW pipelined loop w/ 5 cycles and 2 columns w/ cnt 128 gend for line 164
Vect: streaming data and stripmining loop at line 392. strip size = 336.
```

These messages are similar to messages discussed earlier.

```
Vect: loop at line 392 replaced by call to __zxmy4s.
```

This means that the vectorizer has performed vector idiom recognition by replacing user code with a call to an optimized built-in routine (in this case **__zxmy4s()**, a single-precision complex multiply). The list of these routines is not documented because it is subject to change.

```
Distributing loop at line 751, 2 new loops
```

This means that the vectorizer has split a loop with two or more sequences of operations in it into two separate loops, one or both of which may be vectorizable.

Loop Unrolling (-Munroll)

The loop unroller expands the contents of a loop and reduces the number of times a loop is executed. With the **-Munroll** option, you can unroll loops either partially or completely. There are several possible benefits from loop unrolling, including the following:

- Reducing the loop's branching overhead.
- Providing better opportunities for instruction scheduling.

Branching overhead is reduced when a loop is unrolled two or more times, since each iteration of the unrolled loop corresponds to two or more iterations of the original loop. The number of branch instructions executed is proportionately reduced. When a loop is unrolled completely, the loop's branch overhead is eliminated altogether.

Loop unrolling can also be beneficial for the instruction scheduler. When a loop is completely unrolled or unrolled two or more times, opportunities for improved scheduling may be presented. The code generator can take advantage of more possibilities for instruction grouping or filling instruction delays found within the loop.

You can use the **-Minfo** or **-Minfo=loop** option to have the compiler inform you when code is being unrolled. The compiler displays a message indicating the line number and the number of times the code is unrolled.

Making Loops Parallel

The compiler is able to use the three separate processors of an MP node by making some loops parallel by splitting execution of the loop among two or three processors. Each processor is allocated certain iterations of the loop to perform. This can result in greater performance. Both inner and outer loops can be parallelized. For nested loops, the compiler selects the outermost parallelizable loop and makes it parallel.

A loop can be parallelized if its iterations can be performed in any order without affecting the results computed by the loop. For example, one type of loop that cannot be parallelized is one in which the results of some iteration are used in a later iteration. Loops with reductions, such as vector sum or dot product, fit this description. The compiler will try to parallelize this type of loop, but can only do so by performing the sums in a different order than defined by the original loop. As a result, the final sum computed may be slightly off due to roundoff error. If exact results are important, you can use the **-Mconcur=noassoc** switch to prevent parallelization of loops with reductions.

The following sections describe the compiler switches associated with parallelizing loops.

General Loop Parallelization (-Mconcur)

The **-Mconcur** switch causes the compiler to parallelize certain loops. The following options are available:

-Mconcur=altcode:*count* Make innermost loops without reduction parallel only if their iteration count exceeds *count*. Without this switch, the compiler assumes a default *count* of 100.

-Mconcur=altcode_reduction:*count* Make innermost loops with reduction parallel only if their iteration count exceeds *count*. Without this switch, the compiler assumes a default *count* of 200.

-Mconcur=dist:block Make the outermost valid loop parallel. This is the default option.

-Mconcur=dist:cyclic Make the outermost valid loop in any loop nest parallel. If an innermost loop is made parallel, its iterations are allocated to processors cyclically. That is, processor 0 performs iterations 0, 3, 6, ...; processor 1 performs iterations 1, 4, 7, ...; and processor 2 performs iterations 2, 5, 8, and so on.

-Mconcur=global_vcache	Directs the vectorizer to locate the cache within the area of an external array when generating codes for parallel loops. By default, the cache is located on the stack for parallel loops.
-Mconcur=noassoc	Do not make loops with reductions parallel.

Parallelizing Loops with Calls (-Mcncall)

By default, the compiler does not parallelize loops with calls, since there is no way for the compiler to verify that the called routines are safe to execute in parallel. The **-Mcncall** switch forces the compiler to parallelize loops with calls. When you specify **-Mcncall** on the command line, the compiler also automatically specifies **-Mreentrant**.

-Mcncall also allows several other types of loops to be made parallel:

- loops with I/O statements
- loops with conditional statements
- loops with low loop counts
- non-vectorizable loops

If the compiler can detect a cross-iteration dependency in a loop, it will not make the loop parallel, even if **-Mcncall** is specified.

Getting Information About Parallelization

In addition to providing information about vectorization, the **-Minfo=loop** switch also provides information about any loop parallelization that has occured.

The **-Mneginfo=concur** switch prints information for each countable loop that is not made parallel stating why the loop was not made parallel.

Non-IEEE Math (-Knoieee)

The **-Knoieee** switch makes the compiled program use faster but less accurate floating-point math. This can result in a substantial improvement in performance, but may give unacceptable numeric results. If your program relies on the accuracy and exception handling provided by the IEEE 754 standard for floating-point mathematics, do not use this switch. If you do use it, be certain to check your program's results against the expected values.

The effect of the **-Knoieee** switch depends on whether you use it while compiling, while linking, or both.

- To use **-Knoieee** for compilation but not linking, use **-Knoieee** in conjunction with the **-c** switch to compile a source file to a *.o* file, then link the *.o* file into a compiled program *without* **-Knoieee**. For example:

```
% if77 -c -Knoieee myprog.f
% if77 myprog.o
```

- To use **-Knoieee** for linking but not compilation, compile the source file *without* **-Knoieee**, using the **-c** switch to produce a *.o* file, then use the **-Knoieee** switch while linking the *.o* file into a compiled program. For example:

```
% if77 -c myprog.f
% if77 -Knoieee myprog.o
```

- To use **-Knoieee** for both compilation and linking, compile the source file to an executable program *with* **-Knoieee**. For example:

```
% if77 -Knoieee myprog.f
```

Non-IEEE Divides (Compiling with -Knoieee)

The i860 microprocessor does not include a hardware divide unit. By default, the compiler performs floating-point division by calling a routine that conforms to the IEEE standard. This routine correctly handles overflow, underflow, and other exceptional conditions.

If you use the **-Knoieee** switch while compiling a program, the compiler uses a faster but less accurate division routine. This routine is substantially faster than the IEEE routine, but gives results that may differ from the correctly rounded result by as much as three units in the last place.

The non-IEEE division routine is also implemented as inline code rather than a subroutine call, resulting in even greater performance improvements at some increase in code size.

Non-IEEE Math Library (Linking with -Knoieee)

By default, the standard math library conforms to the IEEE standard. The routines in this library handle out-of-range inputs in a well-defined manner and call an exception handler when a denormal is generated in a calculation.

If you use the **-Knoieee** switch while linking a program, the linker uses a different set of math and runtime libraries. These libraries replace the standard math library with compatible routines, many of which are faster but less accurate than their IEEE counterparts. (The rest are identical to their IEEE counterparts.) The square root function in particular has been very carefully optimized. However, the non-IEEE libraries may give unexpected results in response to arguments that are out of the defined domain for the given operation (such as the tangent of 90 degrees).

Using the **-Knoieee** switch when linking also causes the compiler to link in a different initialization routine. The non-IEEE initialization routine sets a flag that causes the microprocessor to immediately flush all denormals to zero on creation. This can make the program run faster, but may give erroneous results if the denormal range is necessary to the result.

BLAS Library (-lkmath)

The **-lkmath** switch links to a highly-optimized math library. This library includes the BLAS (Basic Linear Algebra Subroutines) levels 1, 2, and 3 and some FFT (fast Fourier transform) routines. See the *CLASSPACK Basic Math Library User's Guide* for complete information on this library. You may have to re-code part of your program to use the routines in this library.

Inlining (-Minline)

The **-Minline** switch replaces subprogram calls with inline code. See Chapter 4 for information on using the inliner.

In general, inlining must be used judiciously. Inlining trades the overhead of a subprogram call for larger code, which can overrun the instruction cache and actually decrease performance. You should inline only those routines that meet the following criteria:

- The routine is very small (10 lines of source code or less).
- The routine is called in only one place in the source code, or a few widely-separated places.
- The call (or calls) to the routine occurs in a section of code that is called very often or is otherwise time-critical.

Inlining routines that do not meet these criteria generally results in little or no improvement.

Ignoring Potential Data Dependencies (-Mnodepchk)

The **-Mnodepchk** switch ignores potential data dependencies.

CAUTION

The -Mnodepchk switch can give incorrect or erroneous results, and gives no improvement for many programs, but is provided for those programmers who can make use of it.

Normally, the compiler emits code that will work properly even where data dependencies exist. For example, consider the following code:

```
a(i) = value
variable = a(j)
```

If the compiler does not know the values of the variables *i* and *j* at compile time, it normally assumes that they may have the same value. This is a *data dependency*: if *i* has the same value as *j*, the second statement depends on the first. This is only one example of data dependency; many other types of data dependency exist.

If you use the **-Mnodepchk** switch, the compiler assumes that no data dependencies exist. This can allow the compiler to generate faster code in some cases. In this example, **-Mnodepchk** would allow the compiler to execute the second statement before the first if it results in a more efficient program. However, if any data dependencies do exist, the results will be unpredictable.

Use the **-Mnodepchk** switch only if you understand the program very well and are sure that no data dependencies exist.

Code Changes for Optimization

This section lists some changes you may be able to make in your code that will make the code more efficient or make the jobs of the optimizer and vectorizer easier.

General Improvements

These changes can improve almost all types of code:

- Split larger programs into smaller pieces and use appropriate optimization levels on each piece. For example, **-Mvect** makes vector codes faster, but can make non-vector codes slower. If a single source file contains both vector and non-vector code, you should split it into vector and non-vector pieces and compile the two pieces separately, with and without **-Mvect**. The program **fsplit** can be used to help split your program up.

- Keep basic blocks under 30 lines of code. A *basic block* is a group of program statements in which the flow of control enters at the beginning and leaves at the end without the possibility of branching (except at the end). Small basic blocks give the compiler more opportunities to rearrange code for optimizations.

- Avoid type conversions (for example, the assignment of a double-precision value to a single-precision variable). Type conversions are time-consuming operations that are often unnecessary. Conversions between floating-point and integer types are particularly difficult. Examine your code and be sure that variables that are used together are of the same type, except where different types are needed.

Loop Improvements

These changes make it easier for the vectorizer to assemble long sequences of similar operations, which allow the i860 microprocessor to work the most efficiently. These changes can be very effective in improving the performance of code that uses floating-point vectors.

- Use unit stride (each iteration of a loop works on the next vector element, rather than skipping elements). This results in efficient pipelines. This is one of the most important changes you can make.

- Use countable loops (loops which are iterated a loop-invariant number of times). The compiler can create more efficient code for a loop whose iteration count is known at compile time than it can for a loop whose iteration count is not known until the program executes (such as a loop from 1 to *n* or a loop that terminates when a certain condition is true).

- Use constants for the bounds and increment value in **do** statements.

- Use perfectly-nested loops (loops that have no code outside the innermost loop). Here is an example of a perfectly-nested loop:

```
      do 100 k = 1,10
         do 100 j = 1,10
            do 100 i = 1,2000
                  •
                  •    all loop operations here
                  •
100   continue
```

  Perfectly-nested loops also terminate only at a loop-control statement; they do not have any "early outs."

- In nested loops, make the loop with the highest iteration count in the innermost loop. This gives the vectorizer the longest uninterrupted string of operations to work with.

- Keep data dependence distances short. The *data dependence distance* of a loop is determined by the proximity in memory of the different data objects that are accessed in the body of a loop. For example, a loop that accesses vector elements *a(n)* and *a(n+5)* has a data dependence distance of 5. For best results, inner loops should have a data dependence distance of less than 8 for double-precision vectors and less than 16 for real vectors.

- Avoid **if** statements within loops. If the compiler can't be sure that the code that is executed on each iteration of a loop is the same as the code in the previous iteration, it cannot set up a pipeline. Instead of writing an **if** statement within a loop, write the loop within the **if** statement. For example, if your code looks like this:

```
      do 100 i = 1, 1000
c        code for all conditions
         if(a .gt. b) then
c           code for a > b
         endif
100   continue
```

Rewrite it as follows:

```
      if(a .gt. b) then
         do 100 i = 1, 1000
c           code for all conditions
c           code for a > b
100      continue
      else
         do 101 i = 1, 1000
c           code for all conditions
101      continue
      endif
```

Note that this example assumes that the variables *a* and *b* are not changed in the loop body. If the condition in the **if** statement depends on code within the loop, you cannot rearrange the loops in this way.

- Avoid divides and type conversions within loops. Division and type conversion are operations that cannot be performed in hardware by the i860 microprocessor, so loops containing these operations cannot be pipelined as effectively.

File I/O Improvements

If your program reads and writes sizeable data files, you can obtain substantial improvements in performance with these changes:

- Move the data files to PFS™ (Parallel File System™) file systems. Access to PFS file systems is substantially faster than access to ordinary non-parallel file systems for large files.

- Use sequential unformatted I/O rather than formatted I/O. Formatted file I/O guarantees portability between different Fortran programs, but uses a lot of compute cycles on each read or write. If you don't need this portability (for example, if the file is used only by one or two programs), you can improve the efficiency of file I/O by using unformatted I/O.

- Use parallel I/O calls (**cread()**, **cwrite()**, **lseek()**) rather than Fortran I/O. These calls are more efficient than Fortran's built-in I/O statements.

- Use asynchronous I/O (**iread()**, **iwrite()**). The asynchronous calls let your program work while reads or writes are in progress. You can also use asynchronous I/O to perform *double buffering*: reading data into a buffer, then reading into a second buffer while simultaneously processing the data in the first buffer.

See the *Paragon™ System User's Guide* for more information on the techniques discussed in this section.

Using the Inliner 4

This chapter describes the compiler's subprogram inlining capability.

Subprogram inlining is a compiler optimization under which the body of a subprogram is expanded in place of a call to the subprogram. This can speed up execution by eliminating the parameter passing and subprogram call and return overhead. Inlining a subprogram body also creates opportunities for other compiler optimizations. Inlining will usually result in larger code size (although in the case of very small subprograms, code size can actually decrease). Using inlining indiscriminately can result in much larger code size and no increase in execution speed; there may even be a decrease in execution speed.

There are basically two ways to accomplish inlining:

- **Automatic inlining** as part of the compilation process. When you use the **-Minline** switch during compilation, the compiler first looks in the source files for subprograms that can be inlined, then replaces calls to those subprograms with the equivalent code automatically.
- Use of **inliner libraries**. When you use the **-Mextract** switch during compilation, the compiler looks for subprograms that can be inlined and extracts them into an *inliner library*. Later, when compiling a program that calls subprograms in the inliner library, you use the **-Minline** switch and specify the library; the compiler replaces calls to the subprograms in the library with the equivalent code.

Compiler Inline Switch

To request subprogram inlining, use the **-Minline** switch:

```
-Minline=option[,option...]
```

where *option* is one of the following:

[name:]*subprogram*
: Specifies a particular subprogram to inline. If **name:** is not used, the subprogram name must not contain a period. Any number of names can be specified.

NOTE

Inlining in Fortran is case sensitive. You must use lowercase when specifying the names of subprograms to be inlined with the **-Minline** switch.

[size:]*number*
: Specifies an upper bound on subprogram size to inline. Any subprogram less than the specified number of lines (approximately) will be inlined.

[lib:]*library*
: Specifies a library of inlined subprograms. If **lib:** is not used, the library name must contain a period. Any number of libraries can be specified. A subprogram is inlined if it is found in any of the libraries.

levels:*number*
: Specifies the number of levels of inlining to perform (default 1). For example, suppose subprogram *a* calls *b* and *b* calls *c*. If you want to completely inline *a* (including the calls to *b* and *c*), you must use **-Minline=a,b,c,levels:2**.

You must specify at least one name, size, or library. If both subprogram name(s) and a size limit are specified, a subprogram is inlined if it is named or if it satisfies the limit.

Inlining can be either automatic or manual. If you do not specify any inliner libraries, the compiler performs a special pass for all source files named on the command line before any of them are compiled. This pass extracts subprograms that meet the requirements for inlining and puts them in a temporary library for use by the compilation pass.

If you specify one or more inliner libraries, the compiler does not perform an initial extract pass. Instead, subprograms to be inlined are selected from the specified libraries. If neither subprogram names nor a size limit are specified, any subprogram in the library meets the conditions for inlining.

Creating an Inliner Library

To create or update an inliner library, use the **-Mextract** switch:

```
-Mextract[=option[,option...]]
```

where *option* is one of the following:

[**name:**]*subprogram*
Extracts the specified subprogram. **name:** must be used if the subprogram name contains a period.

[**size:**]*number* Extracts subprograms containing less than approximately *number* statements.

If you don't specify any *options* with **-Mextract**, the compiler attempts to extract all subprograms of a reasonable size.

When you use **-Mextract**, only extraction is performed; compilation and linking are not performed.

If the **-Mextract** switch is present, you must also specify a single inliner library name on the compiler command line. For example:

```
-o inliner_library_name
```

This specifies the inliner library in which the extracted forms of subprograms are placed. The library may or may not already exist; it is created if it does not.

You can use the **-Minline** switch at the same time as the **-Mextract** switch. In this case, the extracted form of the subprogram can have other subprograms inlined into it. This makes it possible to obtain more than one level of inlining. In this situation, if no library is specified with **-Minline**, processing will consist of two extract passes. The first pass is the hidden pass implied by **-Minline** during which subprograms are extracted into a temporary library. The second pass uses the results of the first pass but puts its results into the library specified with the **-o** switch. See examples below.

Using Inliner Libraries

An inliner library is implemented as a directory. For each element of the library, the directory contains a file containing the encoded form of the inlinable subprogram.

A special file named *TOC* serves as a directory for the library. This is a printable, ASCII file that can be examined to find out information about the library contents, such as names and sizes of subprograms, the source file from which they were extracted, the version number of the extractor that created the entry, etc.

Libraries and their elements can be manipulated using ordinary system commands, for example:

- You can rename a library with **mv**.
- You can remove an element from a library with **rm**, or remove an entire library with **rm -r**.
- You can copy an element from one library to another with **cp**, or copy an entire library with **cp -r**.
- You can examine the contents of a library with **ls**, or determine the modification date of an element with **ls -l**.

Since deleting or adding an element can cause the *TOC* file to become out of date, a utility program **ifixlib** is provided to recreate a correct *TOC* file. Use it as follows:

```
% ifixlib library_name
```

When use of the **if77** command causes an entry to be created or updated, the date of the most recent change of the library directory itself is updated also. This allows a library to be listed as a dependency in a makefile, in order to ensure that the necessary compilations are performed again when a library is changed.

Restrictions on Inlining

The following Fortran subprograms cannot be extracted:

- Main or **BLOCK DATA** programs
- Subprograms containing alternate return, computed **GOTO**, assigned **GOTO**, **DATA**, **SAVE**, or **EQUIVALENCE** statements
- Subprograms containing **FORMAT** statements
- Subprograms containing multiple entries

A Fortran subprogram is not inlined if any of the following applies:

- It is referenced in a statement function.
- There exists a common block mismatch; i.e., the caller must contain all common blocks specified in the callee, and elements of the common blocks must agree in name, order, and type (except that the caller's common block can have additional members appended to the end of the common block).
- There exists an argument mismatch; i.e., the number of actual and formal parameters must be equal.
- There exists a name clash; e.g., a call to subroutine **xyz** in the extracted subprogram and a variable named *xyz* in the caller.
- A constant actual parameter in the caller has an assignment to its associated formal parameter in the extracted subprogram.

The compiler gives you an error message if you violate any of these restrictions. The severity of the error varies, depending on the type of the error and how far the compiler has gone in the inlining process before detecting it.

Error Detection During Inlining

When invoking the inliner, you should always set the diagnostics reporting switch (**-Minfo=inline**).

An additional feature associated with inlining is enhanced compiler error detection. For example:

- If an inlinable subprogram is called with the wrong number of arguments, a warning message is issued and the subprogram is not inlined.
- If an inlinable subprogram is called in a context which assumes that a value is returned, but the body of the subprogram does not contain any statements that set the return value, a severe error is issued.
- If the declaration of an external variable referenced by an inlinable subprogram does not match the declaration in the source file being compiled, a severe error is issued.

Efficiency Considerations

To ensure that compiler vectorizer optimizations are not impeded, observe the following guidelines when inlining Fortran subprograms:

- Avoid inlining subprograms whose formal parameters are adjustable arrays. For example, this fragment will vectorize well:

```
subroutine x(a)
integer n
parameter (n = 100)
double precision a(n, n)
```

  However, this fragment will not vectorize well:

```
subroutine x(a, n)
integer n
double precision a(n, n)
```

- Avoid actual parameters that are elements of arrays, except when the element specified is the first element of the array. For example:

```
      program p
      ...
      integer actparam(3:10,2:8,9)
      ...
C The next call will not inline efficiently
      call inline_sub(actparam(4,6,2))
C The next call will inline efficiently
      call inline_sub(actparam(3,2,1))
      ...
      end
```

Examples

This section contains examples of using the inliner.

Dhry

Assume the program **dhry** consists of a single source file *dhry.f*. Then, the following command line builds an executable for **dhry** in which *Proc7* has been inlined wherever it is called:

```
% if77 dhry.f -Minline=Proc7
```

The following command line builds an executable for **dhry** in which *Proc7* plus any subprograms of roughly three or fewer statements have been inlined (1 level only).

```
% if77 dhry.f -Minline=Proc7,3
```

The following commands build an executable for **dhry** in which all subprograms of roughly ten or fewer statements are inlined. Two levels of inlining will have been performed. This means that if subprogram A calls subprogram B, and B calls C, and both B and C are inlinable, then the version of B that is inlined into A will have had C inlined into it.

```
% if77 dhry.f -Mextract=10 -Minline=10 -o temp.ilib
% if77 dhry.f -Minline=temp.ilib
% rm -r temp.ilib
```

Fibo

Assuming *fibo.f* contains a single subprogram **fibo** that calls itself recursively. Then, the following command line creates file *fibo.o* in which **fibo** has been inlined into itself:

```
% if77 fibo.f -c -Minline=fibo -O
```

Because this version of *fibo* recurses only half as deeply, it should execute noticeably faster.

Makefiles

The following fragment of a makefile assumes that file *utils.f* contains a number of small subprograms that are used in the files *parser.f* and *alloc.f*. An inliner library *utils.ilib* is maintained. Note that the library must be updated whenever *utils.f* or one of the include files it uses is changed. In turn, *parser.f* and *alloc.f* must be compiled again whenever the library is updated.

```
            .
            .
            .
main.o: $(SRC)/main.f $(SRC)/global.h
        $(F77) $(F77FLAGS) -c $(SRC)/main.f
utils.o: $(SRC)/utils.f $(SRC)/global.h $(SRC)/utils.h
        $(F77) $(F77FLAGS) -c $(SRC)/utils.f
utils.ilib: $(SRC)/utils.f $(SRC)global.h $(SRC)/utils.h
        $(F77) $(F77FLAGS) -Mextract=15 -o utils.ilib
parser.o: $(SRC)/parser.f $(SRC)/global.h  utils.ilib
        $(F77) $(F77FLAGS) -Minline=utils.ilib -c $(SRC)/parser.f
alloc.o: $(SRC)/alloc.f $(SRC)/global.h  utils.ilib
        $(F77) $(F77FLAGS) -Minline=utils.ilib -c $(SRC)/alloc.f

myprog: main.o utils.o parser.o alloc.o
        $(F77) -o myprog main.o utils.o parser.o alloc.o
```

Interfacing Fortran and C 5

This chapter describes how to use C and Fortran routines together in the same program.

Calling a C Function from Fortran

The Fortran compiler adds an underscore (_) at the beginning and end of every external name (function, subroutine and common), and expects all external names to begin and end with an underscore. However, the C compiler only adds an underscore at the beginning of each external name. This means that to make a C function callable from Fortran, the name that you give it (in the C source) must end with an underscore. If you want to call an existing function whose name does not end with an underscore, you must write a "wrapper" function, whose name does end with an underscore, which just calls the existing function.

Also, any dollar signs in a C external name are replaced with underscores (or you can choose another replacement character by using the **-Mdollar** switch when you compile the program). For example, to call the C function **my$func_()** from Fortran, you would call it as **my_func()**.

You can also use a C pragma to prevent the compiler from appending an underscore to the function name. The C pragma directive has the following form:

```
comment_char$pragma C (id [,id] ...)
```

where

`comment_char`	C, D, or * in column 1 or a ! in any column
`pragma`	either pragma or PRAGMA
`id`	iname of an external function

The C pragma directive marks external functions writen in C. The compiler does not append an underscore to the specified identifiers. The following example shows a sample C pragma.

```
EXTERNAL FUNC1, FUNC2 !$PRAGMA C(FUNC1, FUNC2)
```

All Fortran arguments are passed by reference. (Temporary storage for non-addressable objects such as literals is provided by the compiler.) Therefore, each parameter in the called C routine must be a pointer of the appropriate type, as shown in Table 5-1.

Table 5-1. Fortran Data Types for Called C Functions

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

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

If the C function being called from Fortran returns a value, then the return types correspond as follows:

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

If a Fortran caller calls a C function as a subroutine with alternate return parameters, the value returned by the C function (using **return(*e*)**) is interpreted as the expression in the Fortran alternate return statement **RETURN** *e*. The Fortran caller does a computed **GOTO** on the return value to implement the alternate return.

Calling a Fortran Routine from C

The Fortran compiler adds an underscore (_) at the beginning and end of every external name (function, subroutine and common), while the C compiler only adds an underscore at the beginning of each external name. This means that to call a Fortran routine or refer to a Fortran **COMMON** block from C, you must append an underscore to its name. For example, to call the Fortran routine **myfunc()** from C, you would call it as **myfunc_()**.

All Fortran parameters are passed by reference. Therefore, the corresponding argument in the C call must be a pointer of the appropriate type, as shown in Table 5-2. For example, to pass the scalar variable **x** from C to Fortran, use the argument value **&x**.

Table 5-2. C Data Types for Called Fortran Routines

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

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

If the Fortran routine being called from C is a **FUNCTION**, then the return types correspond as follows:

- An **INTEGER** or **LOGICAL** Fortran **FUNCTION** must be declared as **int** in the calling C routine.
- A **DOUBLE PRECISION** Fortran function must be declared as **double** in the calling C routine. Since C usually promotes **float** return values to **double**, a **REAL** return value may not be accessible in C. (You can use the **-Msingle** switch when compiling the calling C program to suppress the promotion of **float** to **double**.)

- **COMPLEX, DOUBLE COMPLEX,** and **CHARACTER** are returned from the called Fortran routine by passing the address where the return value is to be stored as an extra first parameter to the C function. The length of a **CHARACTER** return value is passed as an extra second **int** parameter to the C function.

The alternate return statement of Fortran, **RETURN** *e*, has no equivalent in C.

Extensions to ANSI Fortran 6

This chapter describes the following extensions to the standard language (i.e., features and capabilities not described in the *American National Standard Programming Language FORTRAN, ANSI x3.9-1978*):

- Extensions derived from VAX/VMS and IBM/VS
- Extensions derived from Cray Fortran
- Other I/O extensions
- Subroutine and intrinsic extensions
- Additional intrinsic functions
- Vector intrinsics

See the *Paragon™ System Fortran Language Reference Manual* for a complete description of the language accepted by the **if77** compiler and more details on the extensions described in this chapter.

Standard Language

The Fortran compiler compiles programs written in a true superset of ANSI standard Fortran 77, as described in the *American National Standard Programming Language FORTRAN, ANSI x3.9-1978*. There are no deviations from this language standard.

The compiler also supports the requirements of the Military Standard, MIL-STD-1753.

Instead of fully specifying the language accepted by the compiler, this chapter describes only those features that differ from the Fortran language specified in the ANSI standard cited above. Most of the differences (incompatibilities and extensions) are VAX/VMS and IBM/VS features.

Extensions Derived from VAX/VMS and IBM/VS

The Fortran compiler provides partial or full support for the following VAX/VMS and IBM/VS extensions:

- Compiler directives
- Control statements
- Data related
- Format related
- Lexical related
- I/O related

The following VMS Fortran statements are not supported:

DELETE	**FIND**	**REWRITE**
UNLOCK	**DICTIONARY**	

Compiler Directives

The Fortran compiler recognizes three VMS compiler directives:

%NOLIST	Turns off listing of source lines in the listing file (including the **%NOLIST** line itself).
%LIST	Turns the listing back on for the next line.
%EJECT	Causes a new listing page to be started.

These directives have an effect only when the **-Mlist** command line switch is used. All directives must begin in column one.

OPTIONS Statement

The **OPTIONS** statement can be used to override or confirm certain compiler command-line switches. The statement has the form:

```
OPTIONS /option [/option ...]
```

The recognized *option*s are:

CHECK=ALL	No effect (recognized, but ignored).
CHECK=[NO]OVERFLOW	No effect.
CHECK=[NO]BOUNDS	No effect.
CHECK=[NO]UNDERFLOW	No effect.
CHECK=NONE	No effect.
NOCHECK	No effect.
[NO]EXTEND_SOURCE	(Don't) enable the **-Mextend** switch.
[NO]F77	(Don't) enable the **-Mstandard** switch.
[NO]G_FLOATING	No effect.
[NO]I4	(Don't) enable the **-Mi4** switch.
[NO]RECURSIVE	(Don't) enable the **-Mrecursive** switch.
[NO]REENTRANT	(Don't) enable the **-Mreentrant** switch.
[NO]STANDARD	(Don't) enable the **-Mstandard** switch.

See Chapter 2 for more information on these switches.

Restrictions:

- The **OPTIONS** statement must be the first statement in a program unit, preceding the **PROGRAM**, **SUBROUTINE**, **FUNCTION**, and **BLOCKDATA** statements.
- The *option*s override the values from the compiler command line for the program unit immediately following the **OPTIONS** statement.
- Any prefix of the option sufficiently long to uniquely identify the option is a legal abbreviation.
- Upper or lower case is not significant, unless the switch **-Mupcase** is present on the command line. If **-Mupcase** has been selected, the options must be in lower case.

Control Statements (DO, DO WHILE, and ENDDO)

- The **DO** statement has the form:

  ```
  DO [s[,]]v=e1,e2[,e3]
  ```

  Support is provided for the VMS Fortran extension that allows the statement label to be omitted. If the optional label, *s*, is not included, the **DO** statement must be terminated by an **ENDDO** (details follow in this section). VAX/VMS "Extended Range" **DO** loops are supported.

- The **DO WHILE** statement has the form:

  ```
  DO [s[,]] WHILE (e)
  ```

  where *e* is a logical expression and *s* is an optional label of a statement that must physically follow in the same program unit. The **DO WHILE** statement executes for as long as the logical expression *e* continues to be true when tested at the beginning of each iteration. If *e* is false, control transfers to the statement following the loop. The label *s* is optional when an **ENDDO** is used to terminate the loop (see below).

- An **ENDDO** statement may optionally terminate an indexed **DO** or **DO WHILE** statement (see previous section). The **ENDDO** statement is required for a **DO** or **DO WHILE** statement which does not contain a terminal-statement label. The **ENDDO** statement may also be used as a labeled terminal statement if the **DO** or **DO WHILE** statement contains a terminal-statement label.

Data Extensions

Data Types

The size of a data type may be specified by appending a data type length specifier of the form ***n*** to the data type name. For example, **REAL*8** is equivalent to **DOUBLE PRECISION**.

Table 6-1 shows the lengths of data types and their meanings.

Table 6-1. Data Type Extensions

Type	Meaning	Size
LOGICAL*1	Small Logical	1 byte
LOGICAL*2	Short Logical	2 bytes
LOGICAL*4	**LOGICAL**	4 bytes
BYTE	Small Integer	1 byte
INTEGER*2	Short Integer	2 bytes
INTEGER*4	**INTEGER**	4 bytes
REAL*4	**REAL**	4 bytes
REAL*8	**DOUBLE PRECISION**	8 bytes
COMPLEX*8	**COMPLEX**	8 bytes
COMPLEX*16	**DOUBLE COMPLEX**	16 bytes

The new **BYTE** type is treated as a signed one-byte integer.

Assignment of a value too big for the data type to which it is assigned is an undefined operation.

VMS data type length specifiers are fully supported except for **REAL*16** (Quad Precision).

A symbolic name can be followed by a data type length specifier of the form ***s***, where *s* is one of the acceptable lengths for the data type being declared. Such a specification overrides the length attribute that the statement implies and assigns a new length to the specified item. If a data type length specifier is specified with an array declarator, the data type length specifier goes immediately after the array name. Unlike VAX/VMS Fortran, a specifier is allowed after a **CHARACTER** function name even if the **CHARACTER** type word has a specifier. For example:

```
CHARACTER*4 FUNCTION C*8()
```

is allowed by the Fortran compiler, but not by VAX/VMS Fortran.

The storage given to **INTEGER** and **LOGICAL** types is four bytes. A compiler switch to allow the default for these types to be two bytes is not supported.

The storage given to **REAL** type is four bytes; for **DOUBLE PRECISION**, it is eight bytes.

The floating point format supported is machine-dependent. VAX/VMS supports its own floating point format.

Intrinsic support for the new data types is provided.

VAX/VMS Fortran supports logical data items to be used with any operation where a similar sized integer data item is permissible and vice versa. The logical data item is treated as an integer or the integer data item is treated as a logical of the same size without any conversion.

VAX/VMS Fortran sign extends the result when a logical data item is assigned an integer or logical value of a different size. This is supported.

Floating point data items may be used as array subscripts and in computed **GOTO**s. VAX/VMS Fortran allows this and the float is converted to integer. Floating point data items are not permitted in array bounds and alternate returns.

The type of an arithmetic expression corresponds to the type specified for VAX/VMS Fortran. The type of an expression is determined by the rank of its elements. Table 6-2 shows the ranks of data types from lowest to highest.

Table 6-2. Data Type Ranks

Data Type	Rank
LOGICAL	1 (lowest)
INTEGER*2	2
INTEGER*4	3
REAL*4	4
REAL*8 (Double precision)	5
COMPLEX*8 (Complex)	6
COMPLEX*16 (Double complex)	7 (highest)

The data type of a value produced by an operation on two arithmetic elements of different data types is the data type of the highest-ranked element in the operation, except that an operation involving a **COMPLEX*8** data type and a **REAL*8** data type produces a **COMPLEX*16** result (The **REAL*8** element is not rounded).

The type of a logical expression is always a **LOGICAL*4** result.

Decimal Integer Constants

The form for a decimal integer constant is:

$$[s]d_1d_2...d_n$$

where d_i is a digit in the range 0 to 9 and where s is an optional sign. The value of an integer constant must be within the range -2147483648 to 2147483647 inclusive (-2^{31} to (2^{31} - 1)). All integer constants assume a data type of **INTEGER*4** and have a 32-bit storage requirement.

NOTE

VAX/VMS Fortran stores integer constants as either 16-bit quantities or 32-bit quantities depending on their size. Passing integer constants as actual arguments to dummy arguments of smaller size is machine-dependent and is an undefined operation.

Octal/Hexadecimal Constants

Octal and hexadecimal constants are handled alike.

The form for an octal constant is:

$$'c_1c_2...c_n'O$$

The form for a hexadecimal constant is:

$$'a_1a_2...a_n'X$$

where c_i is a digit in the range 0 to 7 and where a_i is a digit in the range 0 to 9 or a letter in the range A to F or a to f (case mixing is allowed). You can specify up to 64 bits (22 octal digits, 16 hexadecimal digits).

NOTE

VAX/VMS Fortran supports up to 128 bits.

Octal and hexadecimal constants stored as either 32-bit or 64-bit quantities. If their number of digits are represented by less than the necessary size, they are padded on the left with zero. They assume data types based on the way they are used. The rules for data type conversion of constants are as follows:

- The size of the constant is *always* either 32 or 64 bits and has a typeless data type. Sign-extension and type-conversion are never performed. All binary operations are performed on 32-bit or 64-bit quantities. This implies that the rules to follow are only concerned with mixing 32-bit and 64-bit data.

- When the constant is used with an arithmetic binary operator, including the assignment operator, and the other operand is not typeless, the constant assumes the type and size of the other operand.

- When the constant is used in a relational expression, such as **.EQ.**, the size is chosen from the operand having the largest size. This implies that 64-bit comparisons are possible.

- When the constant is used as an argument to the generic **AND**, **OR**, **EQV**, **NEQV**, **SHIFT**, or **COMPL** function, a 32-bit operation is performed if no argument is more than 32 bits in size, otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.

  When the constant is used as an actual argument in any other context, no data type is assumed; however, a length of four bytes is always used. If necessary, truncation on the left occurs.

- When a specific 32-bit or 64-bit data type is required, that type is assumed for the constant. An example of a required specific data type is in array subscripting.

- When the constant is used in a context other than that mentioned above, an **INTEGER*4** data type is assumed. Examples include arithmetic binary operations with other untyped constants and in logical expressions.

- When the required data type for the constant implies that the length needed is more than the number of digits specified, the leftmost digits have a value of zero. When the required data type for the constant implies that the length needed is less than the number of digits specified, the constant is truncated on the left. Truncation of nonzero digits is allowed.

  In the example below, the `INTEGER*4 I` and `INTEGER*2 J` will have the hex value `1234` and `4567` respectively. The `REAL*8 D` variable will have the hex value `0x4000012345678954` after its second assignment.

```
I = '1234'X        ! Leftmost Pad with zero.
J = '1234567'X     ! Truncate Leftmost 3 hex digits
D = '40000123456789ab'X
D = EQV(D,'ff'X)   ! 64-bit Exclusive Or
```

Hollerith Constants

Hollerith constants and character constants are handled alike but in a manner somewhat different from hexadecimal and octal constants.

The form of a Hollerith constant is:

$$nHc_1c_2...c_n$$

where *n* specifies the positive number of characters in the constant and cannot exceed 2000 characters. A Hollerith constant is stored as a byte string with four characters per 32-bit word. Hollerith constants are untyped arrays of **INTEGER*4**. The last word of the array is padded on the right with blanks if necessary. Hollerith constants cannot assume a character data type and cannot be used where a character value is expected. Unlike VAX/VMS Fortran, Hollerith constants are permitted with the **%REF** Built-In function. (Refer to the section "Subroutine and Intrinsic Extensions" (on page E-25) for details on **%REF**). A Hollerith constant used in a numeric expression assumes the data type according to the following rules. (Note, these rules also apply to character constants used in a numeric context.)

- Sign-extension is never performed.
- The byte size of the Hollerith constant is determined by its context and is not strictly limited to 32 or 64 bits like hexadecimal and octal constants.
- When the constant is used with a binary operator, including the assignment operator, the data type of the constant assumes the data type of the other operand.
- When a specific data type is required, that type is assumed for the constant. When an integer or logical is required, **INTEGER*4** and **LOGICAL*4** are assumed. When a float is required, **REAL*4** is assumed. An example of a required specific data type is in array subscripting.
- When the constant is used as an argument to the generic **AND**, **OR**, **EQV**, **NEQV**, **SHIFT**, or **COMPL** function, a 32-bit operation is performed if no argument is more than 32 bits in size, otherwise, a 64-bit operation is performed. The size of the result corresponds to the chosen operation.

  When the constant is used as an actual argument, no data type is assumed and the argument is passed as an **INTEGER*4** array. Character constants are passed by descriptor only.
- When the constant is used in any other context, a 32-bit **INTEGER*4** array type is assumed.

When the length of the Hollerith constant is less than the length implied by the data type, spaces are appended to the constant on the right. When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right.

Character Constants

Character constants may be used in a *numeric* context (for example, as the expression on the right side of an arithmetic assignment statement). The rules for typing and sizing of character constants used in a numeric context follows the same rules given for Hollerith constants as outlined in the preceding section. Note that character constants as actual arguments are always passed by descriptor.

Logical Representation

The logical constants **.TRUE.** and **.FALSE.** are defined to be the four-byte values -1 and 0 respectively. A logical expression is defined to be true if its least significant bit is 1 and false otherwise. This definition conforms exactly to the VAX/VMS definition.

The abbreviations, **T** and **F**, can be used as an alternative to **.TRUE.** and **.FALSE.** in data initialization statements or in namelist input.

Data Initialization

The VAX/VMS extension to allow data initialization within data type declaration statements is supported fully. Data is initialized by placing values bounded by slashes immediately following the symbolic name (variable or array) to be initialized. Initialization of fields within structure declarations is allowed. Unnamed fields cannot be initialized. Initialization of records is not allowed.

Hollerith, octal or hexadecimal constants can be used to initialize data in both data type declarations or in **DATA** statements. Truncation and padding occur for constants that differ in size from the data item declared as specified in the previous section on constants.

The requirement that the data initialization part must agree with the number of variable elements is relaxed for declaration statements in order to support an IBM/VS extension. **DATA** statement initialization requirements are not relaxed. For example, the following declaration statement is acceptable and will initialize the first ten elements of the array *A* to the value 3.

```
INTEGER A(20)/10*3/
```

PARAMETER Statement

The extensions to the **PARAMETER** statement supported by VAX/VMS Fortran are fully supported. The two extensions to the **PARAMETER** statement are as follows:

- Its list is not bounded with parentheses.
- The form of the constant rather than the implicit or explicit typing of the symbolic name, determines the data type of the variable.

The form of the alternative **PARAMETER** statement is:

```
PARAMETER p=c [,p=c]...
```

where *p* is a symbolic name and *c* is a constant, symbolic constant, or a compile time constant expression. See the *Paragon™ System Fortran Language Reference Manual* for details.

Common Blocks

Records are allowed to be named within common blocks. Since the storage requirements of records are machine-dependent, the size of a common block containing records may vary between machines. Note that this may also affect subsequent equivalence associations to variables within common blocks that contain records.

Both character and non-character data may reside in one common block. Data is aligned within the common block in order to conform to machine-dependent alignment requirements.

A common block may be data initialized in more than one program unit if the existing system environment allows it (note that COFF-based systems do not). It is up to the programmer to make sure that data within one common block is not initialized more than once.

Blank common may be data initialized.

EQUIVALENCE Statement

An array element may be identified with a single subscript in an **EQUIVALENCE** statement even though the array is defined to be a multidimensional array. See the *Paragon™ System Fortran Language Reference Manual* for details.

Equivalence of character and non-character data is allowed as long as misalignment of non-character data does not occur.

Records and record fields cannot be specified in **EQUIVALENCE** statements.

IMPLICIT Statement

The use of the keyword **NONE** with the **IMPLICIT** statement is supported. The form is:

```
IMPLICIT NONE
```

See the *Paragon™ System Fortran Language Reference Manual* for details.

Since symbol names may begin with dollar sign ($) or underscore (_), these characters by default are of type **REAL**. In an **IMPLICIT** statement, these characters may be used in the same manner as other characters. They cannot be used in a range specification. A valid example is:

```
IMPLICIT INTEGER (A-D,$,_)
```

VOLATILE Statement

The **VOLATILE** statement inhibits all optimizations on the variables, arrays, and common blocks that it identifies. The form of this statement is:

```
VOLATILE nitem [,nitem]...
```

where each *nitem* is the name of a variable, array, or common block. The name of a common block must be enclosed in slashes. If *nitem* names a common block, all members of the common block are volatile.

The volatile attribute of a variable is inherited by any direct or indirect equivalences. For example:

```
COMMON /COM/ C1, C2
VOLATILE /COM/, /DIR/     ! /COM/ and /DIR/ are volatile
EQUIVALENCE (DIR, X)      ! X is volatile
EQUIVALENCE (X, Y)        ! Y is volatile
```

ENTRY Statement

The **ENTRY** statement provides multiple entry points within a subprogram. Entry names within a **FUNCTION** subprogram need not be of the same data type as the function name, but they all must be consistent within one of the following groups of data types:

- **BYTE, INTEGER*2, INTEGER*4, LOGICAL*1, LOGICAL*2, LOGICAL*4, REAL*4, REAL*8, COMPLEX*8**
- **COMPLEX*16**
- **CHARACTER**

If the function is of character data type, all entry names must also have the same length specification as that of the function.

Structures

A structure is an aggregate data type that may consist of multiple heterogeneous data types. A structure declaration block is used to declare this user-defined type. This declaration is composed of a **STRUCTURE** statement followed by the declaration body that declares one or more fields and is finally followed by the **END STRUCTURE** statement. Fields within structures are aligned in order to conform to machine-dependent alignment requirements. Alignment of fields also provides a C-like "struct" building capability and allows convenient inter-language communications. Note that aligning of structure fields is not supported by VAX/VMS Fortran. Refer to the "Records" section (on page E-12) for an example. The form of a structure declaration is as follows:

```
STRUCTURE [/structure_name/][field_namelist]
          field_declaration
          [field_declaration]
                    ...
          [field_declaration]
END STRUCTURE
```

where *structure_name* is unique, is used to identify a structure and is used in subsequent **RECORD** statements to refer to the structure. *Field_namelist* is a list of fields having the structure of the associated structure declaration. A *field_namelist* is allowed only in nested structure declarations. *Field_declaration* can consist of any combination of substructure declarations, typed data declarations, union declarations or unnamed field declarations.

Field names within the same declaration nesting level must be unique, but an inner structure declaration can include field names used in an outer structure declaration without conflict. Also, since periods are used in record references to separate fields, it is not legal to use relational operators (for example, **.EQ.**, **.XOR.**) logical constants (**.TRUE.** or **.FALSE.**) and logical expressions (**.AND.**, **.NOT.**, **.OR.**) as field names in structure declarations.

Fields declared in a structure are aligned according to the dependencies imposed by the hardware and hence a structure's storage requirement is machine-dependent. Note that VAX/VMS Fortran does no padding. The **%FILL** feature is *not* functionally supported since explicit padding of a record is not necessary. However, the **%FILL** will be recognized and result in no action.

Data initialization can occur for the individual fields.

The **UNION** statement and **MAP** statement are supported.

See the *Paragon™ System Fortran Language Reference Manual* for details. The next section on Records provides an example.

Records

A record is a VAX/VMS Fortran extension that is an aggregate entity containing one or more record fields. Each field of a record can be named. This allows one to organize heterogenous data items within one structure and to operate on them individually or collectively.

The form of a record is defined with a "structure definition" block (**STRUCTURE** statement). The record is established in memory by specifying the name of the structure in a **RECORD** statement. The format of a **RECORD** statement is as follows:

```
RECORD /structure_name/record_namelist
      [,/structure_name/record_namelist]
                  ...
      [,/structure_name/record_namelist]
```

where *structure_name* is the name of a previously declared structure and *record_namelist* is a list of one or more variable or array names separated by commas. Records initially have undefined values unless their values have been defined in their corresponding structure declarations.

Individual fields of a record may be referenced by referring to the parent record name, a period (.), and finally the field name. A scalar reference is defined to mean a reference to a name that resolves to a single typed data item (e.g., **INTEGER**). An aggregate reference is defined to mean a reference that resolves to a structured data item.

As in VAX/VMS Fortran, scalar field references may appear wherever normal variable or array elements may appear with the exception of **COMMON**, **SAVE**, **NAMELIST**, **DATA** and **EQUIVALENCE** statements. Aggregate references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms.

VAX/VMS Fortran allows aggregates to be assigned as a whole entity. This type of **RECORD** assignment is fully supported.

RECORDS are fully supported except for the functional support of **%FILL**. See the *Paragon™ System Fortran Language Reference Manual* for a detailed description and use of the **RECORD** statements and structure declarations. The following is an example of **RECORD** and **STRUCTURE** usage.

```
STRUCTURE /person/      ! Declare a structure to define a person
   INTEGER id
   LOGICAL living
   CHARACTER*50 first, last, middle
   INTEGER age
END STRUCTURE
   ! Define population to be an array where each element is of
   ! type person.  Also define a variable, me, of type person.
RECORD /person/ population(1000), me
```

```
   ...
me.age = 34              ! Assign values for the variable me to
me.living = .TRUE.       ! some of the fields.
me.first = 'Steve'
me.id = 542124822
   ...
population(1).last = 'Jones'  ! Assign the "last" field of
                              ! element 1 of array population.
population(2) = me            ! Assign all the values of record
                              ! "me" to the record population(2)
```

UNION/MAP

A union declaration is a multistatement declaration defining a data area that can be shared intermittently during program execution by one or more fields or groups of fields. It declares groups of fields that share a common location within a structure. Each group of fields within a union declaration is declared by a map declaration, with one or more fields per map declaration.

Union declarations are used when one wants to use the same area of memory to alternately contain two or more groups of fields. Whenever one of the fields declared by a union declaration is referenced in a program, that field and any other fields in its map declaration become defined. Then, when a field in one of the other map declarations in the union declaration is referenced, the fields in that map declaration become defined, superseding the fields that were previously defined.

A union declaration is initiated by a **UNION** statement and terminated by an **END UNION** statement. Enclosed within these statements are two or more map declarations, initiated and terminated by **MAP** and **END MAP** statements, respectively. Each unique field or group of fields is defined by a separate map declaration. The format of a **UNION** statement is as follows:

```
UNION
   map_declaration
   [map_declaration]
         ...
   [map_declaration]
END UNION
```

where the format of the *map_declaration* is as follows:

```
MAP
   field_declaration
   [field_declaration]
         ...
   [field_declaration]
END MAP
```

where *field_declaration* is a structure declaration or **RECORD** statement contained within a union declaration, a union declaration contained within a union declaration, or the declaration of a typed data field within a union. Refer to the section "Structures" (on page E-11) and the *Paragon™ System Fortran Language Reference Manual* for more on field declarations.

Data can be initialized in field declaration statements in union declarations. Note, however, that if fields within multiple map declarations in a single union are initialized, the initialization of the overlapping data is undefined.

Field alignment within multiple map declarations are performed as previously defined in structure declarations.

The size of the shared area for a union declaration is the size of the largest map defined for that union. The size of a map is the sum of the sizes of the field(s) declared within it along with area reserved for alignment purposes.

Manipulating data using union declarations is similar to what happens using **EQUIVALENCE** statements. However, union declarations are probably more similar to union declarations for the language C. The main difference here is that the language C requires one to associate a name with each **MAP** and uses that name to differentiate multiple "maps" in a single union. The Fortran compiler's requirement that field names be unique within the same declaration nesting level eliminates the need for naming the **MAP**s.

The following is an example of **RECORD**, **STRUCTURE** and **UNION** usage. The size of each element of the *recarr* array would be the size of *typetag* (4 bytes) plus the size of the largest **MAP**—the *employee* map (24 bytes).

```
STRUCTURE /account/
   INTEGER typetag              ! Tag used to determine defined
map.
   UNION
      MAP                       ! Structure for an employee
         CHARACTER*12 ssn       ! Social Security Number
         REAL*4    salary
         CHARACTER*8 empdate    ! Employment date
      END MAP
      MAP                       ! Structure for a customer
         INTEGER*4 acct_cust
         REAL*4    credit_amt
         CHARACTER*8 due_date
      END MAP
      MAP                       ! Structure for a supplier
         INTEGER*4 acct_supp
         REAL*4    debit_amt
         BYTE      num_items
         BYTE      items(12)    ! Items supplied
      END MAP
   END UNION
END STRUCTURE

RECORD /account/ recarr(1000)
```

For more on **UNION**s and **MAP**s, see the *Paragon™ System Fortran Language Reference Manual.*

Exclusive OR

The Exclusive Or operator, **.XOR.**, is supported. See the *Paragon™ System Fortran Language Reference Manual* for details.

Format Extensions

A, O, Z, Q, and $ Field Descriptors

The O, Z, Q and $ field descriptors are new; the A edit descriptor is extended.

The A field descriptor is extended to process any data type. When not specified, the width is determined by the size of the data item. Note that the A field descriptor is the only repeatable edit descriptor whose field width specifier is optional as specified in Fortran 77.

The O field and Z field transfers octal or hexadecimal values and can be used with any data type. They have the form:

```
Ow[.m]  and  Zw[.m]
```

Where *w* specifies the field width and *m* indicates minimum field width on output.

On input, the external field to be input must contain (unsigned) octal or hexadecimal characters only. An all blank field is treated as a value of zero. If the value of the external field exceeds the range of the corresponding list element, an error occurs.

On output, the O and Z field descriptors transfers the octal and hexadecimal value of the corresponding I/O list element (respectively), right-justified, to an external field that is *w* characters long. If the value to be transmitted does not fill the field, leading spaces are inserted; if the value is too large for the field, the entire field is filled with asterisks. If *m* is present, the external field consists of at least *m* digits, and is zero-filled on the left if necessary. Note that if *m* is zero, and the internal representation is zero, the external field is blank-filled.

A typeless value output with list directed I/O is output in hexadecimal form by default. There is no other octal or hexadecimal capability with list directed I/O.

The Q edit descriptor calculates the number of characters remaining in the input record and stores that value in the next I/O list item. On output, the Q descriptor skips the next I/O item. See the *Paragon™ System Fortran Language Reference Manual* for details. It has the form:

```
Q
```

The $ descriptor allows the programmer to control carriage control conventions on output. It is ignored on input. For example, on terminal output, it can be used for prompting. See the *Paragon™ System Fortran Language Reference Manual* for more details. It has the form:

```
$
```

For F, E, and D output editing, the VAX/VMS Fortran output format is adhered to. See the *Paragon™ System Fortran Language Reference Manual* for details on output processing for F, E, and D field descriptors.

Carriage Control Characters

The I/O system recognizes characters as carriage controls when appearing as the first character of a record being written. In addition to the standard "1", " " (blank), "+", and "0"; the "$" and "\0" (ASCII NUL) are supported. The "$" allows for prompting by causing output to start at the beginning of the next line, and suppressing carriage return at the end of the line. The "\0" overprints with no advance; that is, it starts output at the beginning of the current line and does not return to the left margin after printing.

Note that a "$" appearing as the first character in a record to be written is interpreted as a carriage control character and is different from the "$" being used as an edit descriptor in a format statement. See the *Paragon™ System Fortran Language Reference Manual* for details.

Commas in External Fields

Use of the comma in an external field eliminates the need to "count spaces" to have data match format edit descriptors. The use of a comma to terminate an input field and thus avoid padding the field is fully supported, as described in the *Paragon™ System Fortran Language Reference Manual.*

Reading Non-Quoted Data into CHARACTER Variables

When reading string data from a formatted file into a **character*n** variable, the string need not be quoted. Characters are read until the **character*n** variable is full or until a linefeed is read, whichever comes first. See the *Paragon™ System Fortran Language Reference Manual* for more information.

Variable Format Expressions <expr>

Variable format expressions are supported in full. They provide a means for substituting runtime expressions for the field width and other parameters for the field and edit descriptors in a **FORMAT** statement (except for the H field descriptor). Variable format expressions are evaluated each time they are encountered in the scan of a format. If the value of a variable used in the expression changes during the execution of the I/O statement, the new value is used the next time the format item containing the expression is processed. Restrictions apply as indicated in the *Paragon™ System Fortran Language Reference Manual.*

Format Specification Separators

The ANSI Fortran 77 requirement that a format specification separator within a format statement be a comma or a slash (/) is relaxed. The comma may be eliminated as a format specifier whenever the end of the specifier and the beginning of the next specifier can be unambiguously determined. For example the following is legal:

```
format('1'7H1234567I6)
```

Commas were eliminated between the following format specifiers:

```
'1'
7H1234567
I6
```

ENCODE/DECODE Statements

The **ENCODE** and **DECODE** statements are unique to VAX/VMS Fortran and are fully supported. The **ENCODE** and **DECODE** statements transfer data between variables or arrays in internal storage and translate that data from internal to character form, and vice versa, according to format specifiers. Similar results can be accomplished using internal files with formatted sequential **WRITE** and **READ** statements. The **ENCODE** and **DECODE** statements have the form:

```
ENCODE (c,f,b[,IOSTAT=ios] [,ERR=s])[list]
DECODE (c,f,b[,IOSTAT=ios] [,ERR=s])[list]
```

where *c* is an integer expression specifying the number of bytes involved in translation, *f* is the format identifier, *b* is a scalar or array reference for the buffer area and *list* is the buffer area either containing data or receiving data. See the *Paragon™ System Fortran Language Reference Manual* for details and restrictions.

Lexical Extensions

Identifier Names

Identifiers may be arbitrarily long. The number of significant characters is 30. In addition to alphabetic and numeric characters, identifiers may contain the dollar sign ($) and the underscore (_). The first character of a name must be either alphabetic, the dollar sign, or an underscore. The default data type for identifiers beginning with "$" or "_" is **REAL**.

By default, all uppercase letters, except those in character or Hollerith constants are translated to lower case. As a result, keywords may be in either upper or lower case, and case is not significant in identifier names. This can be changed by use of the **-Mupcase** switch. When this switch is used, keywords must be in lower case and case is significant in identifier names.

Character Constants

For compatibility with C usage, the following backslash escapes are recognized within character string constants:

\v	vertical tab
\a	alert (bell)
\n	newline
\t	tab
\b	backspace
\f	formfeed
\r	carriage return
\0	null
\'	apostrophe (does not terminate a string)
\"	double quotes (does not terminate a string)
\\	\
\x	*x*, where *x* is any other character
ddd	character with the given octal representation.

Character string constants may be delimited using either an apostrophe (') or a double quote ("). If a string begins with one variety of quote mark, the other may be embedded within it without using the repeated quote (as in standard Fortran 77) or backslash escape.

Inline Comments

An exclamation point (!) can be used anywhere in the statement field (except when used in a Hollerith or character constant) to start an end-of-line comment.

Debug Statements

The letter "D" in column 1 designates the statement on that line to be a debugging statement. The compiler will treat the debugging statement as a comment unless the command line switch **-Mdlines** is used during the compilation. In that case, the compiler acts as if the "D" were a blank and compiles the line according to the standard rules.

INCLUDE Statements

The **INCLUDE** statement directs the compiler to start reading from another file. The format for the **INCLUDE** statement is:

```
INCLUDE 'pathname[/[NO]LIST]'
```

The **INCLUDE** statement may be nested to a depth of 20 and can appear anywhere within a program unit as long as the statement-ordering restrictions for Fortran statements are not violated. The directory search rules are as follows:

- If *pathname* is a fully qualified pathname, then that pathname specifies the directory to search.
- The current directory is searched.
- The directories specified via the **-I** switch on the compile line are searched in the order in which they occurred.

The qualifiers **/LIST** or **/NOLIST** can be used to control whether the include file is expanded in the listing file (if generated).

Note that there is no support for VAX/VMS "text libraries." Also note the lack of support for the "module_name" pathname qualifier that exists in the VAX/VMS version of the **INCLUDE** statement.

Statement Ordering

The rules defining the order in which statements appear in a program unit have been relaxed as follows:

- **DATA** statements can be freely interspersed with **PARAMETER** statements, other specification statements and executable statements.
- **NAMELIST** statements are supported and have the same order requirements as **FORMAT** and **ENTRY** statements.
- The **IMPLICIT NONE** statement can precede other **IMPLICIT** statements.

See the *Paragon™ System Fortran Language Reference Manual* for details on statement ordering.

Input File Format

Input source file format has been extended from Fortran 77 to allow a number of VAX/VMS Fortran extensions. Fortran 77 input source file format is supported in full as indicated in the ANSI standard.

- A continuation line may also be indicated by using an ampersand (&) in column one of a line.
- Tab-Format lines are supported. A tab in columns 1-6 ends the statement label field and begins an optional continuation indicator field. If a non-zero digit follows the tab character, the continuation field exists and indicates a continuation field. If anything other than a non-zero digit follows the tab character, the statement body begins with that character and extends to the end of the source statement. Note that this does not override Fortran 77's source line handling since no valid Fortran statement can begin with a non-zero digit. The tab character is ignored if it occurs anywhere else in a line except in Hollerith or character constants.

- Input lines may be of varying lengths. If there are fewer than seventy two characters, the line is padded with blanks; characters after the 72nd are ignored unless the **-Mextend** switch is used on the compile line.
- If the **-Mextend** switch is used on the command line then the input line can extend to 132 characters. The line is padded with blanks if it is fewer than 132; characters after the 132nd are ignored. Note that use of this switch extends the statement field to position 132.
- Blank lines are allowed at the end of a program unit.
- The number of continuation lines allowed is extended to 99.

VAX/VMS Fortran's Sequence Number Field support is not provided.

I/O Extensions

I/O statements are composed of three basic components: the *statement keyword*, the *control list*, and the *I/O list*. The statement keywords supported are **READ**, **ACCEPT**, **WRITE**, **TYPE**, and **PRINT**. **ACCEPT** and **TYPE** are VAX/VMS extensions. **ENCODE** and **DECODE** are supported as previously described. The control list and I/O list have a few extensions and are discussed in this section. The concept of namelist directed I/O is a VAX/VMS Fortran extension and is supported.

Namelist Directed I/O

The **NAMELIST** statement is fully supported. This feature allows for the definition of namelist groups for namelist directed I/O. See the *Paragon™ System Fortran Language Reference Manual* for details on namelist directed I/O.

ACCEPT and TYPE Statements

The **ACCEPT** statement has the same syntax as the **PRINT** statement and causes formatted input to be performed on *stdin*. It is identical to the **READ** statement with a unit specifier of asterisk (*). The **ACCEPT** statement is supported.

The **TYPE** statement has the same syntax and effect as the **PRINT** statement and is supported.

I/O Lists

Aggregate references can be used in unformatted input and output statements.

An extension that allows the programmer to freely parenthesize I/O items and groups of I/O items within I/O lists is supported as in VAX/VMS Fortran.

Control List Extensions

VAX/VMS allows a much larger set of I/O specifiers than Fortran 77. A subset of VAX/VMS extensions is supported by the Fortran compiler.

A namelist specifier is a parameter that specifies that namelist directed I/O is being used and identifies the group-name of the list of entities that may be modified on input or written on output. The namelist specifier has the form:

```
[NML=]group_name
```

where *group_name* is the name of a list previously defined in a **NAMELIST** statement.

The keyword **NML** is optional only if (1) the namelist specifier is the second parameter in the control list and (2) the first parameter is a logical unit specifier without an optional keyword **UNIT**. A namelist specifier cannot be used in a statement that contains a format specifier.

Extensions Derived from Cray Fortran

The Fortran compiler supports the following Cray extensions:

- Pointer-based variables (**POINTER** statement).
- Dynamic **COMMON** blocks (**ALLOCATABLE** attribute).
- Memory allocation statements (**ALLOCATE** and **DEALLOCATE** statements).

POINTER Statement

A *pointer* variable is an integer variable that contains the address of a corresponding *pointer-based* variable. The storage located by the pointer variable is viewed according to what's implied by the pointer-based variable (subscripted, data type, etc.). A reference to a pointer-based variable appears in **FORTRAN** statements like a normal variable reference (such as a local variable, common block variable, or dummy variable). When the pointer-based variable is referenced, the address to which it refers is always taken from its associated pointer (that is, its pointer variable is *dereferenced*).

The **POINTER** statement declares a scalar variable to be a *pointer* and another variable to be its *pointer-based* variable. The syntax of the **POINTER** statement is:

```
POINTER (p1, v1) [, (p2, v2) ...]
```

Where:

p1, p2, ... are the pointer variables; a pointer variable must have type **INTEGER** and must not be an array.

v1, v2, ... are the corresponding pointer-based variables; a pointer-based variable can be of any type, including structure.

A pointer-based variable can be dimensioned in the **POINTER** statement or in a separate type or **DIMENSION** statement. The dimension expression may be adjustable, where the rules for adjustable dummy arrays regarding any variables which appear in the dimension declarators apply.

The pointer-based variable does not have an address until its corresponding pointer is defined. A pointer can be defined by any of the following:

- Assigning the value of the **LOC** function to the pointer variable.
- Assigning a value defined in terms of another pointer variable to the pointer variable.
- Dynamically allocating a memory area for the pointer-based variable.

Also, if a pointer-based variable is dynamically allocated, it may also be freed (see "Memory Allocation Statements" on page 6-26).

For example:

```
REAL XC(10)
COMMON IC, XC
POINTER (P, I)
POINTER (Q, X(5))
P = LOC(IC)
I = 0            ! IC gets 0

P = LOC(XC)
Q =  P + 20      ! same as LOC(XC(6))
X(1) = 0         ! XC(6) gets 0

ALLOCATE (X)     ! Q locates a dynamically allocated
                 ! memory area
```

Restrictions:

- No storage is allocated when a pointer-based variable is declared.
- If a pointer-based variable is referenced, it's assumed that its pointer variable is defined.

- A pointer-based variable may not appear in the argument list of a **SUBROUTINE** or **FUNCTION** and may not appear in **COMMON, EQUIVALENCE, DATA, NAMELIST**, or **SAVE** statements.
- A pointer-based variable can only be adjustable in a **SUBROUTINE** or **FUNCTION** subprogram. If a pointer-based variable is an adjustable array, it's assumed that the variables in the dimension declarator(s) are defined with an integer value at the time the subroutine or function is called. For a variable which appears in a pointer-based variable's *adjustable* declarator, modifying its value during the execution of the subroutine or function does not modify the bounds of the dimensions of the pointer-based array.
- A pointer-based variable is assumed *not* to overlap with another pointer-based variable.

Dynamic COMMON

A *dynamic*, or *allocatable*, common block is a common block whose storage is not allocated until an explicit **ALLOCATE** statement is executed.

The syntax of the **COMMON** statement is extended to allow an attribute (**ALLOCATABLE**) after the **COMMON** keyword:

```
COMMON [, ALLOCATABLE] named_common_list
```

where *named_common_list* is the same form used to declare named (statically allocated) common blocks.

If the **ALLOCATABLE** attribute is present, all named common blocks appearing in the common statement are marked as *allocatable*. Like a normal **COMMON** statement, the name of an *allocatable* common block may appear in more than one **COMMON** statement. Note that the **ALLOCATABLE** attribute need not appear in every **COMMON** statement.

For example:

```
COMMON, ALLOCATABLE /all1/ a, b, /all2/aa, bb
COMMON /stat/d, /all1/ c
```

These statements declare *all1* and *all2* as *allocatable* common blocks whose members are *a*, *b*, *c*, and *aa*, *bb*, respectively, and *stat* as a *statically-allocated* common block (whose member is *d*).

A reference to a member of an *allocatable* common block appears in a **FORTRAN** statement just like a member of a normal (static) common block. No special syntax is required to access members of *allocatable* common blocks. For example, using the above declarations, the following statement is legal:

```
aa = b * d
```

Restrictions:

- Before members of an *allocatable* common block can be referenced, the common block must have been explicitly allocated using the **ALLOCATE** statement.
- Members of an *allocatable* common block cannot be data initialized.
- The memory used for an *allocatable* common block may be freed using the **DEALLOCATE** statement.
- If a subprogram declares a common block to be *allocatable*, all other subprograms containing **COMMON** statements of the same common block must also declare the common to be *allocatable*.

Memory Allocation Statements

The **ALLOCATE** and **DEALLOCATE** statements provide a mechanism to allocate and free memory during the execution of a program.

ALLOCATE Statement

The syntax of the **ALLOCATE** statement is:

```
ALLOCATE ( al [, al ] ... [, STAT=var ] )
```

Where:

al is a *pointer-based* variable or the name of an *allocatable* common enclosed in slashes.

var is an integer variable, integer array element, or an integer member of a structure.

The **ALLOCATE** attempts to allocate storage for each of the pointer-based variables and *allocatable* common blocks which appear in the statement. For a pointer-based variable, its associated pointer variable is defined with the address of the allocated memory area. If the **STAT=** specifier is present, successful execution of the **ALLOCATE** statement causes the status variable to become defined with the value 0 (zero). If an error occurs during the execution of the statement and the **STAT=** is present, the status variable is defined with the value 1 (one). If an error occurs and the **STAT=** specifier is not present, program execution is terminated.

DEALLOCATE Statement

The syntax of the **DEALLOCATE** statement is:

```
DEALLOCATE ( al [, al ] ... [ , STAT=var ] )
```

Where:

al	is a *pointer-based* variable or the name of an *allocatable* common enclosed in slashes.
var	is an integer variable, integer array element, or an integer member of a structure.

The **DEALLOCATE** statement causes the memory allocated for each of the pointer-based variables or *allocatable* common blocks which appear in the statement to be deallocated (freed). An attempt to deallocate a pointer-based variable or an *allocatable* common block which was not created by an **ALLOCATE** statement results in an error condition.

If the **STAT=** specifier is present, successful execution of the **DEALLOCATE** statement causes the status variable to become defined with the value 0 (zero). If an error occurs during the execution of the statement and the **STAT=** is present, the status variable is defined with the value 1 (one). If an error occurs and the **STAT=** specifier is not present, program execution is terminated.

Using Memory Allocation Statements

Here is an example of the **ALLOCATE** and **DEALLOCATE** statements:

```
COMMON P, N, M
POINTER (P, A(N,M))
COMMON, ALLOCATABLE /ALL/X(10), Y
ALLOCATE (/ALL/, A, STAT=IS)
PRINT *, IS
X(5) = A(2, 1)
DEALLOCATE (A)
DEALLOCATE (A, STAT=IS)
PRINT *, 'should be 1', IS
DEALLOCATE (/ALL/)
```

Other I/O Extensions

This section describes the following I/O extensions:

- How the underlying I/O is performed
- The different file organizations
- Extensions to the **OPEN** statement
- Extensions to the **CLOSE** statement
- Extensions to the **BACKSPACE** statement
- Extensions to the **READ/WRITE** statement

General Input/Output

The Fortran compiler supports the full I/O facilities of ANSI Standard Fortran 77 along with several extensions as listed in the following sections. A number of different file and access types are also supported. They include:

- Internal and external files
- Fixed and variable record length files
- Formatted and unformatted I/O
- Direct and sequential I/O

The Fortran 77 standard gives a certain amount of latitude to compiler implementors concerning default features of Fortran I/O. Unless specified otherwise, all file units opened are initially connected for sequential, formatted, variable length record, synchronous buffered I/O, and have a default value of `NULL` for the `BLANK` specifier in the **OPEN** statement. All *STATUS* and *LIMIT* variables must be of type **INTEGER*4**.

The operating system provides three standard data streams called *standard input*, *standard output*, and *standard error*. All of these streams are normally connected to the user's terminal, but this can be overridden by using I/O redirection on the command line (see the *OSF/1 User's Guide* for more information on I/O redirection). Standard input is preconnected to logical unit 5, standard output is preconnected to logical unit 6, and standard error is preconnected to logical unit 0. This means that you can read from unit 5 and write to units 6 and 0 without opening them first.

File Formats

Four kinds of external files are supported by the Fortran compiler: variable length and fixed length, formatted and unformatted records. To promote portability, files are implemented as ordinary UNIX files with their internal structure defined by the method used to write them.

A file is opened for *fixed length record* I/O by specifying the record length of the records using the **RECL** specifier of the **OPEN** statement. This is used by the I/O system to make the file look as if it is made up of records of the given length. The record length must be in units of bytes. For unformatted files, the size of the data items in a record must be added up (e.g., an **INTEGER*2** item requires two bytes). Each data item in an unformatted file immediately follows the previous item with no alignment requirements.

For *fixed length record formatted* files, each record consists of exactly the number of bytes that is specified by the user.

Each record of a *variable length unformatted* file is preceded and followed by a four-byte integer containing the record's length in bytes.

For *variable length record formatted input*, each newline character is interpreted as a record separator. On output, the I/O system writes a newline at the end of each record. If a program writes a newline itself, the single record containing the newline will appear as two records when read or backspaced over. The maximum allowed length of a record in a variable length record formatted file is 2000 characters.

Any file opened for direct access must be via fixed length records.

OPEN Statement

VAX/VMS Fortran introduces a number of extensions to the **OPEN** statement. Many of these relate only to the VMS file system and are not supported (e.g., `KEYED` access for indexed files). All of the standard Fortran 77 features for the **OPEN** statement are supported. The following keywords for the **OPEN** statement have been added or augmented as shown below. See the *Paragon™ System Fortran Language Reference Manual* for details on these keywords.

ACCESS The value of `'APPEND'` will be recognized and implies sequential access and positioning *after* the last record of the file. Opening a file with append access means that each appended record is written at the end of the file (i.e., the **lseek()** system call has no effect on the file pointer).

ASSOCIATEVARIABLE
This new keyword specifies an **INTEGER*4** variable which is updated to the next sequential record number after each direct access I/O operation. Only for direct access mode.

DISPOSE and **DISP**
: These new keywords specify the disposition for the file after it is closed. `'KEEP'` or `'SAVE'` is the default on anything other than **STATUS='SCRATCH'** files. `'DELETE'` indicates that the file is to be removed after it is closed. The `PRINT` and `SUBMIT` values are not supported.

NAME
: This new keyword is a synonym for **FILE**.

READONLY
: This new keyword specifies that an existing file can be read but prohibits writing to that file. The default is `read/write`.

RECL=*len*
: The record length given is interpreted as number of words in a record if the runtime environment parameter *FORTRANOPT* is set to `vaxio`. This is to ease porting of VAX/VMS programs. The default is that *len* is given in number of bytes in a record.

`TYPE`
: This new keyword is a synonym for `STATUS`.

CLOSE Statement

The **DISPOSE** or **DISP** keyword is added to the **CLOSE** statement. It is synonymous with the **STATUS** keyword. This new keyword specifies the existence of the file after closing. The default is `'SAVE'` or `'KEEP'`. The value of `'DELETE'` will delete the file after closing. These new values can be used with the **STATUS** keyword also.

BACKSPACE Statement

The **BACKSPACE** statement is supported as defined by Fortran 77. However, you must not issue a **BACKSPACE** statement for a file that is open for direct or append access.

READ/WRITE Statement

List directed reads or writes on internal files is supported in addition to the Fortran 77 standard of formatted reads or writes on internal files.

Namelist directed formatting is not permitted with internal reads or writes.

Subroutine and Intrinsic Extensions

This section describes added system built-in function and subroutine support and non-Fortran 77 intrinsic support. Table 6-5 (on page 6-37) summarizes the intrinsic functions.

Built-In Functions

The following built-in functions are fully supported:

- **%VAL**
- **%REF**
- **%DESCR**
- **%LOC**

See the *Paragon™ System Fortran Language Reference Manual* for details.

VAX/VMS System Subroutines

The following VAX/VMS Fortran system subroutines are provided.

DATE

Returns a nine-byte string containing the ASCII representation of the current date. It has the form:

```
CALL DATE(buf)
```

where *buf* is a nine-byte variable, array, array element, or character substring. The date is returned as a nine-byte ASCII character string of the form:

```
dd-mmm-yy
```

where:

dd	Is the two-digit day of the month
mmm	Is the three-character abbreviation of the month
yy	Is the last two digits of the year

IDATE

The **IDATE** subroutine returns three integer values representing the current month, day, and year. It has the form:

```
CALL IDATE(i, j, k)
```

If the current date were October 9, 1984, the values of the integer variables upon return would be:

i = 10
j = 9
k = 84

EXIT

The **EXIT** subroutine causes program termination, closes all open files, and returns control to the operating system. It has the form:

```
CALL EXIT[(exit_status)]
```

where *exit_status* is an optional integer argument used to specify the image exit value.

SECNDS

Provides system time of day, or elapsed time, as a floating point value in seconds. It has the form:

```
y = SECNDS(x)
```

where (**REAL** or **DOUBLE PRECISION**) *y* is set equal to the time in seconds since midnight, minus the user supplied value of the (**REAL** or **DOUBLE PRECISION**) *x*. Elapsed time computations can be performed with the following sequence of calls.

```
X = SECNDS(0.0)
    ...
    ...              ! Code to be timed
    ...
DELTA = SECNDS(X)
```

The accuracy of this call is the same as the resolution of the system clock.

TIME

Returns the current system time as an ASCII string. It has the form:

```
CALL TIME(buf)
```

where *buf* is an eight-byte variable, array, array element, or character substring. The **TIME** call returns the time as an eight-byte ASCII character string of the form:

```
hh:mm:ss
```

For example:

```
16:45:23
```

Note that a 24-hour clock is used.

MVBITS

The **MVBITS** subroutine transfers a bit field from one storage location (source) to a field in a second storage location (destination). **MVBITS** transfers *a3* bits from positions *a2* through (*a2* + *a3* - 1) of the source, *a1*, to positions *a5* through (*a5* + *a3* - 1) of the destination, *a4*. Other bits of the destination location remain unchanged. The values of (*a2* + *a3*) and (*a5* + *a3*) must be less than or equal to 32. It has the form:

```
CALL MVBITS(a1, a2, a3, a4, a5)
```

where:

a1 Is an integer variable or array element that represents the source location.

a2 Is an integer expression that identifies the first position in the field transferred from *a1*.

a3 Is an integer expression that identifies the length of the field transferred from *a1*.

a4 Is an integer variable or array element that represents the destination location.

a5 Is an integer expression that identifies the starting position within *a4*, for the bits being transferred.

RAN

Returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1. The result is a floating point number that is uniformly distributed in the range between 0.0 and 1.0 exclusive. It has the form:

```
y = RAN(i)
```

where *y* is set equal to the value associated by the function with the seed argument *i*. The argument *i* must be an **INTEGER*4** variable or **INTEGER*4** array element.

The argument *i* should initially be set to a large, odd integer value. The **RAN** function stores a value in the argument that it later uses to calculate the next random number.

There are no restrictions on the seed, although it should be initialized with different values on separate runs in order to obtain different random numbers. The seed is updated automatically, and **RAN** uses the following algorithm to update the seed passed as the parameter:

```
SEED = 69069 * SEED + 1 ! MOD 2**32
```

The value of `SEED` is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

VAX/VMS Intrinsics

Additional intrinsics are added to support the new data types of **INTEGER*2**, **LOGICAL*1**, **LOGICAL*2**, and **COMPLEX*16**. Typeless generic intrinsics (**AND**, **OR**, **NEQV**, **EQV**, **SHIFT**, and **COMPL**) are added to support 64-bit constant manipulations. VAX/VMS Fortran Bit-manipulation intrinsic functions are supported. The **REAL*16** intrinsic functions are not supported. Table 6-5 (on page 6-37) lists only supported VAX/VMS intrinsics *not* part of ANSI Fortran 77.

Support for trigonometric functions with arguments in degrees is provided as well as for arguments of type **COMPLEX*16**.

Table 6-3 lists the intrinsics that support the new data types.

Table 6-3. Intrinsics That Support The New Data Types

Intrinsic Function	Generic Name
Absolute Value	**ABS**
Truncation	**INT, IINT**
Nearest Integer	**ININT**
Conversion to **REAL*4**	**REAL**
Conversion to **REAL*8**	**DBLE**
Conversion to **COMPLEX*8**	**CMPLX**
Imaginary part of **COMPLEX*16**	(no generic name, specific names only)
Complex Conjugate	(no generic name, specific names only)
Maximum and Minimum	**MAX, MIN**
Positive Difference	**DIM**
Remainder	**MOD**
Transfer of Sign	**SIGN**
Trigonometrics	**SIN, COS**
Misc	**SQRT, LOG, EXP**

Table 6-4 lists other new intrinsics (generics).

Table 6-4. Other New Intrinsics (1 of 2)

Intrinsic Function	Generic Name
Trigometrics with degree arguments	**SIND, COSD, TAND, ASIND, ACOSD, ATAND, ATAN2D**
Zero-Extend Functions	**ZEXT, IZEXT**
Conversion to COMPLEX*16	**DCMPLX**
Bitwise AND	**IAND, AND**
Bitwise OR	**IOR, OR**
Bitwise Exclusive OR	**IEOR, XOR, NEQV**
Bitwise Exclusive NOR	**EQV**
Bitwise Complement	**NOT, COMPL**

Table 6-4. Other New Intrinsics (2 of 2)

Intrinsic Function	Generic Name
Bitwise Shift	**ISHFT, LSHIFT, RSHIFT, SHIFT**
Bitwise Circular Shift	**ISHFTC**
Bit Extract	**IBITS**
Bit Set	**IBSET**
Bit Test	**BTEST**
Bit Clear	**IBCLR**

Table 6-5 (on page 6-37) lists generic and intrinsic names for all intrinsics supported by the Fortran compiler that are *not* defined by ANSI Fortran 77. See the *Paragon™ System Fortran Language Reference Manual* for details on these new and extended intrinsic functions.

UNIX Related System Subroutines

The following subroutines are supplied to support C-like command line argument processing by Fortran routines.

GETARG

```
SUBROUTINE GETARG(N, ARG)
INTEGER*4 N
CHARACTER* (*) ARG
```

Returns *N*th command line argument in character variable *ARG*. For *N* equal to zero, the name of the program is returned.

IARGC

```
INTEGER*4 FUNCTION IARGC()
```

Returns the number of command line arguments following the program name.

Additional Intrinsic Functions

Table 6-5 lists intrinsic functions that are in addition to those described for Fortran 77. All the Fortran 77 intrinsics are supported and are detailed in the *American National Standard Programming Language FORTRAN, ANSI x3.9-1978*. See the *Paragon™ System Fortran Language Reference Manual* for details on the intrinsics in this chapter.

Table 6-5. Additional Intrinsic Functions (1 of 4)

Function	Number of Args	Generic Name	Specific Name	Type of Argument	Type of Result
Square Root	1	**SQRT**	**CDSQRT**	**COMPLEX*16**	**COMPLEX*16**
Natural Logarithm	1	**LOG**	**CDLOG**	**COMPLEX*16**	**COMPLEX*16**
Exponential	1	**EXP**	**CDEXP**	**COMPLEX*16**	**COMPLEX*16**
Sine (degree)	1	**SIND**	**SIND** **DSIND**	**REAL*4** **REAL*8**	**REAL*4** **REAL*8**
Cos (degree)	1	**COSD**	**COSD** **DCOSD**	**REAL*4** **REAL*8**	**REAL*4** **REAL*8**
Tan (degree)	1	**TAND**	**TAND** **DTAND**	**REAL*4** **REAL*8**	**REAL*4** **REAL*8**
ArcSine (degree)	1	**ASIND**	**ASIND** **DASIND**	**REAL*4** **REAL*8**	**REAL*4** **REAL*8**
ArcCosine (degree)	1	**ACOSD**	**ACOSD** **DACOSD**	**REAL*4** **REAL*8**	**REAL*4** **REAL*8**
ArcTangent (degree) Arc Tan a	1	**ATAND**	**ATAND** **DATAND**	**REAL*4** **REAL*8**	**REAL*4** **REAL*8**
ArcTangent (degree) Arc Tan a1/a2	2	**ATAN2D**	**ATAN2D** **DATAN2D**	**REAL*4** **REAL*8**	**REAL*4** **REAL*8**
Sine	1	**SIN**	**CDSIN**	**COMPLEX*16**	**COMPLEX*16**
Cos	1	**COS**	**CDCOS**	**COMPLEX*16**	**COMPLEX*16**
Absolute Value	1	**ABS**	**IIABS** **JIABS** **CDABS**	**INTEGER*2** **INTEGER*4** **COMPLEX*16**	**INTEGER*2** **INTEGER*4** **REAL*8**

Table 6-5. Additional Intrinsic Functions (2 of 4)

Function	Number of Args	Generic Name	Specific Name	Type of Argument	Type of Result
Truncation	1	**IINT**	**IINT**	**REAL*4**	**INTEGER*2**
			IIDINT	**REAL*8**	**INTEGER*2**
				COMPLEX*8	**INTEGER*2**
				COMPLEX*16	**INTEGER*2**
		INT		**COMPLEX*16**	**INTEGER*4**
		JINT	**JINT**	**REAL*4**	**INTEGER*4**
			JIDINT	**REAL*8**	**INTEGER*4**
				COMPLEX*8	**INTEGER*4**
				COMPLEX*16	**INTEGER*4**
Nearest Integer [a + .5*sign(a)]	1	**ININT**	**ININT**	**REAL*4**	**INTEGER*2**
			IIDNNT	**REAL*8**	**INTEGER*2**
		JNINT	**JNINT**	**REAL*4**	**INTEGER*4**
			JIDNNT	**REAL*8**	**INTEGER*4**
Zero-Extend Function (Conversion Routine)	1		**IZEXT**	**LOGICAL*1**	**INTEGER*2**
				LOGICAL*2	**INTEGER*2**
				INTEGER*2	**INTEGER*2**
		ZEXT	**JZEXT**	**LOGICAL*1**	**INTEGER*4**
				LOGICAL*2	**INTEGER*4**
				LOGICAL*4	**INTEGER*4**
				INTEGER*2	**INTEGER*4**
				INTEGER*4	**INTEGER*4**
Convert to **REAL*4**	1	**REAL**	**FLOATI**	**INTEGER*2**	**REAL*4**
			FLOATJ	**INTEGER*4**	**REAL*4**
				COMPLEX*16	**REAL*4**
Convert to **REAL*8**	1	**DBLE**	**DFLOTI**	**INTEGER*2**	**REAL*8**
			DFLOAT	`INTEGER*4`	`REAL*8`
			DFLOTJ	`INTEGER*4`	`REAL*8`
			DREAL	`COMPLEX*16`	`REAL*8`
Fix	1		**IIFIX**	**REAL*4**	**INTEGER*2**
			JIFIX	**REAL*4**	**INTEGER*4**
Conv to **COMPLEX*8** or **COMPLEX*8** from two arguments	1,2	**CMPLX**		**INTEGER*2**	**COMPLEX*8**
				COMPLEX*16	**COMPLEX*8**
Conv to **COMPLEX*16** or **COMPLEX*16** from two arguments	1,2	**DCMPLX**		**INTEGER*2**	**COMPLEX*16**
	1,2			**INTEGER*4**	**COMPLEX*16**
	1,2			**REAL*4**	**COMPLEX*16**
	1,2			**REAL*8**	**COMPLEX*16**
	1			**COMPLEX*8**	**COMPLEX*16**
	1			**COMPLEX*16**	**COMPLEX*16**

Table 6-5. Additional Intrinsic Functions (3 of 4)

Function	Number of Args	Generic Name	Specific Name	Type of Argument	Type of Result
Imag Part of Cmplx	1		**AIMAG** **DIMAG**	**COMPLEX*8** **COMPLEX*16**	**REAL*4** **REAL*8**
Complex Conjugate	1	**CONJG**	**DCONJG**	**COMPLEX*16**	**COMPLEX*16**
Maximum	n > 1	**MAX**	**IMAX0I** **MAX1** **AIMAX0** **JMAX0** **JMAX1** **AJMAX0**	**INTEGER*2** **REAL*4** **INTEGER*2** **INTEGER*4** **REAL*4** **INTEGER*4**	**INTEGER*2** **INTEGER*2** **REAL*4** **INTEGER*4** **INTEGER*4** **REAL*4**
Minimum	n > 1	**MIN**	**IMIN0** **IMIN1** **AIMIN0** **JMIN0** **JMIN1** **AJMIN0**	**INTEGER*2** **REAL*4** **INTEGER*2** **INTEGER*4** **REAL*4** **INTEGER*4**	**INTEGER*2** **INTEGER*2** **REAL*4** **INTEGER*4** **INTEGER*4** **REAL*4**
Positive Difference	2	**DIM**	**IIDIM** **JIDIM**	**INTEGER*2** **INTEGER*4**	**INTEGER*2** **INTEGER*4**
Remainder	2	**MOD**	**IMOD** **JMOD**	**INTEGER*2** **INTEGER*4**	**INTEGER*2** **INTEGER*4**
Transfer of Sign	2	**SIGN**	**IISIGN** **JISIGN**	**INTEGER*2** **INTEGER*4**	**INTEGER*2** **INTEGER*4**
Bitwise AND Performs a logical AND on bits	2	**IAND** **AND**	**IIAND** **JIAND**	**INTEGER*2** **INTEGER*4** See note 1	**INTEGER*2** **INTEGER*4** typeless
Bitwise OR Performs a logical OR on bits	2	**IOR** **OR**	**IIOR** **JIOR**	**INTEGER*2** **INTEGER*4** See note 1	**INTEGER*2** **INTEGER*4** typeless
Bitwise XOR Performs a logical Exclusive Or	2	**IEOR** **XOR** **NEQV**	**IIEOR** **JIEOR**	**INTEGER*2** **INTEGER*4** See note 1	**INTEGER*2** **INTEGER*4** typeless
Bitwise Excl. NOR Performs a logical Exclusive Nor	2	**EQV**		See note 1	typeless
Bitwise Complement Complements each bit	1	**NOT** **COMPL**	**INOT** **JNOT**	**INTEGER*2** **INTEGER*4** See note 1	**INTEGER*2** **INTEGER*4** typeless

Table 6-5. Additional Intrinsic Functions (4 of 4)

Function	Number of Args	Generic Name	Specific Name	Type of Argument	Type of Result
Address Extraction The address of a data item is returned. (Assumes 32-bit address)	1		**LOC**	**INTEGER*2** **INTEGER*4** **REAL*4** **REAL*8** **COMPLEX*8** **COMPLEX*16**	**INTEGER*4** **INTEGER*4** **INTEGER*4** **INTEGER*4** **INTEGER*4** **INTEGER*4**
Bitwise Shift a1 logically shifted left a2 bits. If a2 < 0 then right logical shift.	2	**ISHFT** **SHIFT**	**IISHFT** **JISHFT**	**INTEGER*2** **INTEGER*4** See note 2	**INTEGER*2** **INTEGER*4** typeless
Bitwise Left Shift a1 logically shifted left	2	**LSHIFT**		**INTEGER*2** **INTEGER*4**	**INTEGER*2** **INTEGER*4**
Bitwise Right Shift a1 logically shifted right	2	**RSHIFT**		**INTEGER*2** **INTEGER*4**	**INTEGER*2** **INTEGER*4**
Circular Shift Rightmost a3 bits of a1 are shifted circularly by a2 bits; remaining bits in a1 are unaffected.	3	**ISHFTC**	**IISHFTC** **JISHFTC**	**INTEGER*2** **INTEGER*4**	**INTEGER*2** **INTEGER*4**
Character Returns a character that has the ASCII value specified by the argument.	1	**CHAR**	 **CHAR**	**LOGICAL*1** **INTEGER*2** **INTEGER*4**	**CHARACTER** **CHARACTER** **CHARACTER**
Bit Extraction Extracts bits a2 through (a2+a3-1) from a1.	3	**IBITS**	**IIBITS** **JIBITS**	**INTEGER*2** **INTEGER*4**	**INTEGER*2** **INTEGER*4**
Set Bit Returns a1 with bit a2 set to 1.	2	**IBSET**	**IIBSET** **JIBSET**	**INTEGER*2** **INTEGER*4**	**INTEGER*2** **INTEGER*4**
Bit Test .TRUE. if bit a2 of a1 is a 1.	2	**BTEST**	**BITEST** **BJTEST**	**INTEGER*2** **INTEGER*4**	**LOGICAL*2** **LOGICAL*4**
Bit Clear Returns a1 with bit a2 set to 0.	2	**IBCLR**	**IIBCLR** **JIBCLR**	**INTEGER*2** **INTEGER*4**	**INTEGER*2** **INTEGER*4**

1. The arguments to the intrinsics **AND**, **OR**, **NEQV**, **EQV**, and **COMPL** may be of any type except **CHARACTER** and **COMPLEX**.
2. The first argument to the **SHIFT** intrinsic may be of any type except **CHARACTER** and **COMPLEX**. The second argument is any integer type.

Vector Intrinsics

Fortran provides a suite of *vector intrinsics*: subroutines that perform certain mathematical operations on vectors very efficiently. These vector intrinsics are available in both single-precision and double-precision forms. You can specify the number of vector elements and the strides of each input vector and the result vector.

NOTE

To use the vector intrinsics, you must link your program with the switch **-lvect**.

Table 6-6 on page 6-42 lists the vector intrinsics. The names of the arguments in Table 6-6 indicate their meaning and type:

n	The number of elements in the vectors *x*, *y*, and *z*. This argument is always of type **integer**.
x, *y*, *z*	The argument vectors. *x* and *y* are the input vectors (not all vector intrinsics have a *y* argument); z is the result vector. These arguments are vectors of type **double precision** for vector intrinsics whose names start with **dv_**, and of type **real** for vector intrinsics whose names start with **sv_**.
incx, *incy*, *incz*	The strides (increments) of vectors *x*, *y*, and *z*, respectively (may be zero). These arguments are always of type **integer**.
alpha	A scalar multiplier for **dv_recp** and **sv_recp**. This argument is of type **double precision** for **dv_recp**, and of type **real** for **sv_recp**.

For example, the following call to **sv_cos()** performs a single-precision vector cosine of the first *n* elements of the **real** vector *x* with stride *incx*, storing the results in the **real** vector *z* with stride *incz*:

```
call sv_cos(n, x, incx, z, incz)
```

It is similar in effect to the following code (the actual code for **sv_cos()** is written in assembler):

```
      ix = 1
      iz = 1
      if (incx .lt. 0) ix = (-n+1)*incx + 1
      if (incz .lt. 0) iz = (-n+1)*incz + 1
      do 10 i = 1, n
         z(iz) = cos(x(ix))
         ix = ix + incx
         iz = iz + incz
10    continue
```

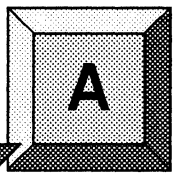
Table 6-6. Vector Intrinsic Functions (1 of 2)

Vector Intrinsic Function	Description
dv_acos(*n*, *x*, *incx*, *z*, *incz*) **sv_acos**(*n*, *x*, *incx*, *z*, *incz*)	Vector arccosine ($z(i)$ = **acos**($x(i)$))
dv_asin(*n*, *x*, *incx*, *z*, *incz*) **sv_asin**(*n*, *x*, *incx*, *z*, *incz*)	Vector arcsine ($z(i)$ = **asin**($x(i)$))
dv_atan(*n*, *x*, *incx*, *z*, *incz*) **sv_atan**(*n*, *x*, *incx*, *z*, *incz*)	Vector arctangent ($z(i)$ = **atan**($x(i)$))
dv_atan2(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*) **sv_atan2**(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)	Vector arctangent from two arguments ($z(i)$ = **atan2**($x(i)$, $y(i)$))
dv_cos(*n*, *x*, *incx*, *z*, *incz*) **sv_cos**(*n*, *x*, *incx*, *z*, *incz*)	Vector cosine ($z(i)$ = **cos**($x(i)$))
dv_div(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*) **sv_div**(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)	Non-IEEE vector divide ($z(i) = y(i)/x(i)$)
dv_exp(*n*, *x*, *incx*, *z*, *incz*) **sv_exp**(*n*, *x*, *incx*, *z*, *incz*)	Vector exponential ($z(i)$ = **exp**($x(i)$))
dv_log(*n*, *x*, *incx*, *z*, *incz*) **sv_log**(*n*, *x*, *incx*, *z*, *incz*)	Vector natural log ($z(i)$ = **log**($x(i)$))
dv_log10(*n*, *x*, *incx*, *z*, *incz*) **sv_log10**(*n*, *x*, *incx*, *z*, *incz*)	Vector logarithm $\log_{10}$ ($z(i)$ = **log10**($x(i)$))
dv_pow(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*) **sv_pow**(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)	Vector power ($z(i) = x(i)^{y(i)}$)
dv_recp(*n*, *alpha*, *x*, *incx*, *z*, *incz*) **sv_recp**(*n*, *alpha*, *x*, *incx*, *z*, *incz*)	Non-IEEE reciprocal times a scalar ($z(i) = alpha/x(i)$)
dv_rsqrt(*n*, *x*, *incx*, *z*, *incz*) **sv_rsqrt**(*n*, *x*, *incx*, *z*, *incz*)	Non-IEEE vector reciprocal square root ($z(i)$ = 1/**sqrt**($x(i)$))

Table 6-6. Vector Intrinsic Functions (2 of 2)

Vector Intrinsic Function	Description
dv_sin(*n*, *x*, *incx*, *z*, *incz*) **sv_sin**(*n*, *x*, *incx*, *z*, *incz*)	Vector sine (*z*(*i*) = **sin**(*x*(*i*)))
dv_sqrt(*n*, *x*, *incx*, *z*, *incz*) **sv_sqrt**(*n*, *x*, *incx*, *z*, *incz*)	Non-IEEE vector square root (*z*(*i*) = **sqrt**(*x*(*i*)))
dv_tan(*n*, *x*, *incx*, *z*, *incz*) **sv_tan**(*n*, *x*, *incx*, *z*, *incz*)	Vector tangent (*z*(*i*) = **tan**(*x*(*i*)))

Compiler Error Messages



This appendix lists the error messages generated by the Paragon™ system Fortran compiler, indicating each message's severity and, where appropriate, the error's probable cause and correction. In the error messages, the dollar sign ($) represents information that is specific to each occurrence of the message.

Each error message is numbered and preceded by one of the following letters, indicating its severity:

I	Informative
W	Warning
S	Severe error
F	Fatal error
V	Variable

`V000  Internal compiler error. $ $`

This message indicates an error in the compiler. The severity may vary; if it is informative or warning, the compiler probably generated correct object code, but there is no way to be sure. Regardless of the severity, please report any internal error to Intel Supercomputer Systems Division Customer Support.

`F001  Source input file name not specified`

On the command line, the source file name should be specified either before all the switches, or after them.

`F002 Unable to open source input file: $`

Source file name misspelled, file not in current working directory, or file is read protected.

`F003 Unable to open listing file`

Probably, user does not have write permission for the current working directory.

`F004 Unable to open object file`

Probably, user does not have write permission for the current working directory.

`F005 Unable to open temporary file`

Compiler uses directory */usr/tmp* or */tmp* in which to create temporary files. If neither of these directories is available on the node on which the compiler is being used, this error will occur.

`S006 Input file empty`

Source input file does not contain any Fortran statements other than comments or compiler directives.

`F007 Subprogram $ too large to compile at this optimization level`

Internal compiler data structure overflow, working storage exhausted, or some other non-recoverable problem related to the size of the subprogram. If this error occurs at optimization level 2, reducing the optimization level to 1 may work around the problem. Moving the subprogram being compiled to its own source file may eliminate the problem. If this error occurs while compiling a subprogram of fewer than 2000 statements it should be reported as a possible compiler problem.

`F008 Error limit exceeded`

The compiler gives up after 50 severe errors.

`F009 Unable to open assembly file`

Probably, user does not have write permission for the current working directory.

`F010 <reserved message number>`

`S011 Unrecognized command line switch: $`

Refer to the **if77** manual page for a list of the allowed compiler switches.

S012 Value required for command line switch: $

Certain switches require an immediately following value, such as **-O 2**.

S013 Unrecognized value specified for command line switch: $

S014 Ambiguous command line switch: $

Too short an abbreviation was used for one of the switches.

W015 Hexadecimal or octal constant truncated to fit data type

I016 Identifier, $, truncated to 31 chars

An identifier may be at most 31 characters in length; characters after the 31st are ignored.

S017 Unable to open include file: $

File is missing, read protected, or maximum include depth (10) exceeded. Remember that the file name should be enclosed in quotes.

S018 Illegal label field

The label field (first five characters) of the indicated line contains a non-numeric character.

S019 Illegally placed continuation line

A continuation line does not follow an initial line, or more than 99 continuation lines were specified.

S020 Unrecognized compiler directive

Refer to the **if77** manual page for a list of the allowed compiler directives.

S021 Label field of continuation line is not blank

The first five characters of a continuation line must be blank.

S022 Unexpected end of file - missing END statement

S023 Syntax error - unbalanced parentheses

W024 CHARACTER or Hollerith constant truncated to fit data type

A character or Hollerith constant was converted to a data type that was not large enough to contain all of the characters in the constant. This type conversion occurs when the constant is used in an arithmetic expression or is assigned to a non-character variable. The character or Hollerith constant is truncated on the right, that is, if 4 characters are needed then the first 4 are used and the remaining characters are discarded.

W025 Illegal character ($) - ignored

The current line contains a character, possibly nonprinting, which is not a legal Fortran character (characters inside of character or Hollerith constants cannot cause this error). As a general rule, all non-printing characters are treated as white space characters (blanks and tabs); no error message is generated when this occurs. If for some reason, a non-printing character is not treated as a white space character, its hex representation is printed in the form *dd* where each *d* is a hex digit.

S026 Unmatched quote

S027 Illegal integer constant: $

Integer constant is too large for 32-bit word.

S028 Illegal real or double precision constant: $

S029 Illegal hexadecimal constant: $

A hexadecimal constant consists of digits 0–9 and letters A–F or a–f. Any other character in a hexadecimal constant is illegal.

S030 Illegal octal constant: $

An octal constant consists of digits 0–7. Any other digit or character in an octal constant is illegal.

S031 Illegal data type length specifier for $

The data type length specifier (e.g. 4 in **INTEGER*4**) is not a constant expression that is a member of the set of allowed values for this particular data type.

W032 Data type length specifier not allowed for $

The data type length specifier (e.g. 4 in **INTEGER*4**) is not allowed in the given syntax (e.g. **DIMENSION A(10)*4**).

```
S033  Illegal use of constant $
```

A constant was used in an illegal context, such as on the left side of an assignment statement or as the target of a data initialization statement.

```
S034  Syntax error at or near $
```

```
I035  Predefined intrinsic $ loses intrinsic property
```

An intrinsic name was used in a manner inconsistent with the language definition for that intrinsic. The compiler, based on the context, will treat the name as a variable or an external function.

```
S036  Illegal implicit character range
```

First character must alphabetically precede second.

```
S037  Contradictory data type specified for $
```

The indicated identifier appears in more than one type specification statement and different data types are specified for it.

```
S038  Symbol, $, has not been explicitly declared
```

The indicated identifier must be declared in a type statement; this is required when the **IMPLICIT NONE** statement occurs in the subprogram.

```
W039  Symbol, $, appears illegally in a SAVE statement
```

An identifier appearing in a **SAVE** statement must be a local variable or array.

```
S040  Illegal common variable $
```

Indicated identifier is a dummy variable, is already in a common block, or has previously been defined to be something other than a variable or array.

```
W041  Illegal use of dummy argument $
```

This error can occur in several situations. It can occur if dummy arguments were specified on a **PROGRAM** statement. It can also occur if a dummy argument name occurs in a **DATA**, **COMMON**, **SAVE**, or **EQUIVALENCE** statement. A **PROGRAM** statement must have an empty argument list.

S042 $ is a duplicate dummy argument

S043 Illegal attempt to redefine $ $

An attempt was made to define a symbol in a manner inconsistent with an earlier definition of the same symbol. This can happen for a number of reasons. The message attempts to indicate the situation that occurred:

`intrinsic` An attempt was made to redefine an intrinsic function. A symbol that represents an intrinsic function may be redefined if that symbol has not been previously verified to be an intrinsic function. For example, the intrinsic **sin** can be defined to be an integer array. If a symbol is verified to be an intrinsic function via the **INTRINSIC** statement or via an intrinsic function reference then it must be referred to as an intrinsic function for the remainder of the program unit.

`symbol` An attempt was made to redefine a symbol that was previously defined. An example of this is to declare a symbol to be a **PARAMETER** which was previously declared to be a subprogram argument.

S044 Multiple declaration for symbol $

A redundant declaration of a symbol has occurred. For example, an attempt was made to declare a symbol as an **ENTRY** when that symbol was previously declared as an **ENTRY**.

S045 Data type of entry point $ disagrees with function $

The current function has entry points with data types inconsistent with the data type of the current function. For example, the function returns type **character** and an entry point returns type **complex**.

S046 Data type length specifier in wrong position

The **CHARACTER** data type specifier has a different position for the length specifier from the other data types. Suppose, we want to declare arrays *ARRAYA* and *ARRAYB* to have 8 elements each having an element length of 4 bytes. The difference is that *ARRAYA* is **character** and *ARRAYB* is **integer**. The declarations would be **CHARACTER ARRAYA(8)*4** and **INTEGER ARRAYB*4(8)**.

S047 More than seven dimensions specified for array

S048 Illegal use of '*' in declaration of array $

An asterisk may be used only as the upper bound of the last dimension.

```
S049  Illegal use of '*' in non-subroutine subprogram
```

The alternate return specifier * is legal only in the **subroutine** statement. Programs, functions, and block data are not allowed to have alternate return specifiers.

```
S050  Adjustable or assumed size array, $, is not a dummy argument
```

```
S051  Unrecognized built-in % function
```

The allowable built-in functions are **%VAL**, **%REF**, **%LOC**, and **%FILL**. One was encountered that did not match one of these allowed forms.

```
S052  Illegal argument to %VAL or %LOC
```

```
S053  %REF or %VAL not legal in this context
```

The built-in functions **%REF** and **%VAL** can only be used as actual parameters in procedure calls.

```
W054  Implicit character $ used in a previous implicit statement
```

An implicit character has been given an implied data type more than once. The implied data type for the implicit character is changed anyway.

```
W055  Multiple implicit none statements
```

The **IMPLICIT NONE** statement can occur only once in a subprogram.

```
W056  Implicit type declaration
```

The **-Mdclchk** switch and an implicit declaration following an **IMPLICIT NONE** statement will produce a warning message for **IMPLICIT** statements.

```
S057  Illegal equivalence of dummy variable, $
```

Dummy arguments may not appear in **EQUIVALENCE** statements.

```
S058  Equivalenced variables $ and $ not in same common block
```

A common block variable must not be equivalenced with a variable in another common block.

```
S059  Conflicting equivalence between $ and $
```

The indicated equivalence implies a storage layout inconsistent with other equivalences.

S060 Illegal equivalence of structure variable, $

STRUCTURE and **UNION** variables may not appear in **EQUIVALENCE** statements.

S061 Equivalence of $ and $ extends common block backwards

W062 Equivalence forces $ to be unaligned

EQUIVALENCE statements have defined an address for the variable which has an alignment not optimal for variables of its data type. This can occur when **INTEGER** and **CHARACTER** data are equivalenced, for instance.

I063 Gap in common block $ before $

S064 Illegal use of $ in DATA statement implied DO loop

The indicated variable is referenced where it is not an active implied **DO** index variable.

S065 Repeat factor less than or equal to zero

S066 Too few data constants in initialization statement

S067 Too many data constants in initialization statement

S068 Numeric initializer for CHARACTER $ out of range 0 through 255

A **CHARACTER*1** variable or character array element can be initialized to an integer, octal, or hexadecimal constant if that constant is in the range 0 through 255.

S069 Illegal implied DO expression

The only operations allowed within an implied **DO** expression are integer **+**, **-**, ***, and **/**.

S070 Incorrect sequence of statements $

The statement order is incorrect. For instance, an **IMPLICIT NONE** statement must precede a specification statement which in turn must precede an executable statement.

S071 Executable statements not allowed in block data

S072 Assignment operation illegal to $ $

The destination of an assignment operation must be a variable, array reference, or vector reference. The assignment operation may be by way of an assignment statement, a data statement, or the index variable of an implied **DO**-loop. The compiler has determined that the identifier used as the destination, is not a storage location. The error message attempts to indicate the type of entity used:

entry point	An assignment to an entry point that was not a function procedure was attempted.
external procedure	An assignment to an external procedure or a Fortran intrinsic name was attempted.

S073 Intrinsic or predeclared, $, cannot be passed as an argument

S074 Illegal number or type of arguments to $

The indicated symbol is an intrinsic or generic function, or a predeclared subroutine or function, requiring a certain number of arguments of a fixed data type.

S075 Subscript, substring, or argument illegal in this context for $

This can happen if you try to doubly index an array such as **ra(2)(3)**. This also applies to substring and function references.

S076 Subscripts specified for non-array variable $

S077 Subscripts omitted from array $

S078 Wrong number of subscripts specified for $

S079 Keyword form of intrinsic argument illegal in this context for $

S080 Subscript for array $ is out of bounds

S081 Matrix/vector $ illegal as subprogram argument

A matrix/vector reference cannot be used as a subprogram argument.

S082 Illegal substring expression for variable $

Substring expressions must be of type **integer** and if constant must be greater than zero.

S083 Vector expression used where scalar expression required

A vector expression was used in an illegal context. For example, **iscalar = iarray**, where a scalar is assigned the value of an array. Also, character and record references are not vectorizable.

S084 Illegal use of symbol $ $

This message is used for many different errors.

S085 Incorrect number of arguments to statement function $

S086 Dummy argument to statement function must be a variable

S087 Non-constant expression where constant expression required

S088 Recursive subroutine or function call of $

A function may not call itself.

S089 Illegal use of symbol, $, with character length = *

Symbols of type **CHARACTER*(*)** must be dummy variables and must not be used as statement function dummy parameters and statement function names. Also, a dummy variable of type **CHARACTER*(*)** cannot be used as a function.

S090 Hollerith constant more than 4 characters

In certain contexts, Hollerith constants may not be more than 4 characters long.

S091 Constant expression of wrong data type

S092 Illegal use of variable length character expression

A character expression used as an actual argument, or in certain contexts within I/O statements, must not consist of a concatenation involving a passed length character variable.

W093 Type conversion of expression performed

An expression of some data type appears in a context which requires an expression of some other data type. The compiler generates code to convert the expression into the required type.

S094 Variable $ is of wrong data type $

The indicated variable is used in a context which requires a variable of some other data type.

S095 Expression has wrong data type

An expression of some data type appears in a context which requires an expression of some other data type.

S096 Illegal complex comparison

The relations **.LT.**, **.GT.**, **.GE.**, and **.LE.** are not allowed for **complex** values.

S097 Statement label $ has been defined more than once

More than one statement with the indicated statement number occurs in the subprogram.

S098 Divide by zero

S099 Illegal use of an aggregate RECORD

Aggregate record references may only appear in aggregate assignment statements, unformatted I/O statements, and as parameters to subprograms. They may not appear, for example, in expressions. Also, records with differing structure types may not be assigned to one another.

S100 Expression cannot be promoted to a vector

An expression was used that required a scalar quantity to be promoted to a vector illegally. For example, the assignment of a character constant string to a character array. Records, too, cannot be promoted to vectors.

S101 Vector operation not allowed on $

Record and character typed entities may only be referenced as scalar quantities.

S102 Arithmetic IF expression has wrong data type

The parenthetical expression of an arithmetic if statement must be an integer, real, or double precision scalar expression.

S103 Type conversion of subscript expression for $

The data type of a subscript expression must be integer. If it is not, it is converted.

S104 Illegal control structure $

This message is issued for a number of errors involving **IF–THEN** statements and **DO** loops. If the line number specified is the last line (**END** statement) of the subprogram, the error is probably an unterminated **DO** loop or **IF–THEN** statement.

S105 Unmatched ELSEIF, ELSE or ENDIF statement

An **ELSEIF**, **ELSE**, or **ENDIF** statement cannot be matched with a preceding **IF–THEN** statement.

S106 DO index variable must be a scalar variable

The **DO** index variable cannot be an array name, a subscripted variable, a **PARAMETER** name, a function name, a structure name, etc.

S107 Illegal assigned goto variable $

S108 Illegal variable, $, in NAMELIST group $

A **NAMELIST** group can only consist of arrays and scalars which are not dummy arguments and pointer-based variables.

I109 Overflow in hexadecimal constant $, constant truncated at left

A hexadecimal constant requiring more than 64 bits produces an overflow. The hexadecimal constant is truncated at left (e.g. **'1234567890abcdef1'x** becomes **'234567890abcdef1'**x).

I110 Overflow in octal constant $, constant truncated at left

An octal constant requiring more than 64 bits produces an overflow. The octal constant is truncated at left (e.g. **'2777777777777777777777'o** becomes **'7777777777777777777777'o**).

```
I111  Underflow of real or double precision constant

I112  Overflow of real or double precision constant

S113  Label $ is referenced but never defined

S114  <reserved message number>

S115  <reserved message number>

S116  Illegal use of pointer-based variable $ $

S117  Statement not allowed within STRUCTURE definition

S118  Statement not allowed in DO, IF, or WHERE block

I119  Redundant specification for $
```

Data type of indicated symbol specified more than once.

```
I120  Label $ is defined but never referenced

I121  Operation requires logical or integer data types
```

An operation in an expression was attempted on data having a data type incompatible with the operation. For example, a logical expression can consist of only logical elements of type **integer** or **logical**. Real data would be invalid.

```
I122  Character string truncated
```

Character string or Hollerith constant appearing in a **DATA** statement or **PARAMETER** statement has been truncated to fit the declared size of the corresponding identifier.

```
W123  Hollerith length specification too big, reduced
```

The length specifier field of a Hollerith constant specified more characters than were present in the character field of the hollerith constant. The length specifier was reduced to agree with the number of characters present.

S124 Relational expression mixes character with numeric data

A relational expression is used to compare two arithmetic expressions or two character expressions. A character expression cannot be compared to an arithmetic expression.

I125 Dummy procedure $ not declared EXTERNAL

A dummy argument which is not declared in an **EXTERNAL** statement is used as the subprogram name in a **CALL** statement, or is called as a function, and is therefore assumed to be a dummy procedure. This message can result from a failure to declare a dummy array.

I126 Name $ is not an intrinsic function

I127 Optimization level for $ changed to opt 1 $

W128 Integer constant truncated to fit data type: $

An integer constant will be truncated when assigned to data types smaller than 32 bits, such as a **BYTE**.

I129 Floating point overflow. Check constants and constant expressions

I130 Floating point underflow. Check constants and constant expressions

I131 Integer overflow. Check floating point expressions cast to integer

I132 Floating pt. invalid oprnd. Check constants and constant expressions

I133 Divide by 0.0. Check constants and constant expressions

W134 <reserved message number>

W135 Missing STRUCTURE name field

A **STRUCTURE** name field is required on the outermost structure.

W136 Field-namelist not allowed

The field-namelist field of the **STRUCTURE** statement is disallowed on the outermost structure.

W137 Field-namelist is required in nested structures

W138 Multiply defined STRUCTURE member name $

A member name was used more than once within a structure.

W139 Structure $ in RECORD statement not defined

A **RECORD** statement contains a reference to a **STRUCTURE** that has not yet been defined.

S140 Variable $ is not a RECORD

S141 RECORD required on left of .

S142 $ is not a member of this RECORD

W143 <reserved message number>

W144 NEED ERROR MESSAGE $ $

This is used as a temporary message for compiler development.

W145 %FILL only valid within STRUCTURE block

The **%FILL** special name was used outside of a **STRUCTURE** multiline statement. It is only valid when used within a **STRUCTURE** multiline statement even though it is ignored.

S146 Expression must be character type

S147 Character expression not allowed in this context

S148 Non-record where aggregate record reference required

An aggregate reference to a record was expected during statement compilation but another data type was found instead.

S149 Record where arithmetic value required

An aggregate record reference was encountered when an arithmetic expression was expected.

S150 Structure, Record, or member $ not allowed in this context

A structure, record, or member reference was found in a context which is not supported. For example, the use of structures, records, or members within a data statement is disallowed.

S151 Empty STRUCTURE, UNION, or MAP

A **STRUCTURE–ENDSTRUCTURE**, **UNION–ENDUNION**, or **MAP–ENDMAP** declaration contains no members.

S152 <reserved message number>

S153 <reserved message number>

S154 <reserved message number>

S155 <reserved message number>

S156 <reserved message number>

S157 <reserved message number>

S158 Alternate return not specified in SUBROUTINE or ENTRY

An alternate return can only be used if alternate return specifiers appeared in the **SUBROUTINE** or **ENTRY** statements.

S159 Alternate return illegal in FUNCTION subprogram

An alternate return cannot be used in a **FUNCTION**.

S160 ENDSTRUCTURE, ENDUNION, or ENDMAP does not match top

S161 <reserved message number>

W162 Not equal test of loop control variable $ replaced with < or > test.

S163 Cannot data initialize member $ of the ALLOCATABLE COMMON $

S164 Overlapping data initializations of $

An attempt was made to data initialize a variable or array element already initialized.

S165 $ appeared more than once as a subprogram

A subprogram name appeared more than once in the source file. The message is applicable only when an assembly file is the output of the compiler.

S166 $ cannot be a common block and a subprogram

A name appeared as a common block name and a subprogram name. The message is applicable only when an assembly file is the output of the compiler.

I167 Inconsistent size of common block $

A common block occurs in more than one subprogram of a source file and its size is not identical. The maximum size is chosen. The message is applicable only when an assembly file is the output of the compiler.

S168 Incompatible size of common block $

A common block occurs in more than one subprogram of a source file and is initialized in one subprogram. Its initialized size was found to be less than its size in the other subprogram(s). The message is applicable only when an assembly file is the output of the compiler.

W169 Multiple data initializations of common block $

A common block is initialized in more than one subprogram of a source file. Only the first set of initializations apply. The message is applicable only when an assembly file is the output of the compiler.

W170 F77 extension: $

Use of a nonstandard feature. A description of the feature is provided.

W171 F77 extension: nonstandard statement type $

W172 F77 extension: numeric initialization of CHARACTER $

A **CHARACTER*1** variable or array element was initialized with a numeric value.

```
W173  F77 extension: nonstandard use of data type length specifier

W174  F77 extension: type declaration contains data initialization

W175  F77 extension: IMPLICIT range contains nonalpha characters

W176  F77 extension: nonstandard operator $

W177  F77 extension: nonstandard use of keyword argument $

W178  F77 extension: matrix/vector reference $

W179  F77 extension: use of structure field reference $

W180  F77 extension: nonstandard form of constant

W181  F77 extension: & alternate return

W182  F77 extension: mixed numeric and CHARACTER elements in COMMON $

W183  F77 extension: mixed numeric and CHARACTER EQUIVALENCE ($,$)

S197  Invalid qualifier or qualifier value (/$) in OPTIONS statement
```

An illegal qualifier was found or a value was specified for a qualifier which does not expect a value. In either case, the qualifier for which the error occurred is indicated in the error message.

```
S198  $ $ in ALLOCATE/DEALLOCATE

W199  Unaligned memory reference
```

A memory reference occurred whose address does not meet its data alignment requirement.

```
S200  Missing UNIT/FILE specifier

S201  Illegal I/O specifier - $

S202  Repeated I/O specifier - $
```

S203 FORMAT statement has no label

S204 Syntax error - unbalanced angle brackets

S205 Illegal specification of scale factor

The integer following + or - has been omitted, or **P** does not follow the integer value.

S206 Repeat count is zero

S207 Integer constant expected in edit descriptor

S208 Period expected in edit descriptor

S209 Illegal edit descriptor

S210 Exponent width not used in the Ew.dEe or Gw.dEe edit descriptors

S211 Internal I/O not allowed in this I/O statement

S212 Illegal NAMELIST I/O

Namelist I/O cannot be performed with internal, unformatted, formatted, and list-directed I/O. Also, I/O lists must not be present.

S213 $ is not a NAMELIST group name

S214 Input item is not a variable reference

S215 Assumed sized array name cannot be used as an I/O item or specifier

An assumed sized array was used as an item to be read or written or as an I/O specifier (i.e., **FMT = array_name**). In these contexts the size of the array must be known.

S216 STRUCTURE/UNION cannot be used as an I/O item

S217 ENCODE/DECODE buffer must be a variable, array, or array element

S221 `#elif after #else`

A preprocessor **#elif** directive was found after a **#else** directive; only **#endif** is allowed in this context.

S222 `#else after #else`

A preprocessor **#else** directive was found after a **#else** directive; only **#endif** is allowed in this context.

S223 `#if-directives too deeply nested`

Preprocessor **#if** directive nesting exceeded the maximum allowed (currently 10).

S224 `Actual parameters too long for $`

The total length of the parameters in a macro call to the indicated macro exceeded the maximum allowed (currently 2048).

W225 `Argument mismatch for $`

The number of arguments supplied in the call to the indicated macro did not agree with the number of parameters in the macro's definition.

F226 `Can't find include file $`

The indicated include file could not be opened.

S227 `Definition too long for $`

The length of the macro definition of the indicated macro exceeded the maximum allowed (currently 2048).

S228 `EOF in comment`

The end of a file was encountered while processing a comment.

S229 `EOF in macro call to $`

The end of a file was encountered while processing a call to the indicated macro.

S230 EOF in string

The end of a file was encountered while processing a quoted string.

S231 Formal parameters too long for $

The total length of the parameters in the definition of the indicated macro exceeded the maximum allowed (currently 2048).

S232 Identifier too long

The length of an identifier exceeded the maximum allowed (currently 2048).

S233 <reserved message number>

W234 Illegal directive name

The sequence of characters following a # sign was not an identifier.

W235 Illegal macro name

A macro name was not an identifier.

S236 Illegal number $

The indicated number contained a syntax error.

F237 Line too long

The input source line length exceeded the maximum allowed (currently 2048).

W238 Missing #endif

End of file was encountered before a required **#endif** directive was found.

W239 Missing argument list for $

A call of the indicated macro had no argument list.

S240 Number too long

The length of a number exceeded the maximum allowed (currently 2048).

W241 `Redefinition of symbol $`

The indicated macro name was redefined.

I242 `Redundant definition for symbol $`

A definition for the indicated macro name was found that was the same as a previous definition.

F243 `String too long`

The length of a quoted string exceeded the maximum allowed (currently 2048).

S244 `Syntax error in #define, formal $ not identifier`

A formal parameter that was not an identifier was used in a macro definition.

W245 `Syntax error in #define, missing blank after name or arglist`

There was no space or tab between a macro name or argument list and the macro's definition.

S246 `Syntax error in #if`

A syntax error was found while parsing the expression following a **#if** or **#elif** directive.

S247 `Syntax error in #include`

The **#include** directive was not correctly formed.

W248 `Syntax error in #line`

A **#line** directive was not correctly formed.

W249 `Syntax error in #module`

A **#module** directive was not correctly formed.

W250 `Syntax error in #undef`

A **#undef** directive was not correctly formed.

W251 `Token after #ifdef must be identifier`

The **#ifdef** directive was not followed by an identifier.

W252 Token after #ifndef must be identifier

The **#ifndef** directive was not followed by an identifier.

S253 Too many actual parameters to $

The number of actual arguments to the indicated macro exceeded the maximum allowed (currently 31).

S254 Too many formal parameters to $

The number of formal arguments to the indicated macro exceeded the maximum allowed (currently 31).

F255 Too much pushback

The preprocessor ran out of space while processing a macro expansion. The macro may be recursive.

W256 Undefined directive $

The identifier following a # was not a directive name.

S257 EOF in #include directive

End of file was encountered while processing a **#include** directive.

S258 Unmatched #elif

A **#elif** directive was encountered with no preceding **#if** or **#elif** directive.

S259 Unmatched #else

A **#else** directive was encountered with no preceding **#if** or **#elif** directive.

S260 Unmatched #endif

A **#endif** directive was encountered with no preceding **#if**, **#ifdef**, or **#ifndef** directive.

S261 Include files nested too deeply

The nesting depth of **#include** directives exceeded the maximum (currently 20).

S262 Unterminated macro definition for $

A newline was encountered in the formal parameter list for the indicated macro.

S263 Unterminated string or character constant

A newline with no preceding backslash was found in a quoted string.

I264 Possible nested comment

The characters /* were found within a comment.

W268 Cannot inline subprogram; common block mismatch

W269 Cannot inline subprogram; argument type mismatch

This message may be Severe if the compiler has gone too far to undo the inlining process.

F270 Missing -exlib option

W271 Can't inline $ - wrong number of arguments

I272 Argument of inlined function not used

S273 Inline library not specified on command line (-inlib switch)

F274 Unable to access file $/TOC

S275 Unable to open file $ for inlining

F276 Assignment to constant actual parameter in inlined subprogram

Messages 280–300 are reserved for directive handling.

Runtime Error Messages B

`200 fortran i/o internal error`

This message indicates an error in the runtime library, rather than a user error. It is possible for a user error to cause an internal error. Report internal errors to Customer Support.

`201 i/o call contained bad value for specifier`

An improper specifier value has been passed to an I/O runtime routine. Example: within an **OPEN** statement, `form='unknown'`.

`202 i/o call contained conflicting specifiers`

Conflicting specifiers have been passed to an I/O runtime routine. Example: within an **OPEN** statement, `form='unformatted',blank='null'`.

`203 i/o specifier required but never set`

A specifier required for an I/O runtime routine has not been passed. Example: within an **OPEN** statement, `access='direct'` has been passed, but the record length has not been specified (`recl=specifier`).

`204 attempt to perform a write or open-for-write on a read only file`

Self explanatory. Check file and directory modes.

`205 file disposition conflict - check 'status' and 'dispose'`

In an **OPEN** statement, a file deposition conflict has occurred. Example: within an **OPEN** statement, `status='scratch'` and `dispose='keep'` have been passed.

206 attempt to open a scratch file as a named file

207 attempt to connect two units to the same file

208 attempt to open a previously existing file as 'new'

209 attempt to open a non-existent file as 'old'

210 memory allocation operation failed or fixed buffer overflow

Memory allocation operations occur only in conjunction with namelist I/O. The most probable cause of fixed buffer overflow is exceeding the maximum number of simultaneously open file units.

211 invalid file name

212 invalid unit number

A file unit number less than or equal to zero has been specified.

213 invalid operation on an un-opened file

Unable to open file specified in **ENDFILE** statement.

214 invalid operation for an unconnected unit

Unit specified in **BACKSPACE** statement not connected.

215 file format conflict in read/write operation

Formatted/unformatted file operation conflict.

216 record number error in read/write operation

For direct access, a record number less than one has been specified.

217 attempt to read past end of file

218 item to read/write out of range or smaller than stride

For unformatted files, the I/O item to be read/written is not of a recognizable type, or is smaller than the specified stride. This is an internal error. Report it to Customer Support.

```
219 attempt to read/write value(s) larger than record length
```

For direct access, the record to be read/written exceeds the specified record length.

```
220 attempt to read/write an unopened file
```

```
221 fio$encode_fmt: parsing error
```

A runtime encoded format contains a lexical or syntax error.

```
222 fio$encode_fmt: parse/semantic stack overflow
```

While attempting to encode a runtime format, the parsing or semantic stacks have overflowed. This is an internal error. Report it to Customer Support.

```
223 fmtscan: error in integer constant conversion
```

Integer constant conversion error while encoding a runtime format.

```
224 fmtscan: lexical error within quoted string
```

While attempting to encode a runtime format, an error occurred while scanning a quoted string. Check quote nesting.

```
225 fmtscan: lexical error--unknown token type
```

An unknown token type has been found in a runtime encoded format.

```
226 fmtsemant: unexpected FED in format list
```

An unexpected Fortran edit descriptor has been found in a runtime format item.

```
227 fio$fmt_read: unacceptable input
```

Scan of data fails for indicated data type.

```
230 fio$fmt_read/write: scale factor out of range
```

Fortran P edit descriptor scale factor not within range of -128 to 127.

```
231 fio$fmt_read/write: error on data conversion
```

An internal data conversion error has occurred. Report this error to Customer Support.

232 fio$fmt_read/write: attempt to read/write past end of record

For an internal or direct access file, an attempt to read or write past the end of record has been detected.

234 fio$fmt_read/write: invalid edit descriptor

An invalid edit descriptor has been found in a format.

235 fio$fmt_read/write: i/o list / format edit descriptor mismatch

Data types specified by I/O list item and corresponding edit descriptor conflict.

236 fio$format_decode: parsing error

A runtime encoded format contains a syntax error.

237 fio$fmt_read/write: quad precision type unsupported

238 fio$fmt_read/write: tab value out of range

A tab value of less than one has been specified.

240 fio$fmt_write: item list empty

An unexpected empty write item list has been encountered.

241 fio$fmt_read/write: unix file system error

242 fio$ld_get_token: error in integer constant conversion

Integer constant conversion error while scanning list/namelist-directed input.

243 fio$ld_get_token: lexical error-- unknown token type

Lexical error while scanning list/namelist-directed input.

244 fio$nmlparse: parsing error

Syntax error while parsing namelist-directed input.

```
245 fio$nmlparse: parse/semantic stack overflow
```

While attempting to parse namelist input, the parsing or semantic stacks have overflowed. This is an internal error. Report it to Customer Support.

```
246 fio$fmt_read/write: infinite reversion in format
```

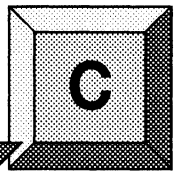
Format does not exhaust item list in **WRITE** statement. Example:

```
      write(6,10) i
10    format(3p)
```

```
247 fio$open: file exists but cannot be opened (check file mode)
```

Likely cause is no user rights (read, write, or execute) to file.

Compiler Internal Structure



This appendix describes the internal structure of the compilers as shown in Figure C-1:

- Scanner and Parser
- Expander
- Optimizer and Vectorizer
- Scheduler and Pipeliner

The front-end of the compiler translates the program into an internal representation called Intermediate Language Macros (ILMs). The ILMs are grouped into basic blocks during the translation phase. A *basic block* represents a sequence of language statements in which the flow of control enters at the beginning and leaves at the end, without the possibility of branching except at the end.

While the source code is translated and grouped into basic blocks, function inlining may occur. Once the translation is complete, optimizations are applied. Depending on the switches selected by the user, a hierarchy of optimizations may be applied: global optimizations, local optimizations, vectorization, and software pipelining.

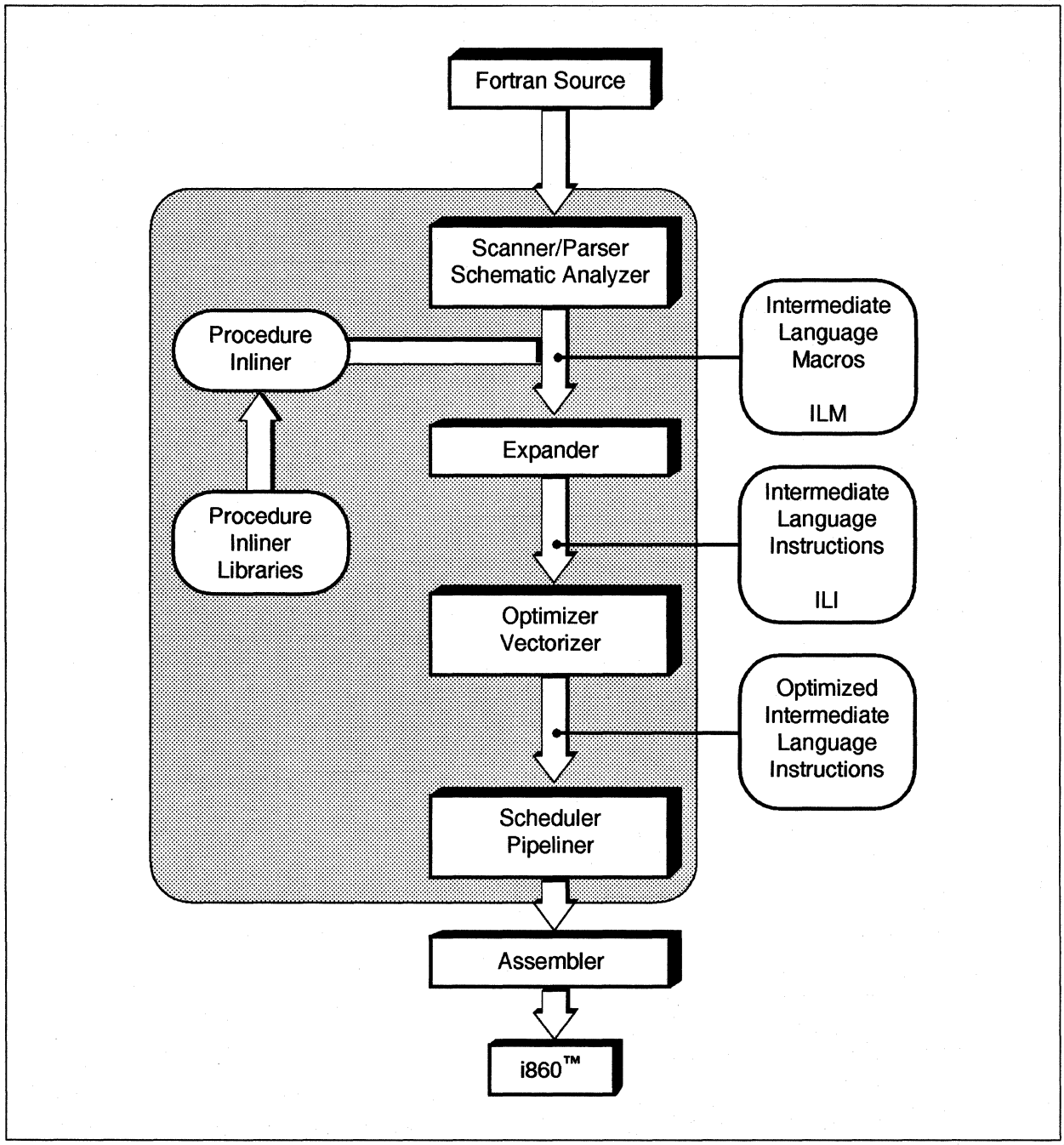


Figure C-1. Compiler Structure

Scanner and Parser

The compiler has a Scanner and Parser that performs syntax and semantic analysis of its respective source language input. The Scanner and Parser create a set of ILMs and a symbol table and various data structures referring back to the original source code for diagnostics and symbolic debugging. They perform error detection and recovery using an advanced multiple parse stack technology.

Expander

The Expander expands the macros in the ILM set along with the semantic analysis information and generates a set of Intermediate Language Instructions (ILIs) and associated data structures including extended basic block tables and information about referenced variables. The Expander also performs certain optimizations, such as constant folding, elimination of identity expressions, and branch folding. The ILI data structure is a directed graph, instead of a tree structure, which simplifies common subexpression elimination.

Optimizer and Vectorizer

The internal, integrated Optimizer/Vectorizer provides both a faster compile time and more efficient code generation than traditional source-to-source preprocessors. The Optimizer/Vectorizer uses advanced optimizations to achieve superior performance. Among these techniques are:

- Procedure Integration
- Internal Vectorization
- Global Optimization
- Local Optimization
- Flexible memory utilization schemes

Procedure Integration

Procedure Integration, also known as function inlining, allows a function to be executed as a part of the originating program instead of having parameters passed and making a call. This results in removing the call overhead and allowing the function to be optimized along with the rest of the program.

Internal Vectorization

The internal vectorizer is oriented to the Intel i860™ microprocessor, which involves transformations that create better opportunities for software pipelining. Recognition of vector forms is only performed when the hand-coded vector library calls will outperform the scheduler. Having an internal vectorizer and software pipeliner allows the compiler to make more precise and informed decisions on code generation opportunities. Other advantages of an internal vectorizer over a source-to-source vectorizer include enhanced debugging capabilities as well as a significant increase in compilation speeds.

Global Optimizations

Global optimizations are those that optimize code over all basic blocks created for a function. Control flow analysis and data flow analysis are performed over a flow graph, where each node of the graph is a basic block. All loops (not just loops created by the language's loop constructs) are detected, and loop optimizations are performed on each loop. These include:

- Invariant Code Motion
- Induction Variable Elimination
- Global Register Allocation
- Dead Store Elimination
- Copy Propagation

Local Optimizations

Local optimizations are performed on an extended basic block. Most of the local optimizations are performed by the code generating phase of the multiple functional units. This technique allows computations from more than one statement to utilize the functional units in parallel, thus providing a fine-grain parallelism that is completely transparent to the program. For loops containing **if** statements (multiple blocks) that are software pipelinable, the compiler provides fine-grain parallelism across multiple blocks. Local optimizations provided by the compilers include:

- Common Subexpression Elimination
- Constant Folding
- Algebraic Identities Removal
- Redundant Load and Store Elimination
- Strength Reduction

- Scratch Register Allocation
- Register Aliasing

The types of code transformations performed on loops include:

- Invariant **if** statement removal
- Loop interchange when advantageous
- Loop invariant vector recognition within nested loops
- Loop fusion
- Common idiom recognition

Flexible Memory Utilization

Support is provided for architectures having an integral data caching scheme. Some techniques provided are:

- Streaming of vectors into cache
- Streaming of invariant vectors into cache and their reuse
- Explicit bypassing of cache for accessing array elements within loops
- Dual and quad loads and stores from and to memory
- Mixing access of arrays from both cache and memory within a loop

Scheduler and Pipeliner

The i860 microprocessor supports parallel activities two ways:

Dual Instruction Mode
: The "core" unit and the floating-point sections can operate independently and in parallel with each other. An example would be a load occurring at the same time that a floating-point add occurs. The compilers test for situations where dual instructions are advantageous and schedules instructions accordingly.

Dual Operation Mode

The floating-point units for some instructions can initiate floating-point adds and multiplies at the same time. In dual operation mode, the two floating-point arithmetic units can operate independently each providing results at the clock rate of the machine. See Figure C-2.

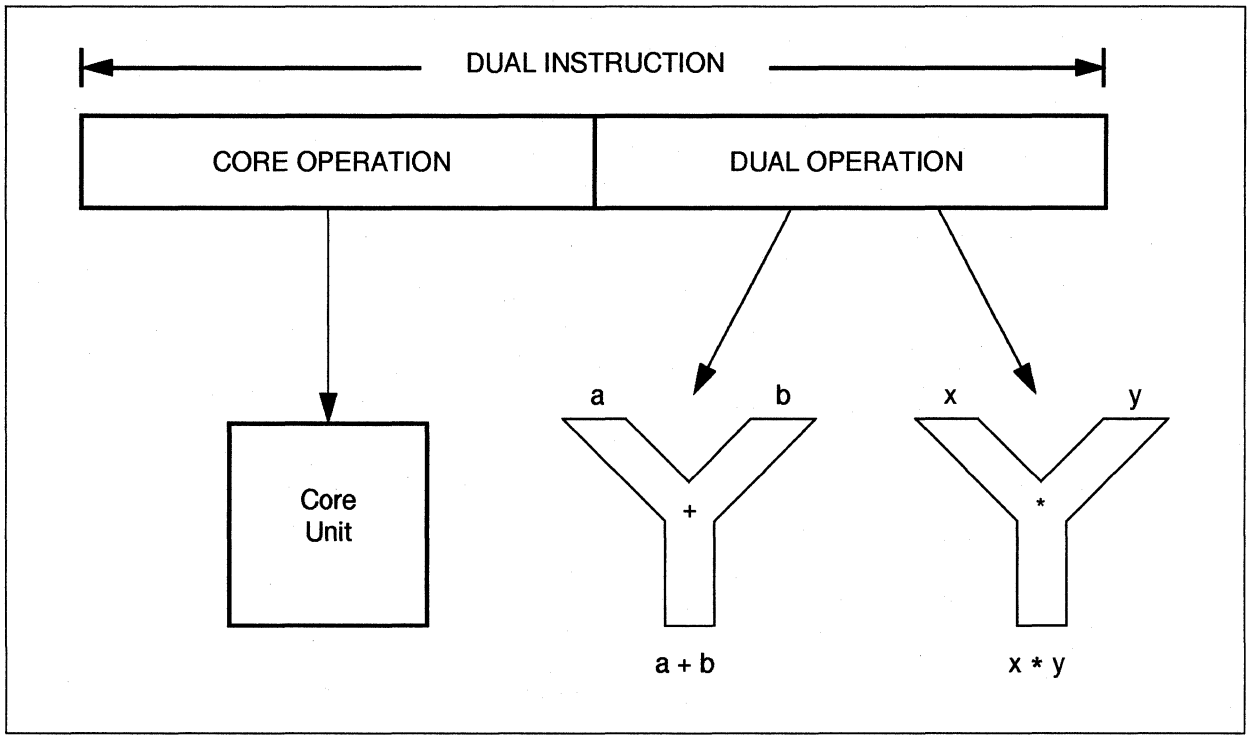


Figure C-2. Parallel Activities of i860™ Microprocessor

The Optimized Intermediate Language Instruction set becomes the input for the Scheduler and Pipeliner, which takes advantage of the i860 microprocessor's dual instruction and operations modes. These unique machine characteristics permit parallel scheduling to multiple functional units and software pipelining.

- Parallel scheduling takes advantage of fine-grain parallelism occurrences in the code and schedules to multiple functional units when possible.

- Software pipelining schedules code so that operations from several iterations of a loop are overlapped. This allows multiple iterations of a loop to be executed during the same instruction. Software pipelining relies on information provided by the global optimizer and vectorizer. This information includes loops that are pipelinable, data dependence information, recurrences, and array references.

The output of the Scheduler and Pipeliner is a list of assembly language instructions that is passed to an assembler to create the final object file.

Manual Pages D

This appendix contains manual pages for compiler-related commands and system calls.

- See the *OSF/1 Command Reference* and *OSF/1 Programmer's Reference* for manual pages for the standard commands and system calls of the operating system.
- See the *Paragon™ System Commands Reference Manual* and the *Paragon™ System Fortran Calls Reference Manual* for manual pages for parallel commands and system calls unique to the Paragon system.

The manual pages in this appendix are also available on-line, using the **man** command.

Table D-1 lists the commands described in this appendix.

Table D-1. Commands Discussed in This Appendix

Manual Page	Commands	Description
ar860	**ar860** (cross) **ar** (native)	Manages object code libraries.
as860	**as860** (cross) **as** (native)	Assembles i860™ source code.
dump860	**dump860** (cross and native)	Dumps object files.
if77	**if77** (cross) **f77** (native)	Compiles Fortran programs.
ifixlib	**ifixlib** (cross and native)	Updates inliner library directories.
ld860	**ld860** (cross) **ld** (native)	Links object files.
mac860	**mac860** (cross) **mac** (native)	Preprocesses assembly-language programs.
nm860	**nm860** (cross) **nm** (native)	Displays symbol table (name list) information.
size860	**size860** (cross) **size** (native)	Displays section sizes of object files.
strip860	**strip860** (cross) **strip** (native)	Strips symbol information from object files.

Except for their names, the cross-development and native versions of each command work the same (with minor exceptions). These commands are available by their cross-development names on the Paragon system and on supported workstations; they are available by their native names on the Paragon system only.

Table D-2 lists the system calls described in this appendix.

Table D-2. System Calls Discussed in This Appendix (1 of 4)

Manual Page	System Calls	Description
abort()	**abort()**	Terminates caller abruptly; writes memory image to core file.
access()	**access()**	Determines access mode or existence of a file.
alarm()	**alarm()**	Executes a subroutine after a specified time.
besj0()	**besj0(), besj1(), besjn(), besy0(), besy1(), besyn(), dbesj0(), dbesj1(), dbesjn(), dbesy0(), dbesy1(), dbesyn()**	Bessel functions.
chdir()	**chdir()**	Changes default directory.
chmod()	**chmod()**	Changes protection mode of a file.
ctime()	**ctime()**	Returns system time as a string.
date()	**date()**	Returns system date as a string.
dv_acos()	**dv_acos(), dv_asin(), dv_atan(), dv_atan2(), dv_cos(), dv_div(), dv_exp(), dv_log(), dv_log10, ()dv_pow(), dv_recp(), dv_rsqrt(), dv_sin(), dv_sqrt(), dv_tan()**	Double-precision vector intrinsics.
erf()	**erf(), erfc(), derf(), derfc()**	Error functions.
etime()	**etime(), dtime()**	Gets elapsed CPU time.
exit()	**exit()**	Terminates program with status.
fdate()	**fdate()**	Returns system date and time as a string.
fgetc()	**fgetc()**	Gets a character from a logical unit.
flmin()	**flmin(), flmax(), ffrac(), dflmin(), dflmax(), dffrac(), inmax()**	Range functions.
flush()	**flush()**	Flushes a logical unit.

Table D-2. System Calls Discussed in This Appendix (2 of 4)

Manual Page	System Calls	Description
fork()	**fork()**	Creates a child process.
fputc()	**fputc()**	Writes a character to a logical unit.
free()	**free()**	Frees memory allocated by **malloc()**.
fseek()	**fseek()**	Positions file pointer.
ftell()	**ftell()**	Determines position of file pointer.
gerror()	**gerror()**	Returns latest system error message.
getarg()	**getarg()**	Gets the *n*th command line argument.
getc()	**getc()**	Gets a character from logical unit 5.
getcwd()	**getcwd()**	Gets the pathname of the current working directory.
getenv()	**getenv()**	Gets the value of an environment variable.
getgid()	**getgid()**	Gets user's group ID.
getlog()	**getlog()**	Gets user's login name.
getpid()	**getpid()**	Gets calling process's OSF/1 process ID.
getuid()	**getuid()**	Gets user's numeric user ID.
gmtime()	**gmtime()**	Formats system time for GMT.
hostnm()	**hostnm()**	Gets name of current host.
iargc()	**iargc()**	Returns index of the last command line argument.
idate()	**idate()**	Returns current system date in numerical form.
ierrno()	**ierrno()**	Returns latest system error number.
ioinit()	**ioinit()**	Initializes I/O.
isatty()	**isatty()**	Determines if logical unit is a TTY.
itime()	**itime()**	Returns current system time in numerical form.
kill()	**kill()**	Sends a signal to a process.

Table D-2. System Calls Discussed in This Appendix (3 of 4)

Manual Page	System Calls	Description
link()	**link()**	Makes a link.
lnblnk()	**lnblnk()**	Returns index of last non-blank in a string.
loc()	**loc()**	Returns the address of an object.
ltime()	**ltime()**	Formats system time for local time zone.
malloc()	**malloc()**	Allocates memory.
mvbits()	**mvbits()**	Moves bits.
outstr()	**outstr()**	Prints a character string to a logical unit.
perror()	**perror()**	Prints error message corresponding to current system error code.
putc()	**putc()**	Writes a character to logical unit 6.
putenv()	**putenv()**	Changes or adds an environment variable.
qsort()	**qsort()**	Quick sort.
rand()	**rand(), irand(), srand()**	Random number generator.
random()	**random(), irandm(), drandm()**	Random number generator.
rename()	**rename()**	Renames a file.
rindex()	**rindex()**	Returns index of substring within a string.
secnds()	**secnds(), dsecnds()**	Returns elapsed time.
signal()	**signal()**	Establishes signal handler.
sleep()	**sleep()**	Suspends execution for a period of time.
stat()	**stat(), lstat(), fstat()**	Gets information about a file.
stime()	**stime()**	Sets system time.
sv_acos()	**sv_acos(), sv_asin(), sv_atan(), sv_atan2(), sv_cos(), sv_div(), sv_exp(), sv_log(), sv_log10(),sv_pow(), sv_recp(), sv_rsqrt(), sv_sin(), sv_sqrt(), sv_tan()**	Single-precision vector intrinsics.

Table D-2. System Calls Discussed in This Appendix (4 of 4)

Manual Page	System Calls	Description
symlnk()	**symlnk()**	Makes a symbolic link.
system()	**system()**	Issues a shell command.
time()	**time()**	Returns system time.
times()	**times()**	Gets process and child process CPU time.
ttynam()	**ttynam()**	Gets pathname of a terminal.
unlink()	**unlink()**	Removes a file.
wait()	**wait()**	Waits for child process to terminate.

AR860 AR860

ar860, ar: Creates and maintains archives for the Paragon(TM) system.

Cross-Development Syntax

ar860 [**-V**] *key* [*options*] *libname* [*filename* ...]

Native Syntax

ar [**-V**] *key* [*options*] *libname* [*filename* ...]

Arguments

libname The name of the archive.

filename The name of the target file.

You must specify one, and only one, *key* from the following list:

d Delete *filename* from the archive.

e Display the symbol tables of COFF objects in the archive.

p Display the archive version of *filename* (may result in binary data being sent to standard output).

q Quickly add the file *filename* to the archive *libname* by appending the file(s) to the end of the archive without checking to see if they duplicate existing files in the archive. If *libname* does not exist, then create it (unless the **c** option is specified). If *filename* does not appear in the archive, then add it.

r Replace the file *filename* in the archive *libname*. If *libname* does not exist, then create it. If *filename* does not appear in the archive, then add it.

t Display the archive table of contents.

x Extract *filename* from the archive. If no file is named, extract all files.

The *key* argument may be preceded by a dash. For example, **ar860 -t file.a** and **ar860 t file.a** are equivalent.

AR860 *(cont.)* AR860 *(cont.)*

You may specify the following *options* in any order:

c Suppress the creation message. This option is used with the **-r** key.

l Use the current working directory for temporary files.

u Replace the archive version only if filename is newer. This option is used only with the **-r** key.

v Verbose mode. For **-r**, display the names of the archive members as they are replaced (or added). For **-d**, display the names of the archive members as they are deleted. For **-t**, display the file mode, the *uid*, the *gid*, the size, and the timestamp of the specified files. For **-x**, display the names of the files as they are extracted.

No space may appear between the *key* and any *options*.

You must specify the following argument, if used, before the *key*:

-V Display the tool banner (tool name, version, etc.).

No space may appear between **-V** and the following *key*, and the *key* may not be preceded by a dash. The dash preceding the **V** is optional. For example, **ar860 -Vt file.a** and **ar860 Vt file.a** are equivalent.

Description

Use **ar860** to manage archives for the Paragon system.

See Also

as860, dump860, icc, if77, ld860, nm860, size860, strip860

AS860 AS860

as860, as: Assembles i860 code for the Paragon(TM) system.

Cross-Development Syntax

as860 [*switches*] [*filename*]

Native Syntax

as [*switches*] [*filename*]

Arguments

filename The name of the i860 assembly language file. If no file is specified, **as860** reads from standard input.

You may specify the following *switches* in any order:

-a Do not automatically import symbols that are referenced but otherwise undefined. Issues an error message for each occurrence.

-l[*listfile*] Write source listing in the file *listfile*, a file in the current working directory. If you omit *listfile*, the listing goes to standard output.

-L Preserve text symbols starting with ".L" in the debug section.

-o *objfile* Put the output object file in *objfile*. If you omit this switch, the default object file name is produced by stripping any directory prefixes from *filename*, stripping any of the suffixes ".n10", ".s", ".mac", or ".860", and appending ".o". An existing file with the same name is silently overwritten.

-R Suppress all **.data** directives. Code and data are both assembled into the **.text** section.

-V Display the tool banner (tool name, version, etc.).

-x Enable additional checks of the source file to find illegal sequences of instructions.

AS860 *(cont.)* AS860 *(cont.)*

Description

Use **as860** to assemble the named file.

You can ensure that the proper switches are passed to **as860** by accessing **as860** using the compiler drivers (**icc** or **if77**).

Not all illegal sequences are detected when the **-x** switch is used.

See Also

ar860, dump860, icc, if77, ld860, nm860, size860, strip860

DUMP860 DUMP860

Dumps parts of a Paragon(TM) system object file.

Syntax

dump860 [*switches*] *filename*

Arguments

filename The name of the object file.

You may specify the following *switches* in any order:

-a Display archive headers.

-c Dump the string table.

-d *number* Dump section headers starting at section *number*. Only effective if the **-h** switch is also specified. Sections are numbered starting at 1. If the **+d** switch is not specified, then only the single section header is dumped.

+d *number* Dump section headers ending at section *number*. Only effective if the **-h** switch is used.

-f Display file headers.

-g Display the archive symbol table.

-h Dump section headers.

-l Dump line numbers.

-n *name* Dump only sections named *name*. Only effective if the **-h** switch is used.

-o Dump (in formatted hexadecimal) optional headers.

-p Do not display headers.

-r Dump relocation data.

-s Dump section data.

DUMP860 *(cont.)*

DUMP860 *(cont.)*

-t [*number*]	Dump symbol table, starting at symbol index *number*. If the **+t** switch is not used, then only the single symbol is displayed.
+t *number*	Dump symbol table, through symbol index *number*. If **-t** was not specified, the start index is zero.
-u	Underline mode. Only works on devices supporting backspace.
-v	Verbose mode. Display some headers and information in an easier-to-comprehend form.
-V	Display the tool banner (tool name, version, etc.).
-z *name,number*	Dump line numbers for function *name*, starting at line *number*.
+z *number*	Dump line numbers for function *name* (specified by **-z**), ending at line *number*.

Description

Use **dump860** to dump (in formatted hexadecimal) parts of the named object file.

See Also

ar860, as860, icc, if77, ld860, nm860, size860, strip860

IF77 IF77

if77, **f77**: Driver for compiling, assembling, and linking Fortran programs for the Paragon(TM) system.

Cross-Development Syntax

if77 [*switches*] *sourcefile...*

Native Syntax

f77 [*switches*] *sourcefile...*

Description

The **if77** command invokes the Fortran compiler, assembler, and linker with switches derived from **if77**'s command line switches.

if77 bases its processing on the suffixes of the files it is passed:

file.F	is a Fortran program with preprocessor directives. It is preprocessed, compiled and assembled. The resulting object file is placed in the current directory.
file.f	is a Fortran program. It is compiled and assembled. The resulting object file is placed in the current directory.
file.s	is an i860 assembly language file. It is assembled and the resulting object file is placed in the current directory.
file.o	is an object file. It is passed directly to the linker if linking is requested.
file.a	is an **ar** library. It is passed directly to the linker if linking is requested.
file.c	is a C program. It is passed to the C compiler.

All other files are taken as object files and passed to the linker (if linking is requested) with a warning message. If a file's suffix does not match its actual contents, unexpected results may occur.

If a single Fortran program is compiled and linked with one **if77** command, then the intermediate object and assembly files are deleted.

By default, Fortran local variables are placed on the stack. Some Fortran programs assume that all variables are allocated statically. Static allocation can be forced with the **-Msave** switch.

IF77 *(cont.)*

IF77 *(cont.)*

Switches

-c Skips the link step; compiles and assembles only. Leaves the output from the assemble step in a file named *file*.o for each file named *file*.f (unless you also use the **-o** switch).

-D*name*[=*def*] Defines *name* to be *def* in the preprocessor. If *def* is missing, it is assumed to be empty. If the = sign is also missing, then *name* is defined to be the string `1`.

-E Preprocesses each ".F" file and sends the result to *stdout*. No compilation, assembly, or linking is performed.

-F Preprocesses each ".F" file and leaves the output in a file named *file*.f for each file named *file*.F.

-g Equivalent to **-Mdebug -O0 -Mframe**.

-I- Accepted, but has no affect.

-I*directory* Add a specified directory to the compiler's search path for include files where *directory* is the pathname of the directory to be added. If you use more than one **-I** switch, the specified directories are searched in the order they were specified (left to right).

The **INCLUDE** statement directs the compiler to begin reading from another file. The compiler uses two rules to locate the specified file. Note that the Fortran **INCLUDE** statement is different from the **#include** statement, whcih uses the C preprocessor.

1. If the filename specified in the **INCLUDE** statement includes a pathname, the compiler begins reading from the file it specifies.

2. If no pathname is provided in the **INCLUDE** statement, the compiler searches for the file in the following order:

- any directories specified with **-I**
- the directory containing the source file
- the current directory

-K*option* Requests special mathematical semantics. The *option* values are:

IF77 *(cont.)*

IF77 *(cont.)*

ieee (default) If used while linking, links in a math library that conforms with the IEEE 754 standard.

If used while compiling, tells the compiler to perform **real** and **double precision** divides in conformance with the IEEE 754 standard.

ieee=enable If used while linking, has the same effects as **-Kieee**, and also enables floating point traps and underflow traps. If used while compiling, has the same effects as **-Kieee**.

ieee=strict If used while linking, has the same effects as **-Kieee=enable**, and also enables inexact traps. If used while compiling, has the same effects as **-Kieee**.

noieee If used while linking, produces a program that flushes denormals to 0 on creation (which reduces underflow traps) and links in a math library that is not as accurate as the standard library, but offers greater performance. This library offers little or no support for exceptional data types such as **INF** and **NaN**, and will trap on such values when encountered.

If used while compiling, tells the compiler to perform **real** and **double precision** divides using an inline divide algorithm that offers greater performance than the standard algorithm. This algorithm produces results that differ from the results specified by the IEEE standard by no more than three units in the last place.

trap=fp If used while linking, disables kernel handling of floating point traps. Has no effect if used while compiling.

trap=align If used while linking, disables kernel handling of alignment traps. Has no effect if used while compiling

IF77 *(cont.)*

IF77 *(cont.)*

-l*library* Load the library **lib***library***.a**. The library is loaded from the first library directory in the library search path (see the **-L** switch) in which a file of that name is encountered. (Passed to the linker.)

-L*directory* Adds *directory* to beginning of the library search path. Also see the **nostdlib** and **nostartup** options of the **-M** switch. (Passed to the linker; see the **ld860** manual page for more information on the library search path.)

-m Produces a link map. (Passed to the linker.)

-M*option* Requests specific actions from the compiler. The *option* values are as follows (an unrecognized **-M** *option* is passed directly to the compiler):

alpha Activate alpha-release compiler features.

anno Produce annotated assembly files, where source code is intermixed with assembly language. **-Mkeepasm** or **-S** should be used as well.

beta Activate beta-release compiler features.

[no]bounds [Don't] enable array bounds checking (default **-Mnobounds**).

clr_reg Clear the internal registers after every procedure invocation. This option is used for diagnostic purposes.

concur[=*option*[,*option*...]]
Make loops parallel as defined by the specified *options*. *option* can be any of the following:

altcode:*count* - Make innermost loops without reduction parallel only if their iteration count exceeds *count*. Without this switch, the compiler assumes a default *count* of 100.

altcode_reduction:*count* - Make innermost loops with reduction parallel only if their iteration count exceeds *count*. Without this switch, the compiler assumes a default *count* of 200.

dist:block - Make the outermost valid loop in any loop nest parallel. This is the default option.

IF77 *(cont.)*

IF77 *(cont.)*

dist:cyclic - Make the outermost valid loop in any loop nest parallel. If an innermost loop is made parallel, its iterations are allocated to processors cyclically. That is, processor 0 performs iterations 0, 3, 6, ...; processor 1 performs iterations 1, 4, 7, ...; and processor 2 performs iterations 2, 5, 8, and so on.

global_vcache - Directs the vectorizer to locate the cache within the area of an external array when generating codes for parallel loops. By default, the cache is located on the stack for parallel loops.

noassoc - Do not make loops with reductions parallel.

cncall

Make loops with calls parallel. By default, the compiler does not make loops with calls parallel since there is no way for the compiler to verify that the called routines are safe to execute in parallel. When you specify **-Mcncall** on the command line, the compiler also automatically specifies **-Mreentrant**.

-Mcncall also allows several other types of loops to be made parallel:

- loops with I/O statements

- loops with conditional statements

- loops with low loop counts

- non-vectorizable loops

If the compiler can detect a cross-iteration dependency in a loop, it will not make the loop parallel, even if **-Mcncall** is specified.

IF77 *(cont.)*

IF77 *(cont.)*

cpp860 Direct the internal preprocessor to not compress white space.

cray=*option*[**,***option*...]
Force Cray Fortran (CF77) compatibility with respect to listed options. Currently, only one option is supported.

pointer For puposes of optimization is is assumed that pointed-based variables do not overlay the storage of any other variable.

[no]dclchk [Don't] require that all variables be declared (default **-Mnodclchk**).

[no]debug [Don't] generate symbolic debug information (default **-Mnodebug**). If -Mdebug is specified with an optimization level greater than zero, line numbers will not be generated for all program statements.

[no]depchk [Don't] check for potential data dependencies (default **-Mdepchk**). **-Mnodepchk** is especially useful in disambiguating unknown data dependencies arising from use of array subscripts that cannot be derived at compile time. For example, if an array is referenced in a loop using the induction variable plus some other unknown non-induction-based variable as a subscript, the compiler must assume that the array conflicts with a similar array reference based on the induction variable alone. If it is known that the two array references do not conflict, then this switch may result in better code. Do not use this switch if such data dependencies do exist, because incorrect code may result.

IF77 *(cont.)*

IF77 *(cont.)*

[no]dlines [Don't] treat lines beginning with D in column 1 as executable statements, ignoring the D (default **-Mnodlines**).

dollar,*char* Specify *char* as the character to which the compiler maps the dollar sign. The compiler allows the dollar sign in names.

extend Allow 132-column source lines (normally only 72 columns are allowed).

extract=[*option*[,*option*...]]
Pass options to the subprogram extractor (see the **inline** option for more information). The *options* are:

[**name:**]*subprogram*—Extract the specified subprogram. **name:** must be used if the subprogram name contains a period.

[**size:**]*number*—Extract subprograms containing less than approximately *number* statements.

If both *number*(s) and *subprogram*(s) are specified, then subprograms matching the given name(s) *or* meeting the size requirements are extracted.

The **-o***file* switch must be used with **-Mextract** to tell the compiler where to place the extracted subprograms. The name of the specified *file* must contain a period.

[no]frame [Don't] include the frame pointer (default **-Mnoframe**). **-Mnoframe** can improve execution time and decrease code, but makes it impossible to get a call stack traceback when using a debugger.

[no]i4 [Don't] treat **integer** as **integer*4** (default **-Mi4**). **-Mnoi4** treats **integer** as **integer*2**.

IF77 *(cont.)*

IF77 *(cont.)*

info=[*option*[**,***option*...]]
Produce useful information to the standard error output. The *option*s are:

time or **stat**—Output compilation statistics.

loop—Output information about loops. This includes information about vectorization, software pipelining, and parallelization.

concur—Same as **-Minfo=loop**.

inline—Output information about subprograms extracted and inlined.

cycles or **block** or **size**—Output block size in cycles. Useful for comparing various optimization levels against each other. The cycle count produced is the compiler's static estimate of freeze-free cycles for the block.

ili—Output intermediate language as comments in assembly file.

all—All of the above.

inline=[*option*[**,***option*...]]
Pass options to the subprogram inliner. The *options* are:

[**lib:**]*library*—Inline subprograms in the specified inliner library (produced by **-Mextract**). If **lib:** is not used, the library name must contain a period. If no library is specified, subprograms are extracted from a temporary library created during an extract prepass.

[**name:**]*subprogram*—Inline the specified subprogram. If **name:** is not used, the subprogram name must not contain a period.

[**size:**]*number*—Inline subprograms containing less than approximately *number* statements.

IF77 *(cont.)*

IF77 *(cont.)*

levels:*number*—Perform *number* levels of inlining (default 1).

If both *number*(s) and *subprogram*(s) are specified, then subprograms matching the given name(s) *or* meeting the size requirements are inlined.

iomutex Place critical sections around I/O statements.

A critical section is a portion of the code that is only executed by one thread. The switch **-Miomutex** places critical sections around all I/O statements whether they are in loops or not. This is a switch that should be active if you want to parallelize loops that contain I/O statements.

You must specify **-Mncall** or use a directive to parallelize loops that contain I/O statements.

The switch **-Mconcur** includes an imbedded **-Miomutex**. Also, the switch **-Mncall** includes an imbedded **-Miomutex**.

If you are not parallelizing loops with I/O statements, specify **-Mnoiomutex**. This is because, if you are not parallelizing loops with I/O statements, you do not need to put I/O statements in critical sections.

-Miomutex does not affect correct execution of the program. It makes sure that only one thread is executing the I/O statement at a time.

keepasm Keep the assembly file for each Fortran source file, but continue to assemble and link the program. This is mainly for use in compiler performance analysis and debugging.

IF77 *(cont.)*

IF77 *(cont.)*

list[=*name*] Create a source listing in the file *name*. If *name* is not specified, the listing file has the same name as the source file except that the ".f" suffix is replaced by a ".lst" suffix. If *name* is specified, the listing file has that name; no extension is appended.

nolist Don't create a listing file (this is the default).

[no]longbranch [Don't] allow compiler to generate **bte** and **btne** instructions (default **-Mlongbranch**). **-Mnolongbranch** should be used only if an assembly error occurs.

neginfo=concur
Print information for each countable loop that is not made parallel stating why the loop was not made parallel.

noansi Allow multiple implicit statements.

nostartup Don't link the usual start-up routines (*crt0.o* and *ifmain.o*), which contain the entry point for the program.

nostdinc Remove the default include directory (*/usr/include* for **f77**, *$(PARAGON_XDEV)/paragon/include* for **if77**) from the include files search path.

nostdlib Don't link the standard libraries (*libpm.o*, *guard.o*, *libf.a*, *libm.a*, *libc.a*, *libic.a*, and *libmach3.a*) when linking a program.

onetrip Force each **do** loop to be iterated at least once.

IF77 *(cont.)*

IF77 *(cont.)*

[no]perfmon [Don't] link the performance monitoring module (*libpm.o*) (default **-Mperfmon**). See the *Paragon(TM) System Application Tools User's Guide* for information on performance monitoring.

prof=*x* This option is ignored.

[no]quad [Don't] force top-level objects (such as local arrays) of size greater than or equal to 16 bytes to be quad-aligned (default **-Mquad**). Note that **-Mquad** does not affect items within a top-level object; such items are quad-aligned only if appropriate padding is inserted. Common blocks are always quad-aligned.

[no]r8 [Don't] treat **real** as **double precision** and **real** constants as **double precision** constants (default **-Mnor8**).

[no]r8intrinsics [Don't] treat intrinsics as follows (default **-Mnor8intrinsics**):

cmplx as **dcmplx**
real as **dble**
alog as **dlog**
amax1 as **dmax1**
amin1 as **dmin1**
amod as **dmod**
csqrt as **cdsqrt**
clog as **cdlog**
cexp as **cdexp**
csin as **cdsin**
ccos as **cdcos**

[no]recursive [Don't] allocate local variables on the stack, thus allowing recursion (default **-Mnorecursive**). **SAVE**d, data-initialized, or namelist members are always allocated statically, regardless of the setting of this switch.

IF77 *(cont.)*

IF77 *(cont.)*

[no]reentrant [Don't] generate reentrant code (default **-Mnoreentrant**). **-Mreentrant** disables certain optimizations that can improve performance but may result in code that is not reentrant. Even with **-Mreentrant**, the code may still not be reentrant if it is improperly written (e.g., declares static variables). You may need to increase the stacksize before executing programs compiled with **-Mreentrant**. A segmentation violation occurs if stacksize is too low.

reloc_libs Causes **-l** switches that appear before source or object file names on the compiler command line to appear after these file names on the **ld** command line.

safealloc Informs the compiler that all allocatable commons are allocated once and can be treated as ordinary commons for optimization purposes. This option can improve performance for some applications, but should be used with caution.

[no]save [Don't] allocate all local data in static locations instead of on the stack (default **-Msave**). The effect is similar to using the **save** statement for all local variables. Recursion is not allowed with this switch in effect. **-Msave** may allow some older Fortran programs to run, but may decrease performance.

[no]signextend [Don't] sign-extend the result of a conversion of a signed integer to a smaller signed type (default **-Mnosignextend**). For example, if **-Msignextend** is in effect and an **integer*4** containing the value 65535 is converted to an **integer*2**, the value of the **integer*2** will be -1. This option is provided for compatibility with other compilers. **-Msignextend** will decrease performance.

split_loop_ops=*n*
Set a threshold of *n* floating-point operations within a loop. Innermost loops whose number of floating-point operations exceeds *n* are split. Each floating-point operation counts as two. The default for *n* is 40 when **-Mvect** is used.

IF77 *(cont.)*

IF77 *(cont.)*

nosplit_loop_ops
Do not split loops when the floating-point operation threshold is exceeded. When **-Mvect** is specified, innermost loops whose number of floating point operations exceed 40 are split by default. This switch turns the default off.

split_loop_refs=*n*
Set a threshold of *n* array element loads and stores within a loop. Innermost loops whose number of loads and stores exceeds *n* are split. The default for *n* is 20 when **-Mvect** is used.

nosplit_loop_refs
Do not split loops when the array element loads and stores threshold is exceeded. When **-Mvect** is specified, innermost loops whose number of array element loads and stores exceeds 20 are split by default. This switch turns the default off.

standard Flag non-ANSI-Fortran77 usage.

[no]streamall [Don't] stream all vectors to and from cache in a vector loop (default **-Mstreamall**). When **-Mnostreamall** is in effect, the compiler chooses one vector to come directly from or go directly to main memory, without being streamed into or out of cache.

[no]stride0 [Don't] inhibit certain optimizations and allow for stride 0 array references. **-Mstride0** may degrade performance, and should only be used if zero stride induction variables are possible. (default **-Mnostride0**).

unixlogical Set the value of a logical expression to one if the result is **.TRUE.**.

[no]upcase [Don't] preserve case in names (default **-Mnoupcase**). **-Mnoupcase** causes all names to be converted to lower case. Note that, if **-Mupcase** is used, then variable name *Q* is different than variable name *q*, and keywords must be in lower case.

IF77 *(cont.)*

IF77 *(cont.)*

unroll[=*option* [,*option* ...]]
Invoke the loop unroller and set the optimization level to 2 if it is set to less than 2. *option* is one of the following:

c:*m* - Completely unroll loops with a constant loop count less than or equal to *m*. If *m* is not supplied, the default value is 4.

n:*u* - Unroll loops that are not completely unrolled or have a non-constant loop count *u* times. If *u* is not supplied, the unroller computes the number of times a loop is unrolled.

nounroll Do not unroll loops.

vect[=*option*[,*option*...]]
Perform vectorization (also enables **-Mvintr**). If no *options* are specified, then all vector optimizations are enabled. The available *options* are:

altcode[:*number*] - Produce non-vectorized code to be executed if the loop count is less than or equal to *number*. Otherwise execute vectorized code. The default value for *number* is 10.

cachesize:*number*—This sets the size of the portion of the cache used by the vectorizer to *number* bytes. *Number* must be a multiple of 16, and less than the cache size of the microprocessor (16384 for the i860 XP, 8192 for the i860 XR). In most cases the best results occur when *number* is set to 4096, which is the default (for both microprocessors).

noassoc—When scalar reductions are present (for example, dot product), and loop unrolling is turned on, the compiler may change the order of operations so that it can generate better code. This transformation can change the result of the computation due to round-off error. The use of **noassoc** prevents this transformation.

[no]recog—[Don't] Recognize certain loops as simple vector loops and call a special routine.

IF77 *(cont.)*

IF77 *(cont.)*

smallvect[*:number*]—This option allows the vectorizer to assume that the maximum vector length is no greater than *number*. *Number* must be a multiple of 10. If *number* is not specified, the value 100 is used. This option allows the vectorizer to avoid stripmining in cases where it cannot determine the maximum vector length. In doubly-nested, non-perfectly nested loops this option can allow invariant vector motion that would not otherwise have been possible. Incorrect code may result if this option is used, and a vector takes on a length greater than specified.

streamlim:*n* This sets a limit for application of the vectorizer data streaming optimization. If data streaming requires cache vectors of length less than *n*, the optimization is not performed. Other vectorizer optimizations are still performed. The data streaming optimization has a high overhead compared to other loop optimizations, and can be counter-productive when used for short vectors. The *n* specifier is not optional. The default limit is 32 elements if **streamlim** is not used.

transform—Perform high-level transformations such as loop splitting and loop interchanging. This is normally not useful without **-Mvect=recog**.

-Mvect with no options means **-Mvect=recog,transform,cachesize:4096,altcode:10**.

[no]vintr [Don't] perform recognition of vector intrinsics (default **-Mnovintr**, unless **-Mvect** is used).

[no]xp [Don't] use i860 XP microprocessor features (default **-Mxp**).

-nostdinc Equivalent to **-Mnostdinc**.

IF77 *(cont.)*

IF77 *(cont.)*

-nx Creates an executable application for multiple nodes.

- Using **-nx** while compiling has no effect.
- Using **-nx** while linking creates an application that automatically copies itself into multiple nodes. It also links in *libnx.a*, the library that contains the calls in the *Paragon(TM) System Fortran Calls Reference Manual*. You can control the execution of an application linked with **-nx** by using command-line switches and environment variables, as described in the *Paragon(TM) System User's Guide*.

To link in *libnx.a* without creating an application that automatically copies itself into multiple nodes, use **-lnx** instead. An application linked with **-lnx** can use operating system calls to create node processes under program control.

-node is currently accepted as a synonym for **-nx**, but this support may be dropped in a future release.

-o*file* Uses *file* for the output file, instead of the default **a.out** (or *file*.o if used with the **-c** switch).

-O[*level*] Set the optimization level:

0	A basic block is generated for each Fortran statement. No scheduling is done between statements. No global optimizations are performed.
1	Scheduling within extended basic blocks is performed. Some register allocation is performed. No global optimizations are performed.
2	All level 1 optimizations are performed. In addition, traditional scalar optimizations such as induction recognition and loop invariant motion are performed by the global optimizer.
3	All level 2 optimizations are performed. In addition, software pipelining is performed.
4	All level 3 optimizations are performed, but with more aggressive register allocation for software pipelined loops. In addition, code for pipelined loops is scheduled several ways, with the best way selected for the assembly file.

IF77 *(cont.)*

IF77 *(cont.)*

If a *level* is not supplied with **-O**, the optimization level is set to 2. If **-O** is not specified, the default level is 1. Setting optimization to levels higher than 0 may reduce the effectiveness of symbolic debuggers.

-p This option is ignored.

-r Generates a relinkable object file. (Passed to the linker.)

-s Strips symbol table information. (Passed to the linker.)

-S Skips the link and assemble steps. Leaves the output from the compile step in a file named *file*.s for each file named *file*.f.

-U*name* Remove any initial definition of *name* in the preprocessor. Since all **-D** switches are processed before all **-U** switches, the **-U** switch can be used to override the **-D** switch.

-v Prints the entire command line for each tool as it is invoked, and invokes each tool in verbose mode (if it has one).

-V Prints the version banner for each tool (assembler, linker, etc.) as it is invoked.

-VV Display the driver version number and the location of the online release notes. No compilation is performed.

-W*pass*,*option*[,*option*...]
Passes the specified *option*s to the specified *pass*:

0 (zero)	Compiler.
a	Assembler.
l	Linker.

Each comma-delimited string is passed as a separate argument.

IF77 *(cont.)*

IF77 *(cont.)*

-Y*pass,directory*

Looks for the executable file for the specified *pass* in the specified *directory* (rather than in the default location), where *pass* is one of the following:

0 (zero)	Compiler executable file.
a	Assembler executable file.
l	Linker executable file.
S	Startup object files.
I	Standard include files.
L	Standard libraries (passes **-YL***directory* to the linker).
U	Secondary libraries (passes **-YU***directory* to the linker).
P	All libraries (passes **-YP***directory* to the linker).

See the **ld860** manual page for more information on the **-YL**, **-YU**, and **-YP** switches.

Files

a.out	Executable output file.
file.a	Library of object files.
file.f	Fortran source file.
file.F	Fortran source file for preprocessing.
file.lst	Listing file produced by **-Mlist**.
file.o	Object file.
file.s	Assembler source file.

IF77 *(cont.)*

IF77 *(cont.)*

The following files and directories are used in the cross-development environment (**if77**). *PARAGON_XDEV* is an environment variable that can be set to the root of the compiler installation directory. If *PARAGON_XDEV* is not set, the default is */usr/paragon/XDEV*. The directory where the driver, compiler, and tools are located must be included in your path. For Sun4 users, for example, $*PARAGON_XDEV/paragon/bin.sun4* would be included in the path.

$(PARAGON_XDEV)/paragon/bin.arch	Directory containing executables for system *arch* (*arch* identifies the architecture of the system, e.g. **sgi** or **sun4**).
$(PARAGON_XDEV)/paragon/bin.arch/if77	Fortran compiler driver.
$(PARAGON_XDEV)/paragon/bin.arch/iftn	Fortran compiler.
$(PARAGON_XDEV)/paragon/bin.arch/as860	Intel (COFF) assembler.
$(PARAGON_XDEV)/paragon/bin.arch/ld860	Intel (COFF) linker.
$(PARAGON_XDEV)/paragon/include	Standard include directory.
$(PARAGON_XDEV)/paragon/lib-coff	Standard library directory.
$(PARAGON_XDEV)/paragon/lib-coff/crt0.o	C start-up routine.
$(PARAGON_XDEV)/paragon/lib-coff/ifmain.o	Fortran initial routine.
$(PARAGON_XDEV)/paragon/lib-coff/libpm.o	Performance monitoring module.
$(PARAGON_XDEV)/paragon/lib-coff/guard.o	Barrier between user and system code.
$(PARAGON_XDEV)/paragon/lib-coff/libf.a	Fortran runtime library.
$(PARAGON_XDEV)/paragon/lib-coff/libm.a	Math library.
$(PARAGON_XDEV)/paragon/lib-coff/libc.a	Standard C library.
$(PARAGON_XDEV)/paragon/lib-coff/libic.a	C built-in intrinsic library.
$(PARAGON_XDEV)/paragon/lib-coff/libmach.a	Mach operating system library.
$(PARAGON_XDEV)/paragon/lib-coff/noieee	Library directory used when linking with **-Knoieee** (contains non-IEEE versions of *libf.a* and *libm.a*).
$(PARAGON_XDEV)/paragon/lib-coff/options/autoinit.o	Routine linked in when **-nx** is used.

IF77 *(cont.)*

IF77 *(cont.)*

The following files and directories are used by default in the native environment (**f77**). If / is not the root of the compiler installation directory, you must set *PARAGON_XDEV* to this directory and add $*PARAGON_XDEV*/*usr*/*ccs*/*bin* to your path.

/usr/ccs/bin	Directory containing executables.
/usr/ccs/bin/f77	Fortran compiler driver.
/usr/ccs/bin/iftn	Fortran compiler.
/usr/ccs/bin/as	Assembler.
/usr/ccs/bin/ld	Linker.
/usr/include	Standard include directory.
/usr/lib	Standard library directory.
/usr/lib/crt0.o	C start-up routine.
/usr/lib/ifmain.o	Fortran initial routine.
/usr/lib/libpm.o	Performance monitoring module.
/usr/lib/guard.o	Barrier between user and system code.
/usr/lib/libf.a	Fortran runtime library.
/usr/lib/libm.a	Math library.
/usr/lib/libc.a	Standard C library.
/usr/lib/libic.a	C built-in intrinsic library.
/usr/lib//libmach.a	Mach operating system library.
/usr/lib/noieee	Library directory used when linking with **-Knoieee** (contains non-IEEE versions of *libf.a* and *libm.a*).
/usr/lib/options/autoinit.o	Routine linked in when **-nx** is used.

Environment Variables

The environment variable *MAKECPP* is supported. *MAKECPP* is a colon-separated list of directories that is added to the compiler search path for include files.

If you use the **-Knoieee** switch and define *LPATH* or *PARAGON_LPATH*, be sure that the directory containing the noieee versions of *libf.a* and *libm.a* is listed before a directory containing the ieee versions of these libraries. If in doubt, compile with the **-v** switch to see which libraries are linked in. See the **ld860** manual page for more information.

Diagnostics

The compiler produces information and error messages as it translates the input program. The linker and assembler may generate their own error messages.

See Also

ar860, as860, dump860, icc, ifixlib, ld860, nm860, size860, strip860

IFIXLIB IFIXLIB

Update an inliner library directory.

Syntax

ifixlib *library_name*

Arguments

library_name The name of an inliner library.

Description

An inliner library is implemented as a directory. For each element of the library, the directory contains a file containing the encoded form of the inlinable function. A special file named *TOC* serves as a directory for the library. This is a printable ASCII file that can be examined for information about the library contents. When an element is added to or removed from the library, the *TOC* file becomes out of date. The **ifixlib** command updates the *TOC* file for the specified inliner library.

See Also

icc, **if77**

LD860 LD860

ld860, ld: Link editor for Paragon(TM) system object files.

Cross-Development Syntax

ld860 [*switches*] *filename ...*

Native Syntax

ld [*switches*] *filename ...*

Arguments

filename The name of the object file or library.

You may specify the following *switches* in any order:

-B *integer* Specify the address to use for the base of the **.bss** section for all following object modules. This switch may be used multiple times, and affects only objects that appear after the switch in the command line.

-contig Force the **.data** section to follow the **.text** section. Overrides **-d**.

-d *integer* Specify the address at which the **.data** section is to be loaded. The default is 0x4010000.

-debug Provide a listing of where routines are referenced.

-D Display the C++ **.debug** section.

-D *integer* Specify the length of the **.data** section to be *integer* bytes. The **.data** section is padded with zero to the specified length, which may not be less than the summed length derived from the object modules.

-e *symbol* Specify *symbol* as the entry-point. The default entry-point is `start`.

-f *filelist* Read in a list of files to be linked from file *filelist*. Names in the file can be separated by a comma, a space, a tab, or a linefeed. This switch may be used multiple times.

-k Start the **.text** and **.data** sections exactly at the addresses specified by the **-T** and **-d** switches (or at the defaults if the switches are not given) without performing the normal modifications to those addresses to make the file pageable.

LD860 *(cont.)*

LD860 *(cont.)*

-l*library* Load the library **lib***library***.a**. The library is loaded from the first library directory in the library search path in which a file of that name is encountered.

-L Display the C++ **.line** section.

-L*directory* Add *directory* to the beginning of the library search path.

-m Generate a link map (listing of modules and addresses).

-o *objfile* Put the output object file in *objfile*. If this switch is not specified, the default object file name is *a.out*. If a file with the same name already exists, it is silently replaced.

-p Align the **.data** section of the following module on a logical page boundary. (Other switches may appear between **-p** and the filename.) This switch may be repeated as necessary, and applies only to the next object file.

-P *integer* Set the logical page size to *integer* bytes (default 65536). The value of *integer* must be a power of two multiple of 4096 bytes.

-r Retain relocation entries in the output object file to allow incremental linking. The output object file produced with **-r** can be used as an input object file in another link. When **-r** is used, **-o** must also be specified.

-s Strip all symbols from the output object file.

-t Display the name of each object file or library as it is processed.

-T *integer* Specify the address at which the **.text** section is to be loaded. The default is 0x10000. If used without **-d**, implies **-contig**.

-u *symbol* Initialize the symbol table with *symbol*. The linker considers *symbol* to be undefined.

-V Display the tool banner (tool name, version, etc.).

-y*file* Load the library *file*. The library is loaded from the first library directory in the library search path in which a file of that name is encountered. (**-y** is like **-l**, but uses the specified filename without modifications.)

-YL*directory* Replace the standard library directory (the first directory in the library search path) with *directory*.

-YU*directory* Replace the secondary library directory (the second directory in the library search path) with *directory*.

-YP*directory* Replace the entire library search path with *directory*.

LD860 *(cont.)* LD860 *(cont.)*

Description

Use **ld860** to link-edit the named file(s).

Object files and libraries are processed in the order specified.

Libraries are searched for unsatisfied externals when they are processed, and are not reopened to satisfy any symbols that might not have been satisfied. The search for libraries is done in the following order:

- If *PARAGON_LPATH* is defined, it is searched.
- If *PARAGON_LPATH* is not defined and *LPATH* is defined, it is searched.
- Any directories specified using the **-L** switch prior to **-l***libname* on the command line are searched.
- The standard default libraries are searched. In the cross-development environment, the default library directories are:

  $PARAGON_XDEV/paragon/lib-coff:*$PARAGON_XDEV/paragon/lib-coff/options*

  In the native environment, the default library directories are:

  $PARAGON_XDEV/usr/lib :*$PARAGON_XDEV/usr/lib/options*

  If *PARAGON_XDEV* is not set, */usr/lib*:*/usr/lib/options* is the default.

The search path used by the -l switch can be modified by any **-L**, **-YL**, **-YU**, or **-YP** switch to the left of the **-l** switch on the command line. The effect of these switches is cumulative.

The **-r** switch requires the **-o** switch.

If the **-r** and the **-s** switches are used together, the **-s** switch is ignored.

If the **-r** and the **-e** switches are used together, the **-e** switch is ignored.

If the **-f** switch is used, the **-B** and **-p** switches are applied as if the object file names appeared in place of the **-f** switch.

The **-d** (data start address) and **-T** (text start address) switches interact as follows:

- If neither the **-d** nor the **-T** switch is used, the data and text start addresses default.
- If the **-d** switch is used without **-T** (that is, if a data start address is specified, but no text start address is specified), then the data start address specified is used, and the text start address defaults.
- If the **-T** switch is used without **-d** (that is, if a text start address is specified, but no data start address is specified), then the specified text start address is used, and the data section starts on the next logical page boundary following the end of the text section.
- If both the **-d** and **-T** switches are used, the specified data and text start addresses are used.

NOTE

Specifying addresses for the text and data sections different from the defaults may preclude the usage of profiling and performance monitoring tools. These tools require a gap between the text and data sections that is at least as long as the text section.

The profiling tools cannot be used on executables with a text section larger than 32 Mb, although such applications can be executed.

Special Symbols

The following symbols have special meanings to **ld860**:

_etext	The next available address after the end of the output section **.text**.
_edata	The next available address after the end of the output section **.data**.
_end	The next available address after the end of the output section **.bss**.

Programs should not use any of these as external symbols.

The symbols described above are those actually seen by **ld860**. Note that C and several other languages prepend an underscore (_) to external symbols defined by the programmer. This means that, for example, you cannot use **end** as an external symbol. If you use any of these names, you must limit its scope by using the **static** keyword in the declaration or declare the symbol to be local to the function in which it is used. If this is not possible, you will have to use another name.

LD860 *(cont.)*

LD860 *(cont.)*

See Also

ar860, as860, dump860, icc, if77, nm860, size860, strip860

MAC860 MAC860

mac860, mac: Macro preprocessor for the Paragon(TM) system.

Cross-Development Syntax

mac860 [*switches*] *sourcefile*

Native Syntax

mac [*switches*] *sourcefile*

Arguments

sourcefile Source file containing assembler and macro preprocessor commands.

You may specify the following *switches* in any order:

-D*sym=val* Defines *sym* as a local symbol with the value *val* in the macro preprocessor.

-I*incfile* Includes the file *incfile* before the first statement of *sourcefile*. You can use at most one **-I** switch in a single **mac860** command.

-o*objfile* Sets the output file name to *objfile* (the default is the name of the *sourcefile* with any *.s* suffix removed and *.mac* appended).

-V Displays the tool banner (tool name, version, etc.).

-y Makes the macro preprocessor output special directives that the assembler can use for better reporting of line numbers in the source file when errors are detected.

Description

The **mac860** command preprocesses the specified *sourcefile* with the macro preprocessor and produces a source file ready to be assembled with **as860**.

See Also

as860, ar860, dump860, ld860, nm860, size860, strip860

NM860 NM860

nm860, nm: Displays symbol table information for Paragon(TM) system object files.

Cross-Development Syntax

nm860 [*switches*] *filename* ...

Native Syntax

nm [*switches*] *filename* ...

Arguments

filename The name of the object file or library.

You may specify the following *switches* in any order:

-d Display numbers in decimal.

-e Display external relocatable symbols only.

-f Display all symbols, including redundant symbols. Overrides **-e**.

-h Suppress headers.

-n Sort symbols by name.

-o Display numbers in octal.

-p Use short form output. (See "Description" section.)

-r Prepend the current file name to symbols.

-T Truncate symbol names to 19 characters, plus an asterisk to indicate truncation.

-u Display a list of undefined symbols.

-v Sort symbols by value.

-V Display the tool banner (tool name, version, etc.).

-x Display numbers in hexadecimal (default).

NM860 *(cont.)* NM860 *(cont.)*

Description

Use **nm860** to display the symbol tables of the named file(s).

For each symbol in the output of the **-p** switch, one of the following characters identifies its type:

a	Absolute.
b	BSS section symbol.
c	Common symbol.
d	Data section symbol.
f	File tag.
r	Register symbol.
s	Other symbol.
t	Text section symbol.
u	Undefined.

In addition, the characters associated with local symbols appear in lowercase and the characters associated with external symbols appear in uppercase.

When using the **-v** or **-n** switches (sort by value or name, respectively), the scoping information is jumbled, so it is advisable to use the **-e** (externals only) switch.

See Also

as860, ar860, dump860, icc, if77, ld860, size860, strip860

SIZE860 SIZE860

size860, size: Displays section sizes of Paragon(TM) system object files.

Cross-Development Syntax

size860 [*switches*] *filenames*

Native Syntax

size [*switches*] *filenames*

Arguments

filename The name of the object file.

You may specify the following *switches* in any order:

-d Display sizes in decimal (default).

-f Full output.

-n Display the sizes of non-loading sections, as well.

-o Display sizes in octal.

-V Display the tool banner (tool name, version, etc.).

-x Display sizes in hexadecimal.

Description

Use **size860** to display the section sizes of the named files.

Note that the total size of an executable object may be greater than or less than the total of the sizes of all the compiled objects that make up the executable. This is because the true size of the BSS section is not known until after a set of objects is loaded, and because padding is done by **ld860** on other sections.

SIZE860 *(cont.)*

SIZE860 *(cont.)*

See Also

as860, ar860, dump860, icc, if77, ld860, nm860, strip860

STRIP860 STRIP860

strip860, strip: Strips symbol information from Paragon(TM) system object files.

Cross-Development Syntax

strip860 [*switches*] *filename ...*

Native Syntax

strip [*switches*] *filename ...*

Arguments

filename The name of the target object file.

You may specify the following *switches* in any order:

-l Strip line number information only.

-r Do not strip static, external, or relocation information.

-V Display the tool banner (tool name, version, etc.).

Description

Use **strip860** to strip symbol information from object files.

The default is to strip all symbols. This is generally only acceptable for executables.

See Also

as860, ar860, dump860, icc, if77, ld860, nm860, size860

ABORT() ABORT()

Terminates caller abruptly; writes memory image to core file.

Synopsis

SUBROUTINE **ABORT**()

Discussion

Cleans up the I/O buffers and then aborts, producing a core file in the current directory.

ACCESS() ACCESS()

Determines access mode or existence of a file.

Synopsis

INTEGER FUNCTION **ACCESS**(*fil*, *mode*)

CHARACTER*(*) *fil*
CHARACTER*(*) *mode*

Return Value

Returns zero if the file exists and is accessible in the ways specified by *mode*. Returns a nonzero error code if the *mode* argument is incorrectly formatted or the file does not exist or is not accessible in all the ways specified by *mode*.

Description of Parameters

fil Pathname of the file to check.

mode Access modes to check. May include, in any order and in any combination, one or more of the following letters:

r	Test for read permission.
w	Test for write permission.
x	Test for execute permission.
(blank)	Test for existence.

ALARM() ALARM()

Executes a subroutine after a specified time.

Synopsis

INTEGER FUNCTION **ALARM**(*time*, *proc*)

INTEGER *time*
EXTERNAL *proc*

Return Value

Time remaining on previous alarm, if any.

Description of Parameters

time	Length of time until alarm, in seconds, or 0 to turn off the alarm.
proc	Name of procedure to call after *time* seconds.

Discussion

Establishes subroutine *proc* to be called after *time* seconds. If *time* is 0, the alarm is turned off and no routine will be called. The return value of **alarm()** is the time remaining on the last alarm.

BESJ0() BESJ0()

besj0(), besj1(), besjn(), besy0(), besy1(), besyn(), dbesj0(), dbesj1(), dbesjn(), dbesy0(), dbesy1(), dbesyn(): Bessel functions.

Synopsis

REAL FUNCTION **BESJ0**(*x*)

REAL *x*

REAL FUNCTION **BESJ1**(*x*)

REAL *x*

REAL FUNCTION **BESJN**(*n*, *x*)

INTEGER *n*
REAL *x*

REAL FUNCTION **BESY0**(*x*)

REAL *x*

REAL FUNCTION **BESY1**(*x*)

REAL *x*

REAL FUNCTION **BESYN**(*n*, *x*)

INTEGER *n*
REAL *x*

BESJ0() *(cont.)* BESJ0() *(cont.)*

DOUBLE PRECISION FUNCTION **DBESJ0**(*x*)

DOUBLE PRECISION *x*

DOUBLE PRECISION FUNCTION **DBESJ1**(*x*)

DOUBLE PRECISION *x*

DOUBLE PRECISION FUNCTION **DBESJN**(*n*, *x*)

INTEGER *n*
DOUBLE PRECISION *x*

DOUBLE PRECISION FUNCTION **DBESY0**(*x*)

DOUBLE PRECISION *x*

DOUBLE PRECISION FUNCTION **DBESY1**(*x*)

DOUBLE PRECISION *x*

DOUBLE PRECISION FUNCTION **DBESYN**(*n*, *x*)

INTEGER *n*
DOUBLE PRECISION *x*

Discussion

These functions calculate Bessel functions of the first and second kinds for real and double precision arguments and integer orders.

CHDIR() CHDIR()

Changes default directory.

Synopsis

INTEGER FUNCTION **CHDIR**(*path*)

CHARACTER*(*) *path*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

path	Pathname of new current directory.

Discussion

Changes the default directory for creating and locating files to *path*.

See Also

getcwd()

CHMOD() CHMOD()

Changes protection mode of a file.

Synopsis

INTEGER FUNCTION **CHMOD**(*nam*, *mode*)

CHARACTER*(*) *nam*
INTEGER *mode*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

nam	Pathname of the file to change.
mode	File's new protection mode.

Discussion

Changes the protection mode of file *nam* to *mode*. See **chmod(2)** in the *OSF/1 Programmer's Reference* for information on the *mode* parameter.

CTIME() CTIME()

Returns system time as a string.

Synopsis

CHARACTER*(*) FUNCTION **CTIME**(*stime*)

INTEGER *stime*

Return Value

ASCII representation of the time indicated by *stime*, in the following format:

```
Day Mon DD HH:MM:SS YYYY
```

For example:

```
Mon Aug 31 16:02:05 1992
```

The string does not end with a newline or a null character.

Description of Parameters

stime An integer representing a time in seconds since 00:00:00 GMT, January 1, 1970, as returned by **time()**.

Discussion

Converts the system time *stime* to its ASCII form and returns the converted form.

Example

```
      program main
      integer stime, time
      character*24 cur_time, ctime
c
c get the system time
c
      stime = time()
c
```

DV_ACOS() *(cont.)* DV_ACOS() *(cont.)*

```
c convert the system time using ctime
c
      cur_time = ctime(stime)
c
      write(6,10)cur_time
10    format(A)

      call exit
      end
```

The program prints the converted system time. For example:

```
Thu Oct 21 08:32:38 1993
```

See Also

date(), fdate(), gmtime(), idate(), itime(), ltime(), time()

DATE() DATE()

Returns system date as a string.

Synopsis

CHARACTER*(*) FUNCTION **DATE**(*buf*)

CHARACTER*(*) *buf*

Return Value

The current date, as a string in the form *dd-mmm-yy*. The string does not end with a newline or a null character.

Description of Parameters

buf Character array (at least 9 bytes) that receives a copy of the returned string.

Example

```
      program main
      character*9 today, date
c
c get today's date
c
      today = date()
c
      write(6,10)today
10    format(A)

      call exit
      end
```

The program prints the current date. For example:

```
21-Oct-93
```

See Also

ctime(), fdate(), gmtime(), idate(), itime(), ltime(), time()

DV_ACOS() DV_ACOS()

dv_acos(), dv_asin(), dv_atan(), dv_atan2(), dv_cos(), dv_div(), dv_exp(), dv_log(), dv_log10(), dv_pow(), dv_recp(), dv_rsqrt(), dv_sin(), dv_sqrt(), dv_tan(): Perform mathematical operations on **double precision** vectors.

Synopsis

SUBROUTINE **DV_ACOS**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_ASIN**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_ATAN**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

DV_ACOS() *(cont.)* DV_ACOS() *(cont.)*

SUBROUTINE **DV_ATAN2**(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *y*(*)
INTEGER *incy*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_COS**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_DIV**(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *y*(*)
INTEGER *incy*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_EXP**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

DV_ACOS() *(cont.)* DV_ACOS() *(cont.)*

SUBROUTINE **DV_LOG**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_LOG10**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_POW**(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *y*(*)
INTEGER *incy*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_RECP**(*n*, *alpha*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *alpha*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

DV_ACOS() *(cont.)*

DV_ACOS() *(cont.)*

SUBROUTINE **DV_RSQRT**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_SIN**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_SQRT**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

SUBROUTINE **DV_TAN**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
DOUBLE PRECISION *x*(*)
INTEGER *incx*
DOUBLE PRECISION *z*(*)
INTEGER *incz*

DV_ACOS() *(cont.)* DV_ACOS() *(cont.)*

Description of Parameters

n	The number of elements in the vectors *x*, *y*, and *z*.
x, *y*	Input (argument) vectors.
z	Output (result) vector.
incx, *incy*, *incz*	The strides (increments) of vectors *x*, *y*, and *z*, respectively (may be zero).
alpha	A scalar multiplier for **dv_recp**.

Discussion

These subroutines, called the *vector intrinsics*, perform the following mathematical operations on arrays (vectors) very efficiently. You can specify the number of vector elements and the strides of each input vector and the result vector.

dv_acos()	Vector arccosine ($z(i) = \mathbf{acos}(x(i))$).
dv_asin()	Vector arcsine ($z(i) = \mathbf{asin}(x(i))$).
dv_atan()	Vector arctangent ($z(i) = \mathbf{atan}(x(i))$).
dv_atan2()	Vector arctangent from two arguments ($z(i) = \mathbf{atan2}(x(i), y(i))$).
dv_cos()	Vector cosine ($z(i) = \mathbf{cos}(x(i))$).
dv_div()	Non-IEEE vector divide ($z(i) = y(i)/x(i)$).
dv_exp()	Vector exponential ($z(i) = \mathbf{exp}(x(i))$).
dv_log()	Vector natural log ($z(i) = \mathbf{log}(x(i))$).
dv_log10()	Vector logarithm $\log_{10}$ ($z(i) = \mathbf{log10}(x(i))$).
dv_pow()	Vector power ($z(i) = x(i)^{y(i)}$).
dv_recp()	Non-IEEE reciprocal times a scalar ($z(i) = alpha/x(i)$).
dv_rsqrt()	Non-IEEE vector reciprocal square root ($z(i) = 1/\mathbf{sqrt}(x(i))$).

DV_ACOS() *(cont.)* DV_ACOS() *(cont.)*

dv_sin() Vector sine ($z(i)$ = **sin**($x(i)$)).

dv_sqrt() Non-IEEE vector square root ($z(i)$ = **sqrt**($x(i)$)).

dv_tan() Vector tangent ($z(i)$ = **tan**($x(i)$)).

NOTE

To use these calls, you must link your program with the switch **-lvect.**

Example

The following call to **dv_cos()** performs a double-precision vector cosine of the first *n* elements of the **double precision** vector *x* with stride *incx*, storing the results in the **double precision** vector *z* with stride *incz*:

```
call dv_cos(n, x, incx, z, incz)
```

It is similar in effect to the following code (the actual code for **dv_cos()** is written in assembler):

```
      ix = 1
      iz = 1
      if (incx .lt. 0) ix = (-n+1)*incx + 1
      if (incz .lt. 0) iz = (-n+1)*incz + 1
      do 10 i = 1, n
         z(iz) = cos(x(ix))
         ix = ix + incx
         iz = iz + incz
10    continue
```

See Also

sv_acos()

ERF() ERF()

erf(), **erfc()**, **derf()**, **derfc()**: Error functions.

Synopsis

REAL FUNCTION **ERF**(x)

REAL x

REAL FUNCTION **ERFC**(x)

REAL x

DOUBLE PRECISION FUNCTION **DERF**(x)

DOUBLE PRECISION x

DOUBLE PRECISION FUNCTION **DERFC**(x)

DOUBLE PRECISION x

Discussion

erf() and **derf()** return the error function of x.

erfc() and **derfc()** return 1.0–**erf**(x) and 1.0–**derf**(x), respectively.

ETIME() ETIME()

etime(), **dtime()**: Gets elapsed CPU time.

Synopsis

REAL FUNCTION **ETIME**(*tarray*)

REAL *tarray*(2)

REAL FUNCTION **DTIME**(*tarray*)

REAL *tarray*(2)

Return Value

etime() returns the total processor run-time in seconds for the calling process.

dtime() ("delta time") returns the processor time since the previous call to **dtime()**. The first time it is called, it returns the processor time since the start of execution.

Description of Parameters

tarray An array into which is stored the user time (first element) and system time (second element) for the calling process. The returned value is the sum of these two times.

See Also

times()

EXIT() EXIT()

Terminates program with status.

Synopsis

SUBROUTINE **EXIT**(*s*)

INTEGER *s*

Description of Parameters

s	Exit status.

Discussion

Flushes and closes all of the program's files, and returns the value of *s* to the parent process.

See Also

wait()

FDATE() FDATE()

Returns system date and time as a string.

Synopsis

CHARACTER*(*) FUNCTION **FDATE**()

Return Value

The current date, as a string in the form *ddd mmm nn hh:mm:ss yyyy* (for example, `Mon Nov  9 10:48:45 1992`). The string does not end with a newline or a null character.

Example

```
      program main
      character*24 today, fdate
c
c get today's date
c
      today = fdate()
c
      write(6,10)today
10    format(A)

      call exit
      end
```

The program prints the current date and time. For example:

```
Thu Oct 21 09:30:16 1993
```

See Also

ctime(), date(), gmtime(), idate(), itime(), ltime(), time()

FGETC() FGETC()

Gets a character from a logical unit.

Synopsis

INTEGER FUNCTION **FGETC**(*lu*, *ch*)

INTEGER *lu*
CHARACTER*(*) *ch*

Return Value

Returns zero if successful; returns -1 for end-of-file; any other nonzero value is an error code.

Description of Parameters

lu	Logical unit to read from.
ch	Variable into which is stored the next character from *lu*.

Discussion

Stores the next character from the file connected to the logical unit *lu* into the variable *ch*, bypassing normal Fortran I/O statements. If successful, returns zero; a return value of -1 indicates that end-of-file was detected. Any other value is an error code.

NOTE

This routine bypasses normal Fortran I/O.

If normal Fortran I/O is also performed on logical unit *lu*, the results are unpredictable.

See Also

getc()

FLMIN() FLMIN()

flmin(), **flmax()**, **ffrac()**, **dflmin()**, **dflmax()**, **dffrac()**, **inmax()**: Range functions.

Synopsis

REAL FUNCTION **FLMIN**()

REAL FUNCTION **FLMAX**()

REAL FUNCTION **FFRAC**()

DOUBLE PRECISION FUNCTION **DFLMIN**()

DOUBLE PRECISION FUNCTION **DFLMAX**()

DOUBLE PRECISION FUNCTION **DFFRAC**()

INTEGER FUNCTION **INMAX**()

Return Value

flmin() returns the minimum single-precision value.

flmax() returns the maximum single-precision value.

ffrac() returns the smallest positive single-precision value.

dflmin() returns the minimum double-precision value.

dflmax() returns the maximum double-precision value.

dffrac() returns the smallest positive double-precision value.

inmax() returns the maximum integer.

FLUSH() FLUSH()

Flushes a logical unit.

Synopsis

SUBROUTINE **FLUSH**(*lu*)

INTEGER *lu*

Description of Parameters

lu	Logical unit to flush.

Discussion

Flushes the contents of the buffer associated with logical unit *lu* to the corresponding file or device.

FORK() FORK()

Creates a child process.

Synopsis

INTEGER FUNCTION **FORK**()

Return Value

If successful, returns the process ID of the child (new process) to the parent (calling process) and zero to the child. If unsuccessful, returns a negative value that is the negation of the system error code.

Discussion

Creates a copy of the calling process. After the call to **fork()**, both processes can examine the returned value to determine whether they are the copy or the original.

FPUTC() FPUTC()

Writes a character to a logical unit.

Synopsis

INTEGER FUNCTION **FPUTC**(*lu*, *ch*)

INTEGER *lu*
CHARACTER*(*) *ch*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

lu	Logical unit to write to.
ch	Character to write.

Discussion

Writes the character *ch* to the file connected to logical unit *lu*, bypassing normal Fortran I/O.

NOTE

This routine bypasses normal Fortran I/O.

If normal Fortran I/O is also performed on logical unit *lu*, the results are unpredictable.

See Also

outstr(), **putc()**

FREE() FREE()

Frees memory allocated by **malloc()**.

Synopsis

SUBROUTINE **FREE**(*p*)

INTEGER *p*

Description of Parameters

p	Address of the block of memory to free.

Discussion

Deallocates the block of memory whose address is *p*. The block of memory specified by *p* must have been allocated by **malloc()**.

See Also

malloc()

FSEEK() FSEEK()

Positions file pointer.

Synopsis

INTEGER FUNCTION **FSEEK**(*lu*, *offset*, *from*)

INTEGER *lu*
INTEGER *offset*
INTEGER *from*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

lu Logical unit to seek on.

offset New position of the file pointer, expressed as an offset in bytes from the position specified by *from*.

from One of the following values:

0 Beginning of the file.

1 Current position.

2 End of the file.

Discussion

Repositions the read/write file pointer in the file connected to logical unit *lu*.

See Also

ftell()

FTELL() FTELL()

Determines position of file pointer.

Synopsis

INTEGER FUNCTION **FTELL**(*lu*)

INTEGER *lu*

Return Value

Returns the current position of the read/write file pointer in the file connected to logical unit *lu*, expressed as an offset in bytes from the beginning of the file.

If any error occurs, returns the negation of the system error code.

Description of Parameters

lu	Logical unit to query.

See Also

fseek()

GERROR() GERROR()

Returns latest system error message.

Synopsis

CHARACTER*(*) FUNCTION **GERROR**()

Return Value

Returns the system error message corresponding to the last detected system error.

Example

```
      program main
      character*80 gerror, myerror
c
c cause a runtime error by opening a non-existent file
c with status='old'
c
      open(1,file='garbage',status='old',iostat=istat,error=20)
c
c get error detected
c
20    myerror=gerror()
      print *, myerror
c
      call exit
      end
```

Executing the program produces the following output:

```
No such file or directory
```

See Also

ierrno(), perror()

GETARG() GETARG()

Gets the *n*th command line argument.

Synopsis

SUBROUTINE **GETARG**(*n*, *arg*)

INTEGER *n*
CHARACTER*(*) *arg*

Description of Parameters

n	Argument number.
arg	Variable into which is stored the value of argument *n*.

Discussion

Stores the *n*th command line argument into *arg*. The "zero-th" argument is the command name.

See Also

iargc()

GETC() GETC()

Gets a character from logical unit 5.

Synopsis

INTEGER FUNCTION **GETC**(*ch*)

CHARACTER*(*) *ch*

Return Value

Returns zero if successful; returns -1 for end-of-file; any other nonzero value is an error code.

Description of Parameters

ch	Variable into which is stored the next character from logical unit 5.

Discussion

Stores the next character from the file connected to logical unit 5 into the variable *ch*, bypassing normal Fortran I/O statements.

NOTE

This routine bypasses normal Fortran I/O.

If normal Fortran I/O is also performed on logical unit 5, the results are unpredictable.

See Also

fgetc()

GETCWD() GETCWD()

Gets the pathname of the current working directory.

Synopsis

INTEGER FUNCTION **GETCWD**(*dir*)

CHARACTER*(*) *dir*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

dir	Variable into which is stored the pathname of the current working directory.

See Also

chdir()

GETENV() GETENV()

Gets the value of an environment variable.

Synopsis

SUBROUTINE **GETENV**(*en*, *ev*)

CHARACTER*(*) *en*
CHARACTER*(*) *ev*

Description of Parameters

en	Name of an environment variable.
ev	Variable into which is stored the value of *en*.

Discussion

Checks for the existence of the environment variable *en*. If it does not exist or if its value is not present, *ev* is filled with blanks. Otherwise stores the value of *en*, a string, in *ev*.

See Also

putenv()

GETGID() GETGID()

Gets user's group ID.

Synopsis

INTEGER FUNCTION **GETGID**()

Return Value

Returns the numeric group ID of the user of the process.

See Also

getuid()

GETLOG() GETLOG()

Gets user's login name.

Synopsis

CHARACTER*(*) FUNCTION **GETLOG**()

Return Value

Returns the user's login name, or blanks if the process is running detached from a terminal.

See Also

getuid()

GETPID() GETPID()

Gets calling process's OSF/1 process ID.

Synopsis

INTEGER FUNCTION **GETPID**()

Return Value

Returns the OSF/1 process ID of the current process.

GETUID() GETUID()

Gets user's numeric user ID.

Synopsis

INTEGER FUNCTION **GETUID**()

Return Value

Returns the numeric user ID of the user of the process.

See Also

getgid()

GMTIME() GMTIME()

Formats system time for GMT.

Synopsis

SUBROUTINE **GMTIME**(*stime*, *tarray*)

INTEGER *stime*
INTEGER *tarray*(9)

Description of Parameters

stime An integer representing a time in seconds since 00:00:00 GMT, January 1, 1970, as returned by **time()**.

tarray An array into which is stored numeric representations of the components of *stime*.

Discussion

Dissects the system time, *stime*, into month, day, etc., for GMT and returns it in *tarray*. The elements of *tarray* contain the following values:

tarray(1) Seconds (0 - 59)

tarray(2) Minutes (0 - 59)

tarray(3) Hours (0 - 23)

tarray(4) Day of month (1 - 31)

tarray(5) Month of year (0 - 11)

tarray(6) Year - 1900 (for example, 92 = 1992, 102 = 2002)

tarray(7) Day of week (Sunday = 0)

tarray(8) Day of year (0 - 365)

tarray(9) 1 if Daylight Saving Time is in effect, 0 otherwise

GMTIME() *(cont.)*

GMTIME() *(cont.)*

See Also

ctime(), date(), fdate(), idate(), itime(), ltime(), time()

HOSTNM() HOSTNM()

Gets name of current host.

Synopsis

INTEGER FUNCTION **HOSTNM**(*nm*)

CHARACTER*(*) *nm*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

nm	Variable into which is stored the hostname of the system on which the calling process is running.

IARGC() IARGC()

Returns index of the last command line argument.

Synopsis

INTEGER FUNCTION **IARGC**()

Return Value

Returns the index of the last command line argument, which is also the number of arguments after the command name.

See Also

getarg()

IDATE() IDATE()

Returns current system date in numerical form.

Synopsis

SUBROUTINE **IDATE**(*im*, *id*, *iy*)

INTEGER *im*, *id*, *iy*

Description of Parameters

im	Variable into which is stored the current month (a value from 1 to 12 inclusive).
id	Variable into which is stored the current day of the month (a value from 1 to 31 inclusive).
iy	Variable into which is stored the he last two digits of the current year (a value from 0 to 99 inclusive).

Discussion

Stores numeric representations of the current date into *im*, *id*, and *iy*.

See Also

ctime(), **date()**, **fdate()**, **gmtime()**, **itime()**, **ltime()**, **time()**

IERRNO() IERRNO()

Returns latest system error number.

Synopsis

INTEGER FUNCTION **IERRNO**()

Return Value

Returns the system error number of the last detected system error.

See Also

perror(), gerror()

IOINIT() IOINIT()

Initializes I/O.

Synopsis

SUBROUTINE **IOINIT**(*cctl*, *bzro*, *apnd*, *prefix*, *vrbose*)

INTEGER *cctl*
INTEGER *bzro*
INTEGER *apnd*
CHARACTER*(*) *prefix*
INTEGER *vrbose*

Discussion

Currently, no action is performed.

ISATTY() ISATTY()

Determines if logical unit is a TTY.

Synopsis

LOGICAL FUNCTION **ISATTY**(*lu*)

INTEGER *lu*

Return Value

Returns **.TRUE.** if logical unit *lu* is connected to a terminal; otherwise, returns **.FALSE.**.

Description of Parameters

lu	Logical unit number to check.

ITIME() ITIME()

Returns current system time in numerical form.

Synopsis

SUBROUTINE **ITIME**(*iarray*)

INTEGER *iarray*(3)

Description of Parameters

iarray Array into which is stored a numeric representation of the current time:

iarray(1) Current hour (0-23).

iarray(2) Current minute (0–59).

iarray(3) Current second (0–59).

Discussion

Stores the current time into the array *iarray*.

See Also

ctime(), **date()**, **fdate()**, **gmtime()**, **idate()**, **ltime()**, **time()**

KILL() KILL()

Sends a signal to a process.

Synopsis

INTEGER FUNCTION **KILL**(*pid*, *sig*)

INTEGER *pid*
INTEGER *sig*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

pid	Process ID to signal.
sig	Signal number.

Discussion

Sends signal number *sig* to the process whose process ID is *pid*. See **kill(2)** in the *OSF/1 Programmer's Reference* for more information on the *pid* and *sig* parameters.

NOTE

To kill all the processes in the current application, call **kill(0,9)**.

See Also

signal()

LINK() LINK()

Makes a link.

Synopsis

INTEGER FUNCTION **LINK**(*n1*, *n2*)

CHARACTER*(*) *n1*
CHARACTER*(*) *n2*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

n1	Pathname of an existing file.
n2	Pathname for the new link.

Discussion

Creates a link, *n2*, to an existing file, *n1*.

See Also

symlnk()

LNBLNK() LNBLNK()

Returns index of last non-blank in a string.

Synopsis

INTEGER FUNCTION **LNBLNK**(*a1*)

CHARACTER*(*) *a1*

Return Value

Returns the index of the last non-blank character in the string *a1*.

Description of Parameters

a1	Any string.

See Also

rindex()

LOC() LOC()

Returns the address of an object.

Synopsis

INTEGER FUNCTION **LOC**(*a*)

INTEGER *a*

Return Value

Returns the address of *a*.

Description of Parameters

a	Any variable.

LTIME() LTIME()

Formats system time for local time zone.

Synopsis

SUBROUTINE **LTIME**(*stime*, *tarray*)

INTEGER *stime*
INTEGER *tarray*(9)

Description of Parameters

stime	An integer representing a time in seconds since 00:00:00 GMT, January 1, 1970, as returned by **time()**.
tarray	An array into which is stored numeric representations of the components of *stime*.

Discussion

Dissects the system time, *stime*, into month, day, etc., for the local time zone and returns it in *tarray*. The elements of *tarray* contain the following values:

tarray(1)	Seconds (0 - 59)
tarray(2)	Minutes (0 - 59)
tarray(3)	Hours (0 - 23)
tarray(4)	Day of month (1 - 31)
tarray(5)	Month of year (0 - 11)
tarray(6)	Year - 1900 (for example, 92 = 1992, 102 = 2002)
tarray(7)	Day of week (Sunday = 0)
tarray(8)	Day of year (0 - 365)
tarray(9)	1 if Daylight Saving Time is in effect, 0 otherwise

LTIME() *(cont.)*

LTIME() *(cont.)*

See Also

ctime(), date(), fdate(), gmtime(), idate(), itime(), time()

MALLOC() MALLOC()

Allocates memory.

Synopsis

INTEGER FUNCTION **MALLOC**(*n*)

INTEGER *n*

Return Value

Address of the new block of memory.

Description of Parameters

n	Size, in bytes, of the new block of memory.

Discussion

Allocates a block of *n* bytes of memory and returns a pointer to the block of memory.

See Also

free()

MVBITS() MVBITS()

Moves bits.

Synopsis

SUBROUTINE **MVBITS**(*src*, *pos*, *len*, *dest*, *posd*)

INTEGER *src*
INTEGER *pos*
INTEGER *len*
INTEGER *dest*
INTEGER *posd*

Description of Parameters

src	Variable containing bits to be moved.
pos	Beginning position within *src* of the bits to be moved.
len	Number of bits to be moved.
dest	Variable to receive bits.
posd	Beginning position within *dest* for the received bits.

Discussion

Moves *len* bits, beginning at position *pos* of argument *src*, to position *posd* of argument *dest*.

OUTSTR() OUTSTR()

Prints a character string to a logical unit.

Synopsis

INTEGER FUNCTION **OUTSTR**(*ch*)

CHARACTER*(*) *ch*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

ch	String to be output.

Discussion

Outputs the character string *ch* to the file connected to logical unit 6, bypassing normal Fortran I/O.

NOTE

This routine bypasses normal Fortran I/O.

If normal Fortran I/O is also performed on logical unit 6, the results are unpredictable.

See Also

fputc(), putc()

PERROR() PERROR()

Prints error message corresponding to current system error code.

Synopsis

SUBROUTINE **PERROR**(*str*)

CHARACTER*(*) *str*

Description of Parameters

str	String to precede system error message.

Discussion

Writes the message indicated by *str*, followed by the message for the last detected system error, to logical unit 0.

See Also

ierrno(), gerror()

PUTC() PUTC()

Writes a character to logical unit 6.

Synopsis

INTEGER FUNCTION **PUTC**(*ch*)

CHARACTER*(*) *ch*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

ch	Character to be output.

Discussion

Writes the character *ch* to the file connected to logical unit 6, bypassing normal Fortran I/O.

NOTE

This routine bypasses normal Fortran I/O.

If normal Fortran I/O is also performed on logical unit 6, the results are unpredictable.

See Also

fputc(), outstr()

PUTENV() PUTENV()

Changes or adds an environment variable.

Synopsis

INTEGER FUNCTION **PUTENV**(*str*)

CHARACTER*(*) *str*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

str	A string of the form *name=value*.

Discussion

Sets the value of a variable in the process's environment. The argument *str* must contain a character string of the form *name=value*; **putenv()** makes the value of the environment variable *name* equal to *value*.

See Also

getenv()

QSORT() QSORT()

Quick sort.

Synopsis

SUBROUTINE **QSORT**(*array*, *len*, *isize*, *compar*)

DIMENSION *array*(*)
INTEGER *len*
INTEGER *isize*
EXTERNAL *compar*
INTEGER *compar*

Description of Parameters

array	Array to be sorted.
len	Number of elements in the array.
isize	Size of each array element, in bytes.
compar	Integer function that determines the sorting order. This function is called by **qsort()** with two arguments (*arg1* and *arg2*) which are elements of *array*. This function must return a negative value if *arg1* is considered to precede *arg2*, zero if *arg1* is equivalent to *arg2*, or a positive value if *arg1* is considered to follow *arg2*.

Discussion

Sorts the elements of the one-dimensional array *array* according to the comparison function *compar*.

RAND() RAND()

rand(), **irand()**, **srand()**: Random number generator.

Synopsis

DOUBLE PRECISION FUNCTION **RAND**()

INTEGER FUNCTION **IRAND**()

SUBROUTINE **SRAND**(*iseed*)

INTEGER *iseed*

Return Value

rand() returns a pseudo-random double-precision number; **irand()** returns a pseudo-random integer.

Description of Parameters

iseed Seed value used by the random-number generator.

Discussion

rand() generates successive pseudo-random double-precision numbers; **irand()** generates successive pseudo-random integers. **srand()** uses its argument, *iseed*, to re-initialize the seed for successive invocations of **rand()** and **irand()**.

See Also

random()

RANDOM() RANDOM()

random(), **irandm()**, **drandm()**: Random number generator.

Synopsis

REAL FUNCTION **RANDOM**(*flag*)

INTEGER *flag*

INTEGER FUNCTION **IRANDM**(*flag*)

INTEGER *flag*

DOUBLE PRECISION FUNCTION **DRANDM**(*flag*)

INTEGER *flag*

Return Value

A pseudo-random number. Values for **random()** and **drandm()** range from 0.0 to 1.0 inclusive; values for **irandm()** range from 0 to 2147483647 inclusive.

Description of Parameters

flag Zero to generate the next pseudo-random number in the current series, or nonzero to restart the random-number generator.

Discussion

These functions return the next pseudo-random number value of the appropriate type. If the argument *flag* is nonzero, the random number generator is restarted before the next random number is generated.

See Also

rand()

RENAME() RENAME()

Renames a file.

Synopsis

INTEGER FUNCTION **RENAME**(*from*, *to*)

CHARACTER*(*) *from*
CHARACTER*(*) *to*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

from	Pathname of an existing file.
to	New pathname for the file.

Discussion

Renames the file whose pathname is *from* to the new pathname *to*.

RINDEX() RINDEX()

Returns index of substring within a string.

Synopsis

INTEGER FUNCTION **RINDEX**(*a1*, *a2*)

CHARACTER*(*) *a1*
CHARACTER*(*) *a2*

Return Value

Returns the index of the last occurrence of string *a2* in string *a1*.

Description of Parameters

a1	String to search.
a2	String to look for.

See Also

lnblnk()

SECNDS() SECNDS()

secnds(), **dsecnds()**: Returns elapsed time.

Synopsis

REAL FUNCTION **SECNDS**(*x*)

REAL *x*

DOUBLE PRECISION FUNCTION **DSECNDS**(*x*)

DOUBLE PRECISION *x*

Return Value

Returns the elapsed time in units of seconds since midnight, minus the value of *x*.

Description of Parameters

x	Base time.

SIGNAL() SIGNAL()

Establishes signal handler.

Synopsis

INTEGER FUNCTION **SIGNAL**(*signum*, *proc*, *flag*)

INTEGER *signum*
EXTERNAL *proc*
INTEGER *flag*

Return Value

If successful, returns a nonnegative value representing the previous signal handler. Values 0 and 1 represent system signal handlers; a positive value greater than 1 is the address of the subprogram that was the previous signal handler. The returned value can be used to restore the previous signal handler.

If an error occurs, returns the negation of the system error code.

Description of Parameters

signum	Signal number to handle.
proc	Fortran subprogram to use as signal handler.
flag	Any negative value to establish *proc* as a signal handler; any nonnegative value to use one of the system's predefined signal handlers.

Discussion

Establishes *proc* as a signal handler. When the signal *signum* is received, the routine *proc* is called.

If *flag* is negative, *proc* is a Fortran subprogram and is established as the signal handler for the signal. Otherwise, *proc* is ignored and the value of *flag* is passed to the system as the signal action definition. In particular, this how previously saved signal actions can be restored. There are two special cases of *flag*: 0 means "use the default action" and 1 means "ignore this signal." See **signal()** in the *OSF/1 Programmer's Reference* for more information.

SIGNAL()

SIGNAL()

See Also

kill()

SLEEP() SLEEP()

Suspends execution for a period of time.

Synopsis

SUBROUTINE **SLEEP**(*t*)

INTEGER *t*

Description of Parameters

t	Number of seconds to sleep.

Discussion

Suspends the process for *t* seconds.

STAT() STAT()

stat(), lstat(), fstat(): Gets information about a file.

Synopsis

INTEGER FUNCTION **STAT**(*nm*, *statb*)

CHARACTER*(*) *nm*
INTEGER *statb*(*)

INTEGER FUNCTION **LSTAT**(*nm*, *statb*)

CHARACTER*(*) *nm*
INTEGER *statb*(*)

INTEGER FUNCTION **FSTAT**(*lu*, *statb*)

INTEGER *lu*
INTEGER *statb*(*)

Return Value

Returns zero if successful; otherwise, returns -1.

Description of Parameters

nm	Pathname of the file or symbolic link to get information about.
statb	Array into which is stored information about the file.
lu	Logical unit number of the file to get information about.

STAT() *(cont.)* STAT() *(cont.)*

Discussion

These functions store information about a file into the array *statb*. The elements of *statb* contain the following values:

statb(1)	Device file resides on
statb(2)	File serial number
statb(3)	File mode
statb(4)	Number of hard links to the file
statb(5)	User ID of owner
statb(6)	Group ID of owner
statb(7)	Device identifier (special files only)
statb(8)	Total size of file, in bytes
statb(9)	File's last access time
statb(10)	File's last modification time
statb(11)	File's last status change time
statb(12)	Actual number of blocks allocated

stat() obtains information about the file whose name is *nm*; if the file is a symbolic link, information is obtained about the file the link references.

lstat() is similar to **stat()** except **lstat()** returns information about the link.

fstat() obtains information about the file which is connected to logical unit *lu*.

STIME() STIME()

Sets system time.

Synopsis

INTEGER FUNCTION **STIME**(*tp*)

INTEGER *tp*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

tp A time in seconds since 00:00:00 GMT January 1, 1970.

Discussion

Sets the system time and date to the value specified by *tp*.

NOTE

Only the superuser can use this call.

SV_ACOS() SV_ACOS()

sv_acos(), sv_asin(), sv_atan(), sv_atan2(), sv_cos(), sv_div(), sv_exp(), sv_log(), sv_log10(), sv_pow(), sv_recp(), sv_rsqrt(), sv_sin(), sv_sqrt(), sv_tan(): Perform mathematical operations on single-precision **real** vectors.

Synopsis

SUBROUTINE **SV_ACOS**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_ASIN**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_ATAN**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SV_ACOS() *(cont.)* SV_ACOS() *(cont.)*

SUBROUTINE **SV_ATAN2**(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *y*(*)
INTEGER *incy*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_COS**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_DIV**(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *y*(*)
INTEGER *incy*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_EXP**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SV_ACOS() *(cont.)* SV_ACOS() *(cont.)*

SUBROUTINE **SV_LOG**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_LOG10**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_POW**(*n*, *x*, *incx*, *y*, *incy*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *y*(*)
INTEGER *incy*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_RECP**(*n*, *alpha*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *alpha*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SV_ACOS() *(cont.)*

SV_ACOS() *(cont.)*

SUBROUTINE **SV_RSQRT**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_SIN**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_SQRT**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SUBROUTINE **SV_TAN**(*n*, *x*, *incx*, *z*, *incz*)

INTEGER *n*
REAL *x*(*)
INTEGER *incx*
REAL *z*(*)
INTEGER *incz*

SV_ACOS() *(cont.)* SV_ACOS() *(cont.)*

Description of Parameters

n	The number of elements in the vectors *x*, *y*, and *z*.
x, *y*	Input (argument) vectors.
z	Output (result) vector.
incx, *incy*, *incz*	The strides (increments) of vectors *x*, *y*, and *z*, respectively (may be zero).
alpha	A scalar multiplier for **sv_recp**.

Discussion

These subroutines, called the *vector intrinsics*, perform the following mathematical operations on arrays (vectors) very efficiently. You can specify the number of vector elements and the strides of each input vector and the result vector.

sv_acos()	Vector arccosine ($z(i) = \mathbf{acos}(x(i))$).
sv_asin()	Vector arcsine ($z(i) = \mathbf{asin}(x(i))$).
sv_atan()	Vector arctangent ($z(i) = \mathbf{atan}(x(i))$).
sv_atan2()	Vector arctangent from two arguments ($z(i) = \mathbf{atan2}(x(i), y(i))$).
sv_cos()	Vector cosine ($z(i) = \mathbf{cos}(x(i))$).
sv_div()	Non-IEEE vector divide ($z(i) = y(i)/x(i)$).
sv_exp()	Vector exponential ($z(i) = \mathbf{exp}(x(i))$).
sv_log()	Vector natural log ($z(i) = \mathbf{log}(x(i))$).
sv_log10()	Vector logarithm $\log_{10}$ ($z(i) = \mathbf{log10}(x(i))$).
sv_pow()	Vector power ($z(i) = x(i)^{y(i)}$).
sv_recp()	Non-IEEE reciprocal times a scalar ($z(i) = alpha/x(i)$).
sv_rsqrt()	Non-IEEE vector reciprocal square root ($z(i) = 1/\mathbf{sqrt}(x(i))$).

SV_ACOS() *(cont.)*

SV_ACOS() *(cont.)*

sv_sin() Vector sine ($z(i) = \mathbf{sin}(x(i))$).

sv_sqrt() Non-IEEE vector square root ($z(i) = \mathbf{sqrt}(x(i))$).

sv_tan() Vector tangent ($z(i) = \mathbf{tan}(x(i))$).

NOTE

To use these calls, you must link your program with the switch **-lvect.**

Example

The following call to **sv_cos()** performs a single-precision vector cosine of the first *n* elements of the **real** vector *x* with stride *incx*, storing the results in the **real** vector *z* with stride *incz*:

```
call sv_cos(n, x, incx, z, incz)
```

It is similar in effect to the following code (the actual code for **sv_cos()** is written in assembler):

```
      ix = 1
      iz = 1
      if (incx .lt. 0) ix = (-n+1)*incx + 1
      if (incz .lt. 0) iz = (-n+1)*incz + 1
      do 10 i = 1, n
         z(iz) = cos(x(ix))
         ix = ix + incx
         iz = iz + incz
10    continue
```

See Also

dv_acos()

SYMLNK() SYMLNK()

Makes a symbolic link.

Synopsis

INTEGER FUNCTION **SYMLNK**(*n1*, *n2*)

CHARACTER*(*) *n1*
CHARACTER*(*) *n2*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

n1	Pathname of an existing file.
n2	Pathname for the new symbolic link.

Discussion

Creates a symbolic link, *n2*, to an existing file, *n1*.

See Also

link()

SYSTEM() SYSTEM()

Issues a shell command.

Synopsis

INTEGER FUNCTION **SYSTEM**(*str*)

CHARACTER*(*) *str*

Return Value

Exit status of the shell after executing *str*.

Description of Parameters

str	A shell command line.

Discussion

Gives the string *str* to the Bourne shell (**sh**) as input. The current process waits until the shell has completed.

TIME() TIME()

Returns system time.

Synopsis

INTEGER FUNCTION **TIME**()

Return Value

Returns the time since 00:00:00 GMT, January 1, 1970, measured in seconds.

See Also

ctime(), date(), fdate(), gmtime(), idate(), itime(), ltime()

TIMES() TIMES()

Gets process and child process CPU time.

Synopsis

INTEGER FUNCTION **TIMES**(*buff*)

INTEGER *buff*(*)

Return Value

Returns zero if successful; otherwise, returns the negation of the system error code.

Description of Parameters

buff Array that receives time-accounting information for the current process and any terminated child processes, as follows:

buff(1)	User time.
buff(2)	System time.
buff(3)	User time of children.
buff(4)	System time of children.

Discussion

Stores the time-accounting information for the current process and for any terminated child processes of the current process into the array *buff*.

See Also

etime()

TTYNAM() TTYNAM()

Gets pathname of a terminal.

Synopsis

CHARACTER*(*) FUNCTION **TTYNAM**(*lu*)

INTEGER *lu*

Return Value

Returns the blank-padded pathname of the terminal device connected to the logical unit *lu*. If *lu* is not connected to a terminal, blanks are returned.

Description of Parameters

lu	Logical unit to check.

Example

```
      program main
      character*10 mytty, ttynam
      integer lu
c
c get ttyname
c
      lu = 5
      mytty = ttynam(lu)
c
      write(6,10)mytty
10    format(A)
c
      call exit
      end
```

The program prints the name of the terminal device connected to logical unit 5. For example:

```
/dev/ttyp1
```

UNLINK() UNLINK()

Removes a file.

Synopsis

INTEGER FUNCTION **UNLINK**(*fil*)

CHARACTER*(*) *fil*

Return Value

Returns zero if successful; otherwise, returns a nonzero error code.

Description of Parameters

fil Pathname of the file to remove.

Discussion

Removes the file specified by the pathname *fil*.

WAIT() WAIT()

Waits for child process to terminate.

Synopsis

INTEGER FUNCTION **WAIT**(*st*)

INTEGER *st*

Return Value

Returns the OSF/1 process ID of the last child to terminate. If an error occurs, returns the negation of the system error code.

Description of Parameters

st	Variable into which is stored the exit status of the child whose process ID is returned.

Discussion

wait() causes its caller to be suspended until a signal is received or one its child processes terminates. If any child has terminated since the last **wait()**, return is immediate. If there are no child processes, return is immediate with an error code.

If the return value is positive, it is the process ID of the child and *st* is its termination status. If the return value is negative, it is the negation of an error code.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S