May 1995
Order Number: 312490-003

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

Organization

Chapter 1	Introduces the Paragon system software development environment and shows how to create executable files from C source code. This chapter contains enough information to get you started creating executable files for the Paragon system.
Chapter 2	Describes **icc**, the command for compiling, assembling, and linking C 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 C functions that are callable from Fortran and how to call Fortran routines from C.
Chapter 6	Describes the language that the C compiler accepts (ANSI C), extensions to the standard language, and considerations for porting programs written in original C (the language described by Kernighan and Ritchie in *The C Programming Language*).
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 — Describes the internal structure of the compiler, with special emphasis on the vectorizer and optimizer.

Appendix C — Contains reference manual pages for the Paragon system software development commands.

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 icc Driver

Chapter 3
Optimizing Programs

Chapter 4
Using the Inliner

Chapter 5
Interfacing Fortran and C

Chapter 6
Extensions to Standard C

Appendix A
Compiler Error Messages

Appendix B
Compiler Internal Structure

Appendix C
Manual Pages

List of Illustrations

List of Tables

Getting Started 1

This chapter introduces the Paragon™ System software development environment and shows how to create executable files from C source code.

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

The Software Development Environment

The software development environment consists of an Intel supercomputer 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 in a Paragon system 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 for the Paragon system 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 C 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
cpp860	**cpp**	Preprocesses C programs
dump860	**dump860**	Dumps object files
icc	**cc**	Compiles C 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 C.

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 C driver provides an interface to the compiler, assembler, and linker that makes it easy to produce executable files from C 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 node running the operating system.

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

```
% icc myprog.c
```

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

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 C.

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 C.

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

```
% icc myprog.c
```

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 C Calls Reference Manual*.)

The **icc** 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.c* to the executable *a.out*:

  ```
  % icc -nx myprog.c
  ```

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:

```
% icc -node myprog.c
```

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.c* to the executable *a.out*:

```
% icc myprog.c -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 **icc** driver to perform typical tasks. See Chapter 2 for complete information on using the driver and its switches.

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

```
% icc -o x x.c
```

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

```
% icc -nx -o x x.c
```

- Same as above, but include the C math library (**-lm**):

```
% icc -nx -o x x.c -lm
```

- Compile source file *x.c* and link it together with object file *y.o* and library *mylib.a*:

```
% icc -o x x.c y.o mylib.a
```

- Compile and link in *libnx.a*:

```
% icc -o x x.c -lnx
```

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

```
% icc -S x.c
```

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

```
% icc -c x.c
```

- Compile and assemble with optimizations:

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

See Chapter 3 for more information on optimization.

The icc Driver 2

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

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

Invoking the Driver

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

```
icc [switches] source_file...
```

where:

switches Is zero or more of the 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. **icc** bases its processing on the suffixes of the files it is passed:

file.c	is considered to be a C program. It is preprocessed, 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.f or *file*.F — is considered to be a Fortran program. It is passed to the Fortran 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 icc Driver Switches (1 of 2)

-A	Do not inhibit spacing around tokens.
-B	Allow C++-style comments (// to end of line).
-c	Skip link step; compile and assemble only (to *file*.o for each *file*.c).
-C	Preserve comments in preprocessed C source files (implies **-E**).
-D*name*[=*def*]	Define preprocessor symbol *name* to be *def*.
-E	Preprocess every file to *stdout*.
-ES	Equivalent to **-E**.
-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).
-M	Output a list of include files to *stdout*.
-MD	Output a list of include files to *file*.d.
-M*option*	Request special compiler actions (**allow_spacing, alpha, anno, [no]asmkeyword, beta, [no]bounds, clr_reg, concur, cncall, cpp860, [no]dalign, [no]debug, [no]depchk, dollar, extract, fcon, [no]frame, [no]func32, info, inline, keepasm, [no]list, [no]longbranch, neginfo, nostartup, nostddef, nostdinc, nostdlib, [no]perfmon, [no]quad, [no]reentrant, reloc_libs, retain_static, safeptr, [no]signextend, [no]single, [nosplit_loop_ops, [no]split_loop_refs, [no]streamall, [no]stride0, [no]unroll, vect, [no]vintr, [no]xp**).
-nostdinc	Remove the default include directory from the include files search path.
-nx	Create executable application for multiple nodes.

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

-o*file*	Use *file* as name of output file.
-O[*level*]	Set optimization *level* (**0, 1, 2, 3, 4**).
-P	Preprocess only (to *file*.i for each *file*.c).
-r	Generate a relinkable object file (passed to the linker).
-s	Strip symbol table information (passed to the linker).
-S	Skip assemble and link step; compile only (to *file*.s for each *file*.c).
-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**).
-X(a | c | s | t | l | o)	Specify degree of ANSI C conformance.

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*).

-P Skip compile, assemble, and link step; preprocess only (output to *file*.i).

-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.

-B Allow C++-style comments.

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 **icc** manual page in Appendix C for the defaults for these directories. See the **ld860** manual page in Appendix C for more information on the **-YL**, **-YU**, and **-YP** switches.

Preprocess Only

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

-E or **-ES** After preprocessing every input file, regardless of suffix, send the result to *stdout*. No compilation, assembly, or linking is performed.

-C After preprocessing each *file*.c, send the result to *stdout* (like **-E**), but do not remove comments during preprocessing.

-P After preprocessing each *file*.c, send the result to a file named *file*.i.

-A Do not inhibit spacing around tokens. You should not use this option when preprocessing assembly language (*file.s*) files.

-Mallow_spacing
Allow spacing around tokens such as "." and "@" when used with **-ES**.

Preprocess and Compile Only

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

```
-c
```

After assembling each *file*.c, 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:

NOTE

ANSI C predefined macros can be defined and undefined on the command line, but not with **#define** and **#undefine** directives in the source.

-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. (See also the **nostddef** option of the **-M** switch.)

Because all **-D** switches are processed before all **-U** switches, the **-U** switch overrides the **-D** switch.

The **-U** switch affects only *predefined* preprocessor macros, not macros defined in source files. The following macro names are predefined: **__LINE__**, **__FILE__**, **__DATE__**, **__TIME__**, **__STDC__**, **__i860**, **__i860__**, **__PARAGON__**, **__OSF1__**, **__PGC__**, **_PGC_**, **_COFF**, **unix**, **MACH**, **CMU**, **__I860__**, **_I860_**, **__I860**, **_i860_**, **OSF1_ADFS**, **OSF1AD**,and **__NODE** (**__NODE** is only defined when compiling with **-nx** or **-node**). Note that some of these macro names begin and/or end with *two* underscores.

Allow C++ Comments

By default, the driver recognizes and discards only standard C comments (`/* ... */`). The following switch tells the driver to recognize and discard C++ comments (`//` to end of line):

```
-B
```

Specify the Degree of ANSI C Conformance

The **-X** switch lets you control the degree of ANSI C Conformance. The following options are available:

-Xa	ANSI mode. The compiled language conforms to all ANSI features. _ _STDC_ _ is defined to be zero.
-Xc	Conformance mode. The compiled language conforms to ANSI C, but warnings may be produced about some extensions. _ _STDC_ _ is defined to be one. This is the most strict of the ANSI C conformance options. If you want to write full ANSI-conformant code, you should use the **-Xc** option.
-Xs	Pre-ANSI mode. The compiled language includes all features compatible with the C language as defined in *The C Programming Language*, by Kernighan and Ritchie (pre-ANSI C). The compiler warns about all language constructs that differ between ANSI C and pre-ANSI C.
-Xt	Transition mode. This is ANSI C plus pre-ANSI C compatibility extensions without the semantic changes required by ANSI C. Where ANSI C and pre-ANSI C specify different semantics for the same construct, the compiler issues a warning and uses the pre-ANSI C interpretation.
-Xl	Treat [un]signed int and [un]signed long as the same data type. When you use this switch, debug records for [un]signed long are type [un]signed int.
-Xo	Execute the R4.1.1 version of **ic**.

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.
-M	Output a list of include files to *stdout*.
-MD	Output a list of include files to *file*.d.
-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 *option* is passed directly to the compiler, which often removes the need for the **-W0** switch):

allow_spacing Allow spacing around tokens such as "." and "@" when used with **-ES**.

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.

[no]asmkeyword
[Don't] allow the **asm** keyword in C source code (default **-Masmkeyword**). The format is: **asm(s)**.

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**.

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.

cpp860 Direct the internal preprocessor to not compress white space.

[no]dalign [Don't] align **double**s in structures on double-precision boundaries (default **-Mdalign**). **-Mnodalign** may lead to data alignment exceptions.

[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 exist (default **-Mdepchk**). This is especially useful in disambiguating unknown data dependencies between pointers that cannot be resolved at compile time. For example, if two floating point array pointers are passed to a function and the pointers never overlap

and thus never conflict, then this switch may result in better code. The granularity of this switch is rather coarse, and hence the user must use precaution to ensure that other *necessary* data dependencies are not overridden. Do not use this switch if such data dependencies do exist. **-Mnodepchk** may result in incorrect code; the **-Msafeptr** switch provides a less dangerous way to accomplish the same thing.

dollar,*char* Set the character used to replace dollar signs in names to be *char*. Default is an underscore (_).

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

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

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

If both *number*(s) and *function*(s) are specified, then functions 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 functions. The name of the specified *file* must contain a period.

See Chapter 4 for more information on using the compiler's function extractor.

fcon Treat non-suffixed floating point constants as **float**, rather than **double**. This may improve the performance of single-precision code.

[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]func32 [Don't] align functions on 32-byte boundaries (default **-Mfunc32**). **-Mfunc32** may improve cache performance for programs with many small functions.

info=[*option*[**,***option*...]]
Produce useful information on the standard error output. The options 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 functions 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 function inliner. The *options* are:

[**lib:**]*library* Inline functions 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, functions are extracted from a temporary library created during an extract prepass.

[**name:**]*function* Inline the specified function. If **name:** is not used, the function name must not contain a period.

[**size:**]*number* Inline functions containing less than approximately *number* statements.

levels:*number* Perform *number* levels of inlining (default 1).

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

See Chapter 4 for more information on using the compiler's function inliner.

keepasm Keep the assembly file for each C 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 ".c" 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.

nostartup Don't link the usual start-up routine (*crt0.o*), which contains the entry point for the program.

nostddef Don't predefine any system-specific macros to the preprocessor when compiling a C program. (Does not affect ANSI-standard preprocessor macros.) The system-specific predefined macros are **__i860**, **__i860__**, **__PARAGON__**, **__OSF1__**, **__PGC__**, **_PGC_**, **_COFF**, **unix**, **MACH**, **CMU**, **__I860__**, **_I860_**, **__I860**, **_i860_**, **OSF1_ADFS**, **OSF1AD**,and **__NODE** (**__NODE** is only defined when compiling with **-nx**). See also **-U**.

nostdinc Remove the default include directory (*/usr/include* for **cc**, *$(PARAGON_XDEV)/paragon/include* for **icc**) from the include files search path.

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

[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.

[no]reentrant [Don't] generate reentrant code (default **-Mreentrant**). **-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 Cause **-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.

retain_static Do not eliminate static data that is not referenced.

safeptr=[*option*[,*option*...]]
Override data dependence between C pointers and arrays. This is a potentially very dangerous option since the potential exists for code to be generated that will result in unexpected or incorrect results as is defined by ANSI C. However, when used properly, this option has the potential to greatly enhance the performance of the resulting code, especially floating point oriented loops. Combinations of the *options* can be used.

- **dummy** or **arg** C dummy arguments (pointers and arrays) are treated with the same copyin/copyout semantics as Fortran dummy arguments.
- **auto** C local or **auto** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.
- **static** C **static** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.
- **global** C global or **extern** variables (pointers and arrays) are assumed not to overlap or conflict with each other and are independent.

[no]signextend [Don't] sign extend when a narrowing conversion overflows (default **-Msignextend**). For example, if **-Msignextend** is in effect and an integer containing the value 65535 is converted to a **short**, the value of the **short** will be -1. This option is provided for compatibility with other compilers, even though ANSI C specifies that the result of such conversions are undefined. **-Msignextend** will decrease performance on such conversions.

[no]single [Don't] suppress the ANSI-specified conversion of **float** to **double** when passing arguments to a function with no prototype in scope (default **-Mnosingle**). **-Msingle** may result in faster code when single precision is used a lot, but is non-ANSI compliant.

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.

[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**).

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 *option*s are specified, then all vector optimizations are enabled. 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.

cachesize:*number*
Set 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 will 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**).

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).

For include files surrounded by angle brackets (<...>), each **-I** directory is searched, followed by the standard include directory. For include files surrounded by double quotes ("..."), the directory containing the file containing the **#include** directive is searched, followed by the **-I** directories, followed by the standard include directory.

List of Include Files

The following command line switches let you get a list of all the include files used by a source file:

```
-M
-MD
```

The **-M** switch (with no *option*) makes the compiler send to the standard output a list of the pathnames of all files directly or indirectly referenced by **#include** directives in each source file. The **-MD** switch is similar, except that it stores the list of **#include** files for each source file (*file*.c) in a corresponding *file*.d. This information can be useful in writing *makefile*s.

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 C 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 C 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 **float** and **double** 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. If used together with **-lm**, also links in a version of *libm.a* 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 **float** and **double** 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 **icc** driver creates an executable for a single node. The following command line switch creates an executable for multiple nodes:

```
-nx
```

The **-nx** switch has three effects:

- If used while compiling, it defines the preprocessor symbol **__NODE**. The program being compiled can use preprocessor statements such as **#ifdef** to control compilation based on whether or not this symbol is defined.
- If used while linking, it links in *libnx.a*, the library that contains all the calls in the *Paragon™ System C Calls Reference Manual*. It also links in *libmach.a* and *options/autoinit.o*.
- If used while linking, 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 **icc** 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.

C Pragmas

Pragmas 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, pragmas affect only selected functions or loops in the source file. Pragmas allow you to fine tune selected functions or loops.

The general syntax of a pragma is as follows:

#pragma [*scope*] *pragma_body*

scope can be **loop**, **routine**, or **global**.

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

l(loop) Indicates the pragma applies to the next lexical loop. The pragma does not apply to any loops that are enclosed by the next loop. Loop-scoped pragmas are only applied to **do**, **for**, and **while** loops.

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

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

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

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

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

g(global) Indicates the pragma applies to all functions that follow it.

The following rules apply to pragmas **bounds**, **fcon**, and **single**:

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

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

If *scope* is not specified, the default scope for each individual pragma is applied. Table 2-1 lists these defaults.

pragma_body can include any of the pragmas listed in Table 2-1.

Table 2-1 provides a summary of the supported pragmas. The default column specifies the default condition for each pragma. The scope column lists the permitted scopes for each pragma, with the default scope in parentheses. L indicates loop, R indicates routine, and G indicates global scope. The name of a pragma can be preceded by a **-M**. For example, **-Mnoassoc** is equivalent to **noassoc** .

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

PRAGMA	DESCRIPTION	DEFAULT	SCOPE
altcode[*n*]concur	Execute innermost loops without reductions in parallel only if their iteration count exceeds *n*.	*n*=100	(L)RG
altcode[*n*]concurreduction	Execute innermost 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
[no]depchk	[Don't] check for potential data dependencies	**depchk**	(L)RG
[no]fcon	[Don't] assume unsuffixed real constants are single precision	**nofcon**	(R)G
dist=block	Change concurrency characteristics to block	**N/A**	(L)RG
dist=cyclic	Change concurrency characteristics to cyclic	**N/A**	(L)RG
[no]frame	[Don't] set up a complete stack frame	**frame**	(R)G
[no]func32	[Don't] align functions on 32-byte boundaries	**nofunc32**	(R)G
[no]lstval	[Don't] compute last values	**lstval**	(L)RG
opt	Select optimization level	**N/A**	(R)G
[no]recog	[Don't] recognize vector idioms	**recog**	(L)RG
[no]safe	[Don't] treat pointer arguments as safe	**safe**	(R)G

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

PRAGMA	DESCRIPTION	DEFAULT	SCOPE
[no]safeptr	[Don't] ignore potential data dependencies	**nosafeptr**	L(R)G
[no]single	[Don't] suppress the ANSI-specified conversion of **float** to **double** when passing arguments to a function with no prototype in scope	**nosingle**	(R)G
[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

Pragma Descriptions

The following sections provide descriptions of each pragma.

altcode(*n*)concur

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

altcode(*n*)concurreduction

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

(no)assoc

This pragma 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** pragma disables these transformations.

(no)bounds

This pragma alters the effects of the **-Mbounds** command line switch. The **bounds** pragma enables the checking of array bounds when subscripted array references are performed. By default, array bounds checking is not performed. If you use the **bounds** pragma, you must specify **-Mbounds** on the compiler command line.

(no)cncall

This pragma alters the effects of the **-Mcncall** command line switch. The **cncall** pragma 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** pragma, you must specify **-Mconcur** on the compiler command line.

(no)concur

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

(no)depchk

This pragma 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** pragma directs the compiler to ignore these potential data dependencies.

dist=block

Changes the concurrency characteristics to block within the scope of the pragma.

dist=cyclic

Changes the concurrency characteristics to cyclic within the scope of the pragma.

(no)fcon

This pragma alters the effects of the **-Mfcon** command line switch. The **fcon** pragma causes the compiler to treat non-suffixed floating-point constants as **float** rather than **double**. By default, all non-suffixed floating-point constants are treated as **double**.

(no)frame

This pragma alters the effect of the **-Mframe** command line switch. The **frame** pragma causes the compiler to set up a stack frame. By default, the compiler does not set up a stack frame.

(no)func32

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

(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** pragma causes the compiler to not compute last values.

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

opt

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

#pragma [<*scope*>] **opt=**<*level*>

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

(no)recog

This pragma 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** pragma allows selective disabling of vector recognition when the **-Mvect** switch is selected. The **recog** pragma toggles a previous **norecog**.

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

(no)safe

By default, the compiler assumes that all pointer arguments are unsafe and the storage located by the pointer can be accessed by other pointers. The **safe** pragma causes the compiler to consider pointers safe. The **safe** pragma has the following forms:

#pragma [<*scope*>] [**no**]**safe**

#pragma safe (*variable* [, *variable*] ...)

scope can be **routine** or **global**, and *variable* is the name of a pointer variable.

If no variable names are included, all pointer arguments in a routine (routine scope) or in all routines (global scope) will be treated as safe by the compiler. If a single *variable* is specified, the surrounding parentheses can be omitted.

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

(no)safeptr

This pragma alters the effects of the **-Msafeptr** command line switch. The **safeptr** pragma causes the compiler to treat pointer variables if a specified storage class as safe. The **nosafeptr** pragma causes the compiler to treat them as unsafe. The syntax is as follows:

#pragma [<*scope*>] [**no**]**safeptr**=*class*, ...

scope can be **global**, **local**, or **routine**.

class specifies a variable storage class. The classes are **arg**, **local**, **auto**, **global**, **static**, or **all**. The storage classes **local** and **auto** are equivalent.

In a file containing multiple functions, the **-Msafeptr** command line switch might be helpful for one function, but not appropriate for another function because it would produce incorrect results. In such a file, using the **safeptr** pragma with **routine** scope could improve performance and produce correct results.

(no)single

This pragma alters the effects of the **-Msingle** command line switch. The **single** pragma causes the compiler not to convert **float** parameters to **double** parameters in non-prototyped functions. The syntax is as follows:

#pragma [<*scope*>] [**no**]**single**

This can result in faster code if your program uses only **float** parameters, but it results in non-ANSI conformant code.

(no)smallvect

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

#pragma [*scope*] **smallvect**[=*count*]

scope can be **global**, **local**, or **routine**. *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 **smallvect** pragma only affects compilation when the **-Mvect** switch is specified on the command line. 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 pragma is identical to the **(no)smallvect** pragma.

(no)swpipe

The **noswpipe** pragma 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 pragma.

(no)transform

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

(no)vector

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

(no)vintr

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

Pragma Examples

This section presents several examples that illustrate the effects of pragmas and the use of the scope specifiers. During compilation, a pragma either turns a switch on or off, and the pragma only applies to the section of code defined by the scope specified in the pragma. The scope can be the entire file (**global**), the following loop (**loop**), or the current or following routine (**routine**).

The following program is used for the first example:

```
#include "math.h"
main()
{
   float a[100], b[100];
   float x[100][100], y[100][100];
   int i, j;
   for (i=0;i<100;i++)
   {
      a[i]=sin(b[i]);
      for (j=0;j<100;j++)
         x[j][i]=cos(y[j][i]);
   }
}
```

When this program 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:

```
icc -Mvect vect1.c
```

You can use pragmas in the source code to alter this as follows:

```
#include "math.h"
#pragma routine novintr
main()
{
   float a[100], b[100];
   float c[100], d[100];
   int i;
#pragma loop vintr
   for (i=0;i<100;i++)
      a[i]=sin(b[i]);
   for (i=0;i<100;i++)
      c[i]=cos(d[i]);
}
```

In this version of the program, vector intrinsic recognition is disabled for the entire function, since the **novintr** pragma appears before **main( )**. However, the loop-scoped pragma **vintr** appears before the loop containing the reference to **sin( )**, so vector recognition is enabled for only that loop. Since the loop containing the reference to **cos( )** does not have the loop-scoped **vintr** pragma in effect, the vector version of **cos( )** is not recognized.

In the following example, the global **novintr** pragma disables vector intrinsic recognition for the entire file, even if you use the following command line:

```
icc -Mvect vect1.c
```

```
#include "math.h"
#pragma global novintr
main()
{
   float a[100], b[100];
   float x[100][100], y[100][100];
   int i, j;
   for (i=0;i<100;i++)
   {
      a[i]=sin(b[i]);
      for (j=0;j<100;j++)
         x[j][i]=cos(y[j][i]);
   }
}
```

Built-in Math Functions

The compiler now supports the recognition of certain math functions as built-ins. These functions are defined in the file *math.h* with **#define** statements. The **#define** statements are of the form:

#define *routine*(*args*) **__builtin_***routine*(*args*)

routine is the name of a math function, and *args* are the arguments to the function. The following is an example of a **#define** statement that defines the absolute value function as a built-in:

```
#define abs(x)__builtin_abs(x)
```

Having built-in functions provides two benefits:

- Built-in functions allow the vectorizer to recognize vector versions of the functions, if they exist. These vector intrinsics are optimized and provide significant performance improvements for vector operations.
- Built-in functions cause the code for a function to be generated inline, rather than incurring the overhead of a function call.

For functions to be defined as built-ins, the **__PGI** macro must be defined. This macro is defined by default.

The following is a list of the built-in math functions.

abs(*x*)	**fabs**(*x*)	**fabsf**(*x*)
acos(*x*)	**acosf**(*x*)	**asin**(*x*)
asinf(*x*)	**atan**(*x*)	**atanf**(*x*)
atan2(*x*,*y*)	**atan2f**(*x*,*y*)	**cos**(*x*)
cosf(*x*)	**cosh**(*x*)	**coshf**(*x*)
exp(*x*)	**expf**(*x*)	**log**(*x*)
logf(*x*)	**log10**(*x*)	**log10f**(*x*)
pow(*x*,*y*)	**powf**(*x*,*y*)	**sin**(*x*)
sinf(*x*)	**sinh**(*x*)	**sinhf**(*x*)
sqrt(*x*)	**sqrtf**(*x*)	**tan**(*x*)
tanf(*x*)	**tanh**(*x*)	**tanhf**(*x*)

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 Paragon OSF/1 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.

8. 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.

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 3-16). 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 **icc** 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 function calls with inline code.
-Mnodepchk	Ignores potential data dependencies.
-Msafeptr	Overrides data dependence between C pointers and arrays.

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 C 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 C 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:

```
for(j=0; j<1000; j++) {
    for(i=0; i<3; i++) {
        x[i][j] = x[i][j] * a[i][j];
    }
}
```

  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:

```
for(i=0; i<100; i++) {
    s = s + a[i] * b[i];
}
```

  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 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 results 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).

-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:

```
for(i=0; i<100; i++) {
    s = s + a[i] * b[i];
}
```

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

```
s = ((((s + (a[0]*b[0])) + (a[1]*b[1])) + (a[2]*b[2])) + ... )
```

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

```
s = s + (((((a[0]*b[0]) + (a[1]*b[1])) + (a[2]*b[2])) + ... )
```

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:

```
% icc -O4 -Mvect -Knoieee -Minfo=loop -c nas.c
// 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 up to 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 occurred.

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:

```
% icc -c -Knoieee myprog.c
% icc 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:

```
% icc -c myprog.c
% icc -Knoieee myprog.o
```

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

```
% icc -Knoieee myprog.c
```

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 **-lm** 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:

- Using the **-Knoieee** switch when linking with **-lm** replaces the standard **-lm** math library with a compatible non-IEEE version. Many of the routines in this library 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/C 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 function 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 function 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 and -Msafeptr)

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. One of the most common is pointer dereferencing.

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.

The **-Msafeptr=***option* switch causes the compiler to ignore data dependence between C pointers and arrays. This is a potentially very dangerous option since the potential exists for code to be generated that will result in unexpected or incorrect results as is defined by ANSI C. However, when used properly, this option has the potential to greatly enhance the performance of the resulting code, especially floating point oriented loops. Combinations of the *options* can be used. The following are available:

dummy or **arg**	C dummy arguments (pointers and arrays) are treated with the same copyin/copyout semantics as Fortran dummy arguments.
auto	C local or **auto** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.
static	C **static** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.
global	C global or **extern** variables (pointers and arrays) are assumed not to overlap or conflict with each other and are independent.

The following example shows the use of -Msafeptr=global on the command line and the code segment affected.

```
% icc -Minfo=loop -O4 -Msafeptr=global -c safeptrext.c

//Sofware pipelined loop with 5 cycles and 2 columns for line 7.
//Pipe/Dual-instruction 1 column 10-cycle loop gend for line 7.

extern int i, max;
extern float *c, *a, *b;
extern void
```

```
func()
{
   for (i = 0; i < max; i += 1) {
      c[i] = a[i] * b[i];
   }
}
```

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**.

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

```
for(k=0; i<10; k++) {
    for(j=0; j<10; j++) {
        for(i=0; i<2000; i++) {
            •
            •    all loop operations here
            •
        }
    }
}
```

  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** vectors and less than 16 for **float** 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:

```
for(i=0; i<1000; i++) {
    /* code for all conditions */
    if(a > b) {
        /* code for a > b */
    }
}
```

  Rewrite it as follows:

```
if(a > b) {
    for(i=0; i<1000; i++) {
        /* code for all conditions */
        /* code for a > b */
    }
```

```
} else {
    for(i=0; i<1000; i++) {
        /* code for all conditions */
    }
}
```

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 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 function inlining capability.

Function inlining is a compiler optimization under which the body of a function is expanded in place of a call to the function. This can speed up execution by eliminating the parameter passing and function call and return overhead. Inlining a function body also creates opportunities for other compiler optimizations. Inlining will usually result in larger code size (although in the case of very small functions, 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 functions that can be inlined, then replaces calls to those functions with the equivalent code automatically.
- Use of **inliner libraries**. When you use the **-Mextract** switch during compilation, the compiler looks for functions that can be inlined and extracts them into an *inliner library*. Later, when compiling a program that calls functions in the inliner library, you use the **-Minline** switch and specify the library; the compiler replaces calls to the functions in the library with the equivalent code.

Compiler Inline Switch

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

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

where *option* is one of the following:

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

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

[lib:]*library* Specifies a library of inlined functions. If **lib:** is not used, the library name must contain a period. Any number of libraries can be specified. A function 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 function name(s) and a size limit are specified, a function 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 functions 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, functions to be inlined are selected from the specified libraries. If neither function names nor a size limit are specified, any function 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:]*function* Extracts the specified function. **name:** must be used if the function name contains a period.

[size:]*number* Extracts functions 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 functions 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 function can have other functions 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 functions 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 function.

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 functions, 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 **icc** 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 C functions cannot be inlined:

- Functions whose return type is a struct data type, or have a struct argument
- Functions containing switch statements
- Functions that reference a static variable whose definition is nested within the function
- Functions that accept a variable number of arguments

Certain functions can only be inlined into the file that contains their definition:

- Static functions
- Functions that call a static function
- Functions that reference a static variable

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 function is called with the wrong number of arguments, a warning message is issued and the function is not inlined.
- If an inlinable function is called in a context which assumes that a value is returned, but the body of the function 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 function does not match the declaration in the source file being compiled, a severe error is issued.

Examples

This section contains examples of using the inliner.

Dhry

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

```
% icc dhry.c -Minline=Proc7
```

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

```
% icc dhry.c -Minline=Proc7,3
```

The following command line builds an executable for **dhry** in which all functions of roughly ten or fewer statements are inlined. Two levels of inlining will have been performed. This means that if function A calls function 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.

```
% icc dhry.c -Mextract=10 -Minline=10 -o temp.ilib
% icc dhry.c -Minline=temp.ilib
% rm -r temp.ilib
```

Fibo

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

```
% icc fibo.c -c -Minline=fibo -O
```

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

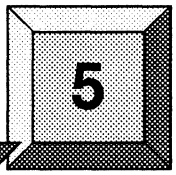
Makefiles

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

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

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

Interfacing Fortran and C



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()**.

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 returned 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**
int *	**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 Standard C 6

This chapter describes the language that the Paragon™ OSF/1 C compiler accepts (ANSI C), extensions to the standard language, and considerations for porting programs written in original C (the language described by Kernighan and Ritchie in *The C Programming Language*).

Standard Language

The standard language is defined in the *American National Standard for Programming Language C (ANS X3.159-1989).*

For additional information on programming in the C language, refer to the following:

- Kernighan, Brian W., and Ritchie, Dennis M., *The C Programming Language*, Prentice Hall, 1978.
- Harbison, Samuel P., and Steele, Guy L., *C: A Reference Manual, Second Edition*, Prentice Hall, 1987.

Instead of fully specifying the language accepted by the compiler, this chapter describes only those features that differ from the C language specified in *The C Programming Language*. Most of the differences (incompatibilities and extensions) are ANSI features.

Extensions

This section lists the extensions to the original C language and, in certain cases, to the ANSI standard, supported by the Paragon OSF/1 C compiler.

1. The **#module** *identifier* directive is supported. The *identifier* is used as the name of the module. If no **#module** directive is present, the name of the input file, without the ".c" suffix, is used.
2. The **#list** and **#nolist** directives are supported. They enable and disable the listing of source code in the listing file.

3. The **#pragma** [*tokens*] ANSI directive is supported. Any pragma that is not recognized is ignored.

4. The **#elif** *expression* ANSI directive is supported. This directive is like a combination of the **#else** and **#if** directives.

5. The **defined** ANSI operator is supported. Both of the following expressions evaluate to `1` if *name* is the name of a macro, or to `0` otherwise:

```
defined(name)
defined name
```

6. The following preprocessor macros are predefined (in addition to the ANSI-standard predefined macros `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, and `__STDC__`):

   - `__I860__`
   - `_I860_`
   - `__I860`
   - `_i860_`
   - `__i860`
   - `__i860__`
   - `__PARAGON__`
   - `__OSF1__`
   - `OSF1_ADFS`
   - `OSF1AD`
   - `__PGC__`
   - `_PGC_`
   - `_COFF`
   - `__NODE` (only defined when compiling with **-nx** or **-node**)
   - `unix`
   - `MACH`
   - `CMU`

   Note that some of these macro names begin and/or end with *two* underscores.

7. The **#ident** directive is supported. The syntax is:

   ```
   #ident "string"
   ```

   For certain assemblers, this results in a **.ident** directive being added to the output file.

8. The *#predicate(value)* extension is supported inside preprocessor **#if** and **#elif** directives. This exists for compatibility with AT&T include files. The compiler driver passes the following predicates to the compiler:

   - **#machine(paragon)**
   - **#lint(off)**
   - **#system(osf1)**
   - **#cpu(i860)**

   Only these predefined predicates exist; you cannot create new predicates.

9. Identifiers may contain the dollar sign character, ($).

10. The ANSI reserved word **void** may be used to indicate the **void** data type (data type with no values). This type is used to indicate that the value of an expression is not used, and to declare functions that return no value. The type **void *** is used to indicate a universal pointer (similar to the old use of **char ***. A **void *** pointer may be quietly converted to and from pointers of other types.

11. Enumeration types are supported. Enumeration constants are implemented as integers. All integer operations are allowed on enumeration types, as per the ANSI standard; thus an enumeration constant has type **int** and enumeration variables are of integral type.

12. Two different structures may contain members with the same name, even when the members have different offsets within each structure. (ANSI)

13. Structures may be assigned, passed as arguments to functions, and returned by functions. (ANSI)

14. The ANSI types **unsigned short int** and **unsigned char** are supported. The keyword **signed** is added as per the ANSI standard. A signed integer type is equivalent to the normal integer type; characters may be specified to be signed by using this keyword. Characters are unsigned by default. The ANSI type **long double** is supported; it is currently implemented the same as **double**.

15. The keywords **const** and **volatile** are supported as per the ANSI standard. Objects of type **const** may not be assigned values. Objects of type **volatile** (objects used for device registers and variables that may change as the result of signals) are immune to optimizations that might change the meaning of the program.

16. ANSI function prototypes are supported. A function declaration may include specification of the types of its parameters. Type conversions are performed as necessary to ensure that the types of actual parameters to such a function match the types of its formal parameters, with error messages issued when appropriate.

17. The new ANSI lexical conventions are supported:

    - Any token may be continued using the "backslash-newline" (\n) conventions.
    - Trigraph sequences are recognized.
    - The letters "u" or "U" may be appended to an integer constant to make it unsigned.
    - The letters "f" or "F" and "l" or "L" may be appended to a floating constant to make it of type **float** or **long double**, respectively.
    - Two or more consecutive string literals are concatenated into one.
    - The "\xZZZ" (hexadecimal) and "\a" (alert) character escape sequences have been added.

18. Initialization of automatic aggregates is allowed as per the ANSI standard. An automatic **struct** may be initialized with an arbitrary structure expression or with a brace-enclosed list of constant expressions. Automatic arrays can only be initialized using a brace-enclosed list of constant expressions. Initialization of a union is allowed by initializing the first element of the union. As in original C, all static variables can be initialized.

19. Both signed and unsigned bit fields are supported as per the ANSI standard.

20. The unary + operator has been added as per the ANSI standard.

21. Data types **signed long** and **unsigned long** are separate data types instead of synonyms for **signed int** and **unsigned int** respectively.

Implementation-Defined Behavior

The sizes and alignments of the various C data types are shown in Table 6-1:

Table 6-1. Sizes and Alignments of Data Types

Type	Size	Alignment
char	1 byte	byte
short	2 bytes	2-byte
int	4 bytes	4-byte
long int	4 bytes	4-byte
float	4 bytes	4-byte
double	8 bytes	8-byte
long double	8 bytes	8-byte
struct	(varies)	Alignment of field with largest alignment
union	(varies)	Alignment of member with largest alignment
array of *type*	*n* * size of *type*	Alignment of *type*

The search rules for **#include** directives are:

- If the pathname is enclosed in angle brackets, the compiler first searches the directories specified with the **-I** command line switch in the order specified, then the system include directory.
- If the pathname is enclosed in double quotes, the compiler first searches the current directory, then follows the search rules above.

Porting Considerations

This section describes incompatibilities between original C and the version of ANSI C supported by the Paragon OSF/1 C compiler. These incompatibilities prevent programs that were legal under the original definition from being accepted by the compiler. In all but the last two cases, the compiler identifies the error and issues a message.

1. The compiler performs strict type-checking. In particular, the base type of a pointer expression used to access a **struct** member must be a structure type that contains a member with that name. (ANSI)

2. Identifier names may be arbitrarily long, but only the first 31 characters are significant (31 is also the ANSI standard). The original definition of C allowed long names but only the first eight characters were significant, implying that misspellings after the eighth character were not errors.

3. Storage class specifiers must come before type specifiers, if both are present (for example, **static int**, not **int static**). The ANSI standard considers placement of the storage class specifier an obsolete feature.

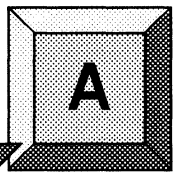
4. If a unary operator is applied to a variable of type **float**, or if a binary operator is applied to two variables of type **float**, the result is computed using single precision arithmetic. This is in accordance with the ANSI standard.

5. No white space (blanks, tabs, comments, or new lines) is allowed between the characters making up the following assignment operator tokens (ANSI):

+=	-=	*=	/=	<<=
>>=	&=	^=	|=	

6. The default numeric conversion rules follow the ANSI convention of *value preserving*. This means that an **unsigned char** or **unsigned short int** is converted to an **int**, rather than an **unsigned int**. The compiler issues no messages for this conversion.

Compiler Error Messages



This appendix lists the error messages generated by the Paragon™ OSF/1 C 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, 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. Also can be issued if include 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.

I006 <reserved message number>

F007 Source file too large to compile at this optimization level

Symbol table overflowed, or compiler working storage space exhausted. If this error occurred at optimization level 2, reducing the optimization level to 1 may work around the problem, otherwise splitting the source file in two should be considered. There is no hard limit on how large a file the compiler can handle, but as a very rough estimate, if the file is less than 2000 lines long (not counting comments), and this error occurs, it may represent a compiler problem.

F008 Error limit exceeded

The compiler gives up after 25 severe errors.

I009 <reserved message number>

I010 <reserved message number>

S011 Unrecognized command line switch: $

Refer to the **icc** manual page for a list of the allowed switches.

S012 Value required for command line switch: $

Certain switches require a value which immediately follows, 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.

```
I015  <reserved message number>
```

```
I016  Identifier, $, truncated to 31 chars
```

An identifier may be at most 31 characters in length; characters after the 31st are ignored.

```
I017  <reserved message number>
```

```
I018  <reserved message number>
```

```
I019  Underflow of real or double precision constant
```

```
I020  Overflow of real or double precision constant
```

```
S021  Input source line too long
```

After macro expansion, a source line must not be more than 3000 characters long. It may be possible to work around the problem by removing unneeded blank characters from certain macro definitions.

```
W022  Char escape does not fit in char
```

The value of a hex escape in a **char** or string constant exceeds the capacity of a **char** (8 bits). The value is truncated.

```
W023  Integer overflow on integer constant: $
```

```
S024  Illegal character constant
```

A character constant was either unterminated or had no characters.

```
S025  Illegal character: $
```

Illegal character encountered in source code. Octal representation of character is given.

S026 Unmatched double quote

S027 Illegal integer constant: $

Integer (or hexadecimal constant) is too large for *32-bit* word.

S028 Illegal real or double precision constant: $

Syntax of constant with exponent is bad.

S029 Syntax error: Recovery attempted by deleting from $

The indicated input was deleted during syntax error recovery.

S030 Syntax error: Malformed $ at $

The indicated construct starting at the indicated token was found to be improperly formed during syntax error recovery.

W031 Multi-character character constant

This error can be caused by an attempt to specify more than one character within single quotes.

S032 Syntax error: Unexpected input at $

The tokens including and following the indicated token caused a syntax error.

W033 Missing declarator for dummy argument

A declaration without a declared identifier appeared in the dummy argument declaration list.

F034 Unrecoverable syntax error reading $

Note that processing of source code is terminated.

S035 Syntax error: Recovery attempted by replacing $ by $

S036 Syntax error: Recovery attempted by inserting $ before $

S037 Syntax error: Recovery attempted by deleting $

```
S038  Illegal combination of standard data types
```

For example, **unsigned double**.

```
S039  Use of undeclared variable $
```

An undeclared variable is treated as an automatic **int**.

```
S040  Illegal use of symbol, $

S041  $ is not an enumeration tag
```

Use of an identifier as an enumeration tag before declaring it.

```
S042  Use of undefined struct or union, $

S043  Redefinition of symbol, $

S044  Redefinition of structure or union tag $

S045  Illegal field size
```

Bit field size must be in range 1 to 32 (0 allowed for unnamed fields).

```
W046  Non-integral array subscript is cast to int

S047  Array dimension less than or equal to zero
```

The number of elements declared for an array must be greater than zero.

```
S048  Illegal nonscalar constant
```

Don't know how user can cause this error.

```
S049  Illegal storage class specifier

S050  Semicolon missing after declaration

S051  Illegal attempt to compute sizeof a function
```

I052 Array dimension not specified. Extern assumed

An array definition such as **int a[];** is treated as the array declaration **extern int a[];**.

S053 Illegal use of void type

S054 Subscript operator ([]) applied to non-array

S055 Illegal operand of indirection operator (*)

S056 Attempt to call non-function

W057 Old-style declaration used; int assumed

A data declaration consisting of just an identifier is used (no type and storage class specified).

S058 Illegal lvalue

Expression on the left hand side of an assignment statement or operand of unary **&** operator is not a legal lvalue.

S059 Struct or union required on left of . or ->

S060 $ is not a member of this struct or union

S061 Sizeof dimensionless array required

An array whose dimensions were not specified is used in a context which requires a computation of its size.

S062 Operand of - must be numeric type

S063 Operand of ~ must be an integer type

W064 Cast expression on LHS of assignment treated as cast type

An expression of the form **(type *)p** = *expr* was found; the left hand side has been treated as if it were ***(type **)&p**.

S065 Break statement not inside loop or switch statement

S066 Continue statement not inside loop

S067 Switch expression must be of integer type

S068 Case or default must be inside switch statement

S069 Dummy parameter specification not allowed here

S070 $ is not a dummy argument

S071 More than one default case for switch

S072 Initializer not allowed in this context

Initializer specified on a dummy parameter, a **typedef** name, or **extern** declaration.

S073 Too many initializers for $

The initializer for an array or structure contains too many constants.

S074 Non-constant expression in initializer

S075 Aggregate initializer used for scalar type

S076 Initializer not allowed for function

S077 Character string too long for array

When initializing an array of characters using a character string constant, the array must be large enough for all the characters or all the characters including the null terminating character.

W078 Character constant too long

A wide character constant contains more than 1 wide character.

W079 Enum value for $ overflows $

V080 Missing braces for array, structure, or union initialization

S081 Array of functions or function returning function not allowed

S082 Function returning array not allowed

S083 Switch case constants must be unique

I084 <reserved message number>

W085 Truncation performed for field initialization

An integer constant used to initialize a structure field is too large for the field.

S086 Division by zero

A division by zero was encountered while constant folding a constant expression.

S087 <reserved message number>

S088 Bit field cannot be the operand of sizeof or &

S089 Array name used in logical expression

S090 Scalar data type required for logical expression

S091 Integer constant expression required

S092 Illegal type conversion of constant required

W093 Type cast required for this conversion of constant

S094 Illegal type conversion required

This message is issued for a number of situations, for example, when the data types of the left and right hand sides of an assignment statement are incompatible.

W095 Type cast required for this conversion

This message is issued for situations such as message 94, except that the compiler has gone ahead and performed the necessary type conversion as if the user had specified a type cast. A typical case is when the left and right hand sides of an assignment statement have different pointer types.

S096 Illegal function arg of type void or function

The actual argument of a function call has an illegal data type.

S097 Statement label $ has been defined more than once

The indicated name is used for more than one label within a function.

S098 Expression of type void * cannot be dereferenced

An attempt was made to apply the unary * operator to a pointer expression of type "pointer to **void**."

W099 Type cast required for this comparison

Comparison of pointers of different types should use a type cast. The compiler has performed the necessary type conversion.

S100 Non-integral operand for mod, shift, or bitwise operator

S101 Illegal operand types for + operator

S102 Illegal operand types for - operator

S103 Illegal operand types for comparison operator

S104 Non-numeric operand for multiplicative operator

W105 Operands of pointer subtraction have different types

Since both operands point to types of the same size, the compiler is able to translate this expression unambiguously.

W106 Shift count out of range

The bit count for a shift operation must be in the range 0 to 31. Note that a shift count of 32 will not produce a result of zero on some machines.

S107 Struct or union $ not yet defined

S108 Unnamed bit fields not allowed in unions

W109 Type specification of field $ ignored

Bit fields must be **int**, **char**, or **short**. Bit field is given the type **unsigned int**.

S110 Bit field $ too large for indicated data type

The size of a bit field exceeds the size of the data type used to declare the field; for example, **char fld:9**.

W111 More than one storage class specified

The additional storage class specifiers are ignored.

W112 Duplicate type modifier

A type modifier is repeated; for example, **const const int x;**.

S113 Label $ is referenced but never defined

W114 More than one type specified

More than one type specifier occurs where at least one of the specifiers is a **typedef**, **struct/union** type, or **enum** type. All but the first type specifier are ignored.

W115 Duplicate standard type

A standard type is repeated; for example, **float float int flt;**.

W116 Constant value out of range for signed short or char

Note that a constant such as **0xFFFF** (**0xff**), interpreted as a positive number, is 1 bit too large for the signed **short** (**char**) data type. Either the type **unsigned short** (**unsigned char**) should be used in place of signed **short** (**char**), or the equivalent negative number should be used in place of the positive constant.

W117 Value missing from return statement in function $

No function value will be returned by this **return** statement.

```
W118  Function $ does not contain a return statement
```

```
W119  void function $ cannot return value
```

The **return** expression is ignored.

```
I120  Label $ is defined but never referenced
```

```
W121  Block with auto initialization jumped into at label $
```

The indicated label was referenced from outside its containing block, and the containing block initialized automatic storage. When such a transfer of control occurs, the automatic initialization does not occur.

```
I122  Value of expression not used
```

This message can result from accidentally typing == where = was intended. As another example, the statement ***p++;** (which is actually equivalent to just **p++;**) will cause the message. Unfortunately, uses of the standard macros **getc** and **putc** will cause this message to be issued because these macros expand to conditional expressions whose values are typically not used by the programmer. In this case, the message can be eliminated by casting the **getc**/**putc** expression to **void**.

```
I123  Definition of function $ is static
```

```
I124  Possible misuse of dummy array $
```

Address of dummy array taken, or assignment to array name.

```
I125  Integer value truncated to fit into unsigned short or char type
```

Using a negative number, or a positive number greater than 16 (8) bits as an **unsigned short** (**unsigned char**) value can cause this message to be issued. Note that such code is nonportable.

```
S126  Parameters cannot follow va_alist
```

```
I127  <reserved message number>
```

```
I128  <reserved message number>
```

```
W129  Floating point overflow. Check constants and constant expressions
```

W130 Floating point underflow. Check constants and constant expressions

W131 Integer overflow. Check floating point expressions cast to integer

S132 Floating pt. invalid oprnd. Check constants and constant expressions

S133 Divide by 0.0. Check constants and constant expressions

W134 Duplicate struct or union member $

A **struct** or **union** member was found with the same name as another member of the same **struct** or **union**.

I135 Function $ should use prototype form of definition

A function that was declared using the prototype form was defined using a non-prototype format. Note that if the function is used after the definition, the prototype does not have an effect.

W136 Function $ has non-prototype declaration in scope

A function is declared using the prototype form, but a declaration or definition for the function that does not use the prototype form is in scope.

S137 Incompatible prototype declaration for function $

A function prototype declaration is incompatible with a previous prototype declaration for that function.

S138 Missing identifier for declarator in function prototype definition

A function declarator in a function prototype was missing an identifier for the formal parameter.

S139 void parameter must be the only parameter

A function prototype of the form **(void, ...)**, **(int, void)**, or **(void, int)** was encountered.

S140 Declaration for formal $ found in prototype function definition

An attempt was made to declare a formal parameter following the function header for a prototype form function definition.

```
S141  Wrong number of parameters to function
```

```
W142  Assignment to const object not allowed
```

An assignment to an object with type modifier **const** was attempted.

```
W143  Useless typedef declaration (no declarators present)
```

typedef declares no declarators; e.g. **typedef int x; typedef int x;** the second **typedef** would give this message. Often occurs with non-ANSI include files (a common culprit is **size_t**).

```
V144  Syntax requires semicolon, semicolon inserted
```

```
V145  Syntax requires no comma, comma deleted
```

```
S146  void parameter cannot have a name ($)
```

```
W147  Inappropriate qualifiers with void
```

const void and **volatile void** are just treated as **void**.

```
S148  Struct/union member $ cannot be a function
```

```
W149  Unnamed struct/union member ignored
```

A member of **struct** or **union** with no declarators was encountered.

```
W150  Useless declaration
```

A declaration does not specify an identifier; e.g., **int; extern;**

```
W158  Use of escape ignored
```

A use of a character escape which is not one of the recognized escapes has occurred; the backslash is ignored.

```
W159  No hex digits follow ignored
```

No hexadecimal digits follow the numeric escape **\x**; the backslash is ignored.

W162 Not equal test of loop control variable $ replaced with < or > test.

W198 Possible conflict ignored between $ and $

W199 Unaligned memory reference

A memory reference occurred whose address does not meet its data alignment requirement.

S201 #elif after #else

A preprocessor **#elif** directive was found after a **#else** directive; only **#endif** is allowed in this context.

S202 #else after #else

A preprocessor **#else** directive was found after a **#else** directive; only **#endif** is allowed in this context.

S203 #if-directives too deeply nested

Preprocessor **#if** directive nesting exceeded the maximum allowed (currently 10).

S204 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).

W205 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.

F206 Can't find include file $

The indicated include file could not be opened.

S207 Definition too long for $

The length of the macro definition of the indicated macro exceeded the maximum allowed (currently 2048).

S208 EOF in comment

The end of a file was encountered while processing a comment.

S209 EOF in macro call to $

The end of a file was encountered while processing a call to the indicated macro.

S210 EOF in string

The end of a file was encountered while processing a quoted string.

S211 Formal parameters too long for $

The total length of the parameters in the definition of the indicated macro exceeded the maximum allowed (currently 2048).

S212 Identifier too long

The length of an identifier exceeded the maximum allowed (currently 2048).

S213 Unable to open dependency file $

W214 Illegal directive name

The sequence of characters following a # sign was not an identifier.

W215 Illegal macro name

A macro name was not an identifier.

S216 Illegal number $

The indicated number contained a syntax error.

F217 Line too long

The input source line length exceeded the maximum allowed (currently 2048).

W218 Missing #endif

End of file was encountered before a required **#endif** directive was found.

W219 Missing argument list for $

A call of the indicated macro had no argument list.

S220 Number too long

The length of a number exceeded the maximum allowed (currently 2048).

W221 Redefinition of symbol $

The indicated macro name was redefined.

I222 Redundant definition for symbol $

A definition for the indicated macro name was found that was the same as a previous definition.

F223 String too long

The length of a quoted string exceeded the maximum allowed (currently 2048).

S224 Syntax error in #define, formal $ not identifier

A formal parameter that was not an identifier was used in a macro definition.

W225 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.

S226 Syntax error in #if

A syntax error was found while parsing the expression following a **#if** or **#elif** directive.

S227 Syntax error in #include

The **#include** directive was not correctly formed.

W228 Syntax error in #line

A **#line** directive was not correctly formed.

W229 Syntax error in #module

A **#module** directive was not correctly formed.

W230 Syntax error in #undef

A **#undef** directive was not correctly formed.

W231 Token after #ifdef must be identifier

The **#ifdef** directive was not followed by an identifier.

W232 Token after #ifndef must be identifier

The **#ifndef** directive was not followed by an identifier.

S233 Too many actual parameters to $

The number of actual arguments to the indicated macro exceeded the maximum allowed (currently 31).

S234 Too many formal parameters to $

The number of formal arguments to the indicated macro exceeded the maximum allowed (currently 31).

F235 Too much pushback

The preprocessor ran out of space while processing a macro expansion. The macro may be recursive.

W236 Undefined directive $

The identifier following a # was not a directive name.

S237 EOF in #include directive

End of file was encountered while processing a **#include** directive.

S238 Unmatched #elif

A **#elif** directive was encountered with no preceding **#if** or **#elif** directive.

S239 Unmatched #else

A **#else** directive was encountered with no preceding **#if** or **#elif** directive.

S240 Unmatched #endif

A **#endif** directive was encountered with no preceding **#if**, **#ifdef**, or **#ifndef** directive.

S241 Include files nested too deeply

The nesting depth of #include directives exceeded the maximum (currently 20).

S242 Unterminated macro definition for $

A newline was encountered in the formal parameter list for the indicated macro.

S243 Unterminated string or character constant

A newline with no preceding backslash was found in a quoted string.

I244 Possible nested comment

The characters **/*** were found within a comment.

I245 Redefining predefined macro $

I246 Undefining predefined macro $

W247 Can't redefine predefined macro $

W248 Can't undefine predefined macro $

F249 #error -- $

W250 #ident not followed by quoted string

2251 Extraneous tokens ignored following # directive

F252 Unexpected EOF following # directive

W253 Unexpected # ignored in #if expression

S254 Illegal number in directive

S255 Illegal token in #if expression

S256 Missing > in #include

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

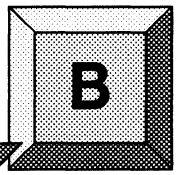
I280 Unrecognized #pragma$

Ignored if not recognized.

W281 <reserved message number>

Messages 280-300 are reserved for **#pragma** handling.

Compiler Internal Structure



This appendix describes the internal structure of the compilers as shown in Figure B-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.

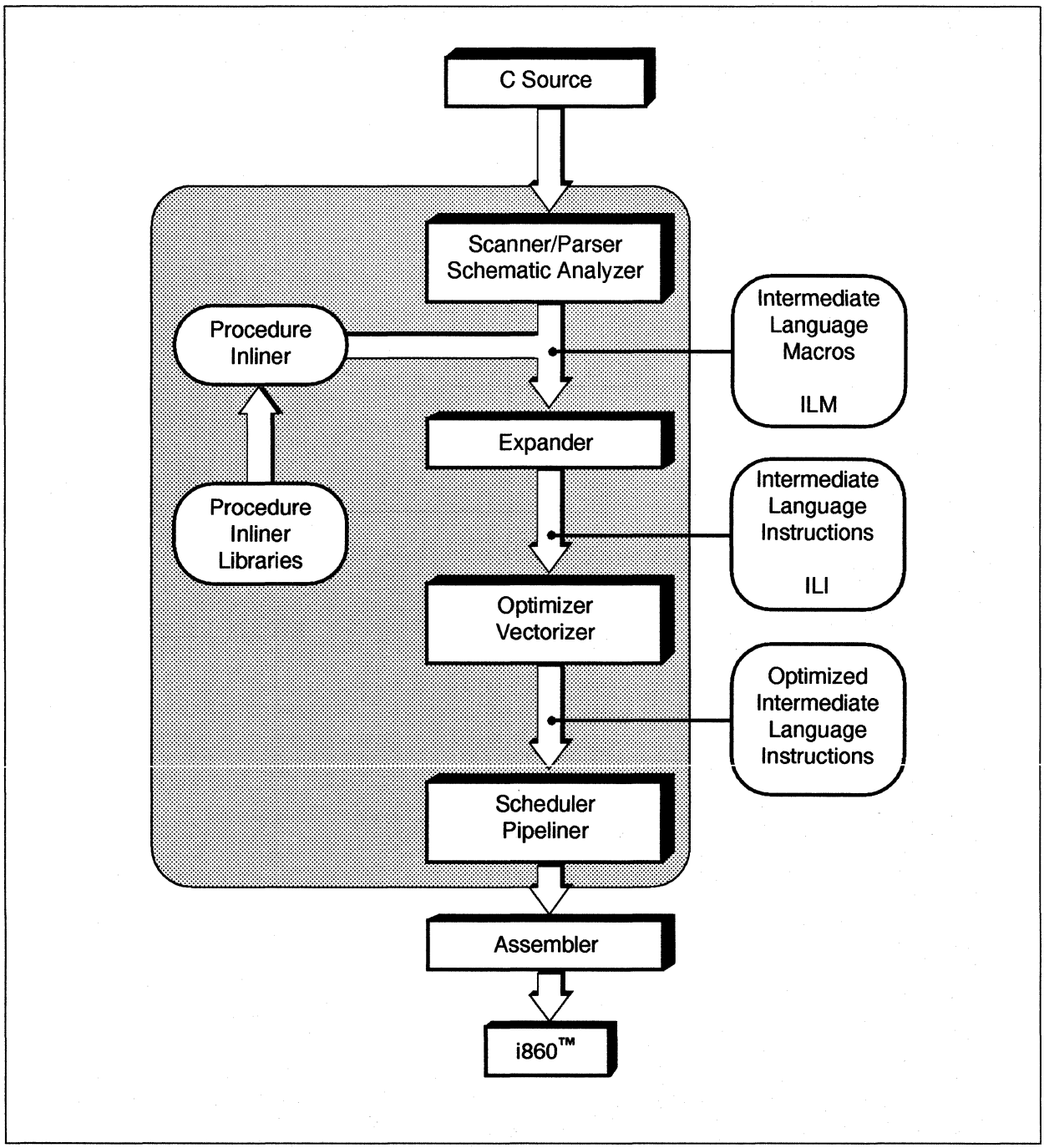


Figure B-1. Compiler Structure

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 B-2.

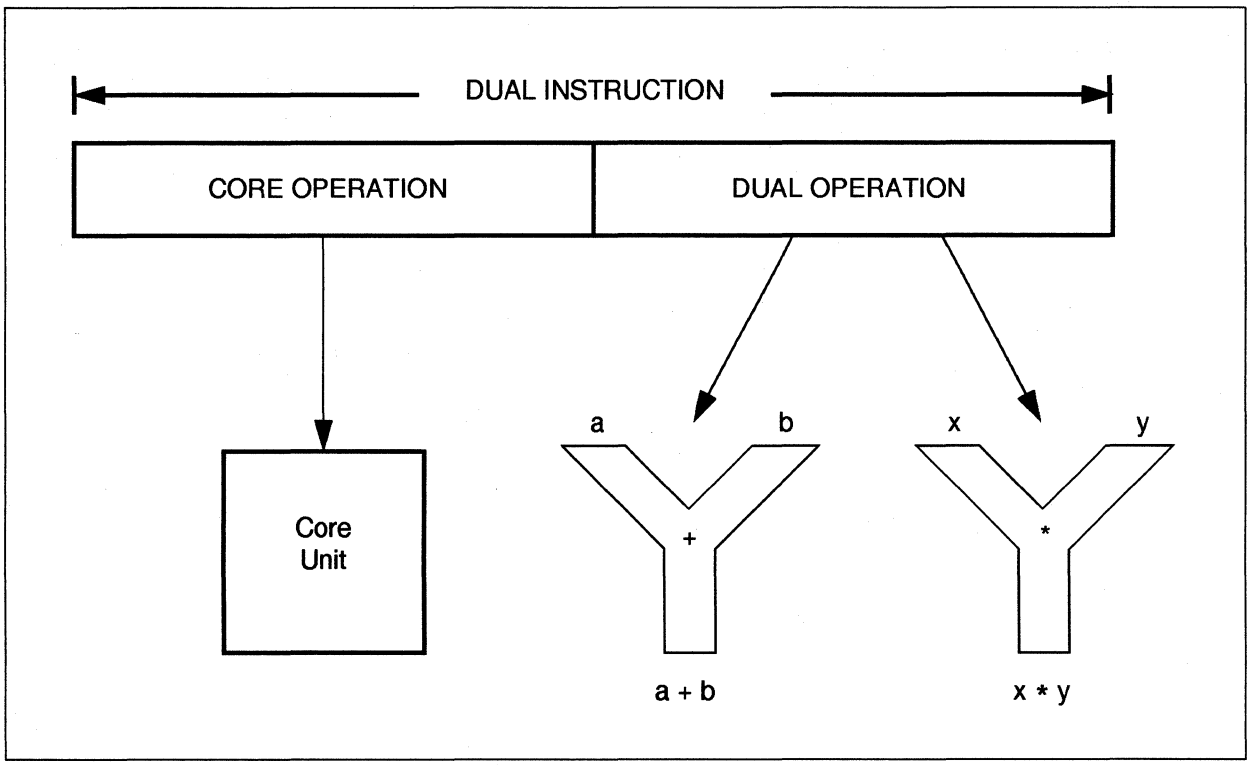


Figure B-2. Parallel Activities of i860™ Microprocessor

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 C

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 OSF/1.
- See the *Paragon™ Commands Reference Manual* and the *Paragon™ C System Calls Reference Manual* for manual pages for parallel commands and system calls unique to Paragon OSF/1.

The manual pages in this appendix are also available on-line, using the **man** command.

Table C-1 lists the commands described in this appendix.

Table C-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.
cpp860	**cpp860** (cross) **cpp** (native)	Preprocesses C programs.
dump860	**dump860** (cross and native)	Dumps object files.
icc	**icc** (cross) **cc** (native)	Compiles C 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 Intel supercomputer and on supported workstations; they are available by their native names on the Intel supercomputer only.

Table C-2 lists the system calls described in this appendix.

Table C-2. System Calls Discussed in This Appendix

Manual Page	System Calls	Description
dv_acos()	**dv_acos(), dv_asin(), dv_atan(), dv_atan2(), dv_cos(), dv_div(), dv_exp(), dv_log(), dv_pow(), dv_recp(), dv_rsqrt(), dv_sin(), dv_sqrt(), dv_tan()**	Double-precision vector intrinsics.
sv_acos()	**sv_acos(), sv_asin(), sv_atan(), sv_atan2(), sv_cos(), sv_div(), sv_exp(), sv_log(), sv_pow(), sv_recp(), sv_rsqrt(), sv_sin(), sv_sqrt(), sv_tan()**	Single-precision vector intrinsics.

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

CPP860 CPP860

cpp860, cpp: C language preprocessor for the Paragon(TM) system.

Cross-Development Syntax

cpp860 [*switches*] [*input_file* [*output_file*]]

Native Syntax

cpp [*switches*] [*input_file* [*output_file*]]

Description

The **cpp860** command invokes the C compiler to preprocess C language source files.

NOTE

ANSI C predefined macros can be defined and undefined on the command line, but not with **#define** and **#undefine** directives in the source file.

Arguments

input_file Input file to be preprocessed (default standard input).

output_file Output file after preprocessing (default standard output).

You may specify the following *switches* in any order:

-A Allows spacing around tokens.

-B Allows C++-style comments (`//` to end of line) in source code.

-C Preserves comments in preprocessed C source files.

-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`.

The normal predefined macros are **__i860, __i860__, __PARAGON__, __OSF1__, __PGC__, _PGC_, _COFF, unix, MACH, CMU, __I860__, _I860_, __I860, _i860_, OSF1_ADFS, OSF1AD.**

CPP860 *(cont.)* CPP860 *(cont.)*

-E Preprocesses the input file, regardless of the file suffix, and sends the output to *stdout*.

-ES Behaves the same as **-E**.

-I*directory* Adds *directory* to the compiler's search path for include files. For include files surrounded by angle brackets (<...>), each **-I** directory is searched followed by the default location. For include files surrounded by double quotes ("..."), the directory containing the file containing the **#include** directive is searched, followed by the **-I** directories, followed by the default location.

-M Outputs a list of include files to *stdout* (used for *makefile* construction). An input file is required with this switch.

-MD Outputs a list of include files to *file*.d (used for *makefile* construction). An input file is required with this switch.

-Mallow_spacing
Allow spacing around tokens such as "." and "@" when used with **-ES**.

-P Preprocesses each input file without producing the line control information used by the next pass of the compiler.

-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 Display the tool banner (tool name, version, etc.).

Files

file.c	C source file.
file.d	List of include files produced by **-MD**.
file.i	C source file after preprocessing.

The following files and directories are used in the cross-development environment (**cpp860**). *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 C compiler is 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**).

CPP860 *(cont.)* CPP860 *(cont.)*

$(PARAGON_XDEV)/paragon/bin.arch/ic	C compiler.
$(PARAGON_XDEV)/paragon/include	Standard include directory.

The following files and directories are used by default in the native environment (**cpp**). 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/ic	C compiler.
/usr/include	Standard include directory.

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.

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**

ICC ICC

icc, cc: Driver for compiling, assembling, and linking C programs for the Paragon(TM) system.

Cross-Development Syntax

icc [*switches*] *sourcefile...*

Native Syntax

cc [*switches*] *sourcefile...*

Description

The **icc** command invokes the C compiler, assembler, and linker with switches derived from **icc**'s command line arguments.

icc bases its processing on the suffixes of the files it is passed:

file.c	is a C program. It is preprocessed, 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.f or *file*.F	is a Fortran program. It is passed to the Fortran 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 C program is compiled and linked with one **icc** command, then the intermediate object and assembly files are deleted.

NOTE

ANSI C predefined macros can be defined and undefined on the command line, but not with **#define** and **#undefine** directives in the source file.

ICC *(cont.)*

ICC *(cont.)*

Switches

-B Allows C++-style comments (// to end of line) in source code.

-c Skips link step; compiles and assembles only. Leaves the output from the assemble step in a file named *file*.o for each file named *file*.c (unless you also use the **-o** switch).

-C Preserves comments in preprocessed C source files. Also enables **-E**.

-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 ".c" file and sends the result to *stdout*. No compilation, assembly, or linking is performed.

-ES Preprocesses every file and sends the result to *stdout*. No compilation, assembly, or linking is performed.

-g Equivalent to **-Mdebug -O0 -Mframe**.

-I- Accepted, but has no effect.

-I*directory* Adds *directory* to the compiler's search path for include files. If you use more than one **-I** switch, the specified directories are searched in the order they were specified (left to right). For include files surrounded by angle brackets (<...>), each *directory* is searched followed by the default location. For include files surrounded by double quotes ("..."), the directory containing the file containing the **#include** directive is searched, followed by the **-I** directories, followed by the default location.

-K*option* Requests special mathematical semantics. The *option* values are:

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 **float** and **double** 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**.

ICC *(cont.)*

ICC *(cont.)*

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. If used together with **-lm**, also links in a version of *libm.a* 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 **float** and **double** 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.

-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 Outputs a list of include files to the standard output (used for *makefile* construction).

-MD Outputs a list of include files to *file*.d (used for *makefile* construction).

ICC *(cont.)*

ICC *(cont.)*

-M*option* Requests specific actions from the compiler. The *option* values are as follows (an unrecognized **-M** *option* is passed directly to the compiler):

allow_spacing Allows spacing around tokens such as "." and "@" when used with **-ES**.

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.

[no]asmkeyword
[Don't] allow the **asm** keyword in C source code (default **-Masmkeyword**). The format is: **asm("***text***")**

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.

ICC *(cont.)*

ICC *(cont.)*

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.

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.

cpp860

Direct the internal preprocessor to not compress white space.

ICC *(cont.)*

ICC *(cont.)*

[no]dalign [Don't] align **double**s in structures on double-precision boundaries (default **-Mdalign**). **-Mnodalign** may lead to data alignment exceptions.

[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**). This is especially useful in disambiguating unknown data dependencies between pointers that cannot be resolved at compile time. For example, if two floating point array pointers are passed to a function and the pointers never overlap and thus never conflict, then this switch may result in better code. The granularity of this switch is rather coarse, and hence the user must use precaution to ensure that other *necessary* data dependencies are not overridden. Do not use this switch if such data dependencies do exist. **-Mnodepchk** may result in incorrect code; the **-Msafeptr** switch provides a less dangerous way to accomplish the same thing.

dollar,*char* Set the character used to replace dollar signs in names to be *char*. Default is an underscore (_).

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

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

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

If both *number*(s) and *function*(s) are specified, then functions 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 functions. The name of the specified *file* must contain a period.

ICC *(cont.)*

ICC *(cont.)*

fcon Treat non-suffixed floating point constants as **float**, rather than **double**. This may improve the performance of single-precision code.

[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]func32 [Don't] align functions on 32-byte boundaries (default **-Mfunc32**). **-Mfunc32** may improve cache performance for programs with many small functions.

info=[*option*[,*option*...]]
Produce useful information on the standard error output. The options 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 functions 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.

ICC *(cont.)*

ICC *(cont.)*

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

[**lib:**]*library*—Inline functions 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, functions are extracted from a temporary library created during an extract prepass.

[**name:**]*function*—Inline the specified function. If **name:** is not used, the function name must not contain a period.

[**size:**]*number*—Inline functions containing less than approximately *number* statements.

levels:*number*—Perform *number* levels of inlining (default 1).

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

keepasm Keep the assembly file for each C source file, but continue to assemble and link the program. This is mainly for use 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 ".c" 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.

ICC *(cont.)*

ICC *(cont.)*

nostartup Don't link the usual start-up routine (*crt0.o*), which contains the entry point for the program.

nostddef Don't predefine any system-specific macros to the preprocessor when compiling a C program. (Does not affect ANSI-standard preprocessor macros.) The system-specific predefined macros are **__i860**, **__i860__**, **__PARAGON__**, **__OSF1__**, **__PGC__**, **_PGC_**, **_COFF**, **unix**, **MACH**, **CMU**, **__I860__**, **_I860_**, **__I860**, **_i860_**, **OSF1_ADFS**, **OSF1AD**,and **__NODE** (**__NODE** is only defined when compiling with **-nx**). See also **-U**.

nostdinc Remove the default include directory (*/usr/include* for **cc**, *$(PARAGON_XDEV)/paragon/include* for **icc**) from the include files search path.

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

[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.

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]reentrant [Don't] generate reentrant code (default **-Mreentrant**). **-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). **-Mnoreentrant** is currently ignored by the compiler.

ICC *(cont.)*

ICC *(cont.)*

retain_static Do not eliminate static data that is not referenced.

safeptr=[*option*[,*option*...]]
Override data dependence between C pointers and arrays. This is a potentially very dangerous option since the potential exists for code to be generated that will result in unexpected or incorrect results as is defined by ANSI C. However, when used properly, this option has the potential to greatly enhance the performance of the resulting code, especially floating point oriented loops. Combinations of the *options* may be used and interact appropriately.

dummy or **arg**—C dummy arguments (pointers and arrays) are treated with the same copyin/copyout semantics as Fortran dummy arguments.

auto—C local or **auto** variables (pointers and arrays) are assumed not to overlap or conflict with each other and are independent.

static—C **static** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.

global—C global or **extern** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.

schar Treat ordinary **char** declarations as **signed char**. This is the default condition.

[no]signextend [Don't] sign extend when a narrowing conversion overflows (default **-Msignextend**). For example, if **-Msignextend** is in effect and an integer containing the value 65535 is converted to a **short**, the value of the **short** will be -1. This option is provided for compatibility with other compilers, even though ANSI C specifies that the result of such conversions are undefined. **-Msignextend** will decrease performance on such conversions.

[no]single [Don't] suppress the ANSI-specified conversion of **float** to **double** when passing arguments to a function with no prototype in scope (default **-Mnosingle**). **-Msingle** may result in faster code when single precision is used a lot, but is non-ANSI compliant.

ICC *(cont.)*

ICC *(cont.)*

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.

[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**).

uchar Treat ordinary **char** declarations as **unsigned char**.

ICC *(cont.)*

ICC *(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 *option*s are specified, then all vector optimizations are enabled. 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.

cachesize:*number*—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.

ICC *(cont.)*

ICC *(cont.)*

smallvect[**:***number*]—Allow 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 will result if this option is used, and a vector takes on a length greater than specified.

streamlim:*n* 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**.

ICC *(cont.)*

ICC *(cont.)*

-nx Creates an executable application for multiple nodes.

- Using **-nx** while compiling defines the preprocessor symbol **__NODE**.
- 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 C Calls Reference Manual*. In addition, it links in *libmach.a* and *options/autoinit.o*. 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*.

-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 C 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 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.

ICC *(cont.)*

ICC *(cont.)*

-p This option is ignored.

-P Preprocesses each file and leaves the output in a file named *file*.i for each file named *file*.c.

-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 step. Leaves the output from the compile step in a file named *file*.s for each file named *file*.c.

-U*name* Removes any initial definition of *name* in the preprocessor. (See also the **nostddef** option of the **-M** switch.) Since all **-D** switches are processed before all **-U** switches, the **-U** switch can be used to override the **-D** switch.

The following macro names are predefined: **__LINE__**, **__FILE__**, **__DATE__**, **__TIME__**, **__STDC__**, **__i860**, **__i860__**, **__PARAGON__**, **__OSF1__**, **__PGC__**, **_PGC_**, **_COFF**, **unix**, **MACH**, **CMU**, **__I860__**, **_I860_**, **__I860**, **_i860_**, **OSF1_ADFS**, **OSF1AD**, and **__NODE** (**__NODE** is only defined when compiling with **-nx** or **-node**). Note that some of these macro names begin and/or end with *two* underscores.

-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 Displays 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.

ICC *(cont.)* ICC *(cont.)*

-X(a | c | s | t | l | o)

Specify the degree of ANSI C conformance.

a	ANSI mode. The compiled language conforms to all ANSI features. _ _STDC_ _ is defined to be zero.
c	Conformance mode. The compiled language conforms to ANSI C, but warnings may be produced about some extensions. _ _STDC_ _ is defined to be one. This is the most strict of the ANSI C conformance options. If you want to write full ANSI-conformant code, you should use the **-Xc** option
s	Pre-ANSI mode. The compiled language includes all features compatible with the C language as defined in *The C Programming Language*, by Kernighan and Ritchie (pre-ANSI C). The compiler warns about all language constructs that differ between ANSI C and pre-ANSI C.
t	Transition mode. This is ANSI C plus pre-ANSI C compatibility extensions without the semantic changes required by ANSI C. Where ANSI C and pre-ANSI C specify different semantics for the same construct, the compiler issues a warning and uses the pre-ANSI C interpretation.
l	Treat [un]signed int and [un]signed long as the same data type. When you use this switch, debug records for [un]signed long are type [un]signed int.
o	Execute the R4.1.1 version of **ic**.

By default _ _STDC_ _ is defined to be one and ANSI conformance is relaxed.

ICC *(cont.)*

ICC *(cont.)*

-Y*pass,directory*

Looks 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.c	C source file.
file.d	List of include files produced by **-MD**.
file.i	C source file after preprocessing.
file.lst	Listing file produced by **-Mlist**.
file.o	Object file.
file.s	Assembler source file.

ICC *(cont.)*

ICC *(cont.)*

The following files and directories are used in the cross-development environment (**icc**). *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/icc	C compiler driver.
$(PARAGON_XDEV)/paragon/bin.arch/ic	C 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/libpm.o	Performance monitoring module.
$(PARAGON_XDEV)/paragon/lib-coff/guard.o	Barrier between user and system code.
$(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 version of *libm.a*).
$(PARAGON_XDEV)/paragon/lib-coff/options/autoinit.o	Routine linked in when **-nx** is used.

ICC *(cont.)* ICC *(cont.)*

The following files and directories are used by default in the native environment (**cc**). 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/cc	C compiler driver.
/usr/ccs/bin/ic	C 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/libpm.o	Performance monitoring module.
/usr/lib/guard.o	Barrier between user and system code.
/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 version of *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, if77, 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

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** vectors.

Synopsis

```
void dv_acos(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );

void dv_asin(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );

void dv_atan(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );

void dv_atan2(
        int n,
        double *x,
        int incx,
        double *y,
        int incy,
        double *z,
        int incz );
```

DV_ACOS() *(cont.)*

DV_ACOS() *(cont.)*

```
void dv_cos(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );

void dv_div(
        int n,
        double *x,
        int incx,
        double *y,
        int incy,
        double *z,
        int incz );

void dv_exp(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );

void dv_log(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );

void dv_log10(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );
```

DV_ACOS() *(cont.)*

DV_ACOS() *(cont.)*

```
void dv_pow(
        int n,
        double *x,
        int incx,
        double *y,
        int incy,
        double *z,
        int incz );

void dv_recp(
        int n,
        double alpha,
        double *x,
        int incx,
        double *z,
        int incz );

void dv_rsqrt(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );

void dv_sin(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );

void dv_sqrt(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );
```

DV_ACOS() *(cont.)* DV_ACOS() *(cont.)*

```
void dv_tan(
        int n,
        double *x,
        int incx,
        double *z,
        int incz );
```

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 functions, 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 ($z[i]$ = **acos**($x[i]$)).
dv_asin()	Vector arcsine ($z[i]$ = **asin**($x[i]$)).
dv_atan()	Vector arctangent ($z[i]$ = **atan**($x[i]$)).
dv_atan2()	Vector arctangent from two arguments ($z[i]$ = **atan2**($x[i]$, $y[i]$)).
dv_cos()	Vector cosine ($z[i]$ = **cos**($x[i]$)).
dv_div()	Non-IEEE vector divide ($z[i] = y[i]/x[i]$).
dv_exp()	Vector exponential ($z[i]$ = **exp**($x[i]$)).
dv_log()	Vector natural log ($z[i]$ = **log**($x[i]$)).
dv_log10()	Vector logarithm $\log_{10}$ ($z[i]$ = **log10**($x[i]$)).

DV_ACOS() *(cont.)* DV_ACOS() *(cont.)*

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 ($z[i] = 1/$**sqrt**$(x[i])$).
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** vector *x* with stride *incx*, storing the results in the **double** vector *z* with stride *incz*:

```
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 = 0;
iz = 0;
if(incx < 0)
    ix = (-n+1)*incx;
if(incz < 0)
    iz = (-n+1)*incz;
for(i=0; i<n; i++) {
    z[iz] = cos(x[ix]);
    ix = ix + incx;
    iz = iz + incz;
}
```

See Also

sv_acos()

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 **float** vectors.

Synopsis

```
void sv_acos(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );

void sv_asin(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );

void sv_atan(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );

void sv_atan2(
        int n,
        float *x,
        int incx,
        float *y,
        int incy,
        float *z,
        int incz );
```

SV_ACOS() *(cont.)*

SV_ACOS() *(cont.)*

```
void sv_cos(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );

void sv_div(
        int n,
        float *x,
        int incx,
        float *y,
        int incy,
        float *z,
        int incz );

void sv_exp(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );

void sv_log(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );

void sv_log10(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );
```

SV_ACOS() *(cont.)*

SV_ACOS() *(cont.)*

```
void sv_pow(
        int n,
        float *x,
        int incx,
        float *y,
        int incy,
        float *z,
        int incz );

void sv_recp(
        int n,
        float alpha,
        float *x,
        int incx,
        float *z,
        int incz );

void sv_rsqrt(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );

void sv_sin(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );

void sv_sqrt(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );
```

SV_ACOS() *(cont.)*

SV_ACOS() *(cont.)*

```
void sv_tan(
        int n,
        float *x,
        int incx,
        float *z,
        int incz );
```

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 functions, 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 ($z[i]$ = **acos**($x[i]$)).
sv_asin()	Vector arcsine ($z[i]$ = **asin**($x[i]$)).
sv_atan()	Vector arctangent ($z[i]$ = **atan**($x[i]$)).
sv_atan2()	Vector arctangent from two arguments ($z[i]$ = **atan2**($x[i]$, $y[i]$)).
sv_cos()	Vector cosine ($z[i]$ = **cos**($x[i]$)).
sv_div()	Non-IEEE vector divide ($z[i] = y[i]/x[i]$).
sv_exp()	Vector exponential ($z[i]$ = **exp**($x[i]$)).
sv_log()	Vector natural log ($z[i]$ = **log**($x[i]$)).
sv_log10()	Vector logarithm $\log_{10}$ ($z[i]$ = **log10**($x[i]$)).

SV_ACOS() *(cont.)* SV_ACOS() *(cont.)*

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 ($z[i] = 1/$**sqrt**$(x[i])$).
sv_sin()	Vector sine ($z[i] =$ **sin**$(x[i])$).
sv_sqrt()	Non-IEEE vector square root ($z[i] =$ **sqrt**$(x[i])$).
sv_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 **sv_cos()** performs a single-precision vector cosine of the first *n* elements of the **float** vector *x* with stride *incx*, storing the results in the **float** vector *z* with stride *incz*:

```
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 = 0;
iz = 0;
if(incx < 0)
    ix = (-n+1)*incx;
if(incz < 0)
    iz = (-n+1)*incz;
for(i=0; i<n; i++) {
    z[iz] = cos(x[ix]);
    ix = ix + incx;
    iz = iz + incz;
}
```

See Also

dv_acos()

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y