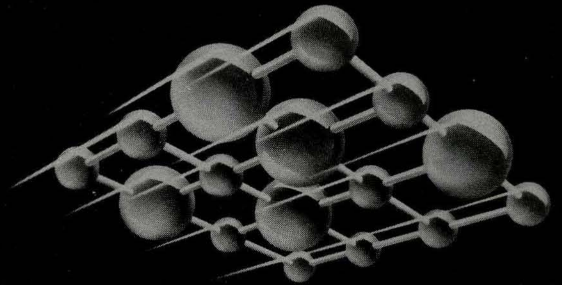
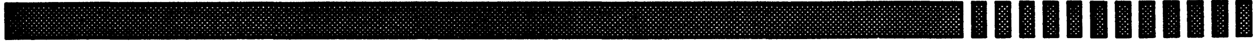
iPSC®/860



C COMPILER USER'S GUIDE

312130-001

April 1991
Order Number: 312130-001



iPSC®/860
C COMPILER USER'S GUIDE

intel® Corporation

Copyright ©1991 by Intel Supercomputer Systems Division, Beaverton, Oregon. All rights reserved. No part of this work may be reproduced or copied in any form or by any means...graphic, electronic, or mechanical including photocopying, taping, or information storage and retrieval systems...without the express written consent of Intel Corporation. The information in this document is subject to change without notice.

Intel Corporation make no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR-7-104.9(a)(9).

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

- 286
- 287
- 4-SITE
- Above
- BITBUS
- COMMputer
- Concurrent File System
- Concurrent Workbench
- CREDIT
- Data Pipeline
- Direct-Connect Module
- FASTPATH
- GENIUS
- i
- I^2ICE
- i386
- i486
- i860
- ICE
- iCEL
- iCS
- iDBP
- iDIS
- iLBX
- im
- Im
- iMDDX
- iMMX
- Insite
- int$_e$l
- int$_e$lBOS
- Intelevision
- int$_e$ligent Identifier
- int$_e$ligent Programming
- Intel
- Intel386
- Intel486
- Intellec
- Intellink
- iOSP
- iPDS
- iPSC
- iRMX
- iSBC
- iSBX
- iSDM
- iSXM
- KEPROM
- Library Manager
- MAP-NET
- MCS
- Megachassis
- MICROMAINFRAME
- MULTI CHANNEL
- MULTIMODULE
- ONCE
- OpenNET
- OTP
- PC BUBBLE
- Plug-A-Bubble
- PROMPT
- Promware
- QUEST
- QueX
- Quick-Pulse Programming
- Ripplemode
- RMX/80
- RUPI
- Seamless
- SLD
- SugarCube
- UPI
- VLSiCEL

Ada is a registered trademark of the U.S. Government, Ada Joint Program Office
APSO is a service mark of Verdix Corporation
CLASSPACK is a trademark of Kuck & Associates, Inc.
Ethernet is a registered trademark of XEROX Corporation
Excelan is a trademark of Excelan Corporation
EXOS is a trademark or equipment designator of Excelan Corporation
FORGE is a trademark of Pacific-Sierra Research Corporation
Green Hills Software, C-386, and FORTRAN-386 are trademarks of Green Hills Software, Inc.
GVAS is a trademark of Verdix Corporation
IBM and IBM/VS are registered trademarks of International Business Machines
Lucid and Lucid Common Lisp are trademarks of Lucid, Inc.
NFS is a trademark of Sun Microsystems
ParaSoft is a trademark of ParaSoft Corporation
Sun Microsystems and the combination of Sun and a numeric suffix are trademarks of Sun Microsystems
The X Window System is a trademark of Massachusetts Institute of Technology
UNIX is a trademark of AT&T
VADS and Verdix are registered trademarks of Verdix Corporation
VAST2 is a registered trademark of Pacific-Sierra Research Corporation
VMS and VAX are trademarks of Digital Equipment Corporation
VP/ix is a trademark of INTERACTIVE Systems Corporation and Phoenix Technologies, Ltd.
XENIX is a trademark of Microsoft Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	04/91

RESTRICTED RIGHTS

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at 52.227-7013. Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

PREFACE

This manual completely documents the iPSC®/860 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 iPSC®/860 software development environment (hardware and software), relates it to the traditional "software development cycle," 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 iPSC®/860 system.
Chapter 2	Describes **icc**, the driver for compiling, assembling, and linking C source code for execution on the iPSC®/860 system.
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	Tells how to use the compiler's function inliner.
Appendix D	Describes the language that the C compiler accepts (draft 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 E	Contains manual pages for the **icc** and **ic** 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 following manuals:

IPSC® System Manuals

iPSC®/2 and iPSC®/860 C Commands and Routines Quick Reference
Summarizes all C routines and commands for the iPSC system.

iPSC®/860 Basic Math Library User's Guide
Describes math library routines (including BLAS and FFT routines) for the iPSC/860 system.

iPSC®/2 and iPSC®/860 Programmer's Reference Manual
Describes iPSC system commands and system calls.

iPSC®/2 and iPSC®/860 User's Guide
Overviews the iPSC system, including hardware and software architectures. Tells how to develop and run programs.

Intel® Manuals

i860™ 64-Bit Microprocessor Assembler and Linker Reference Manual
Tells how to use the i860 microprocessor assembler and linker.

i860™ 64-Bit Microprocessor Object File Utilities Reference Manual
Tells how to use the i860 microprocessor object file utilities.

i860™ 64-Bit Microprocessor Simulator and Debugger Reference Manual
Tells how to use the i860 microprocessor simulator and debugger.

i860™ 64-Bit Microprocessor Programmer's Reference Manual
Describes the i860 microprocessor.

Other Manuals

C: A Reference Manual, Second Edition - Harbison and Steele
Describes the C programming language.

The C Programming Language - Kernighan and Ritchie
Describes the C programming language.

Draft Proposed Standard for Programming Language C (X3J11/88-159)
Describes the proposed ANSI standard C language.

American National Standard for Programming Language C, ANSI x3.159-1989
Describes the accepted ANSI standard C language.

TABLE OF CONTENTS

CHAPTER 1
GETTING STARTED

THE SOFTWARE DEVELOPMENT CYCLE 1-1

THE iPSC®/860 SOFTWARE DEVELOPMENT ENVIRONMENT 1-3

The iPSC®/860 System 1-3

The iPSC®/860 Supporting Software 1-3

THE DRIVER 1-4

The Compiler 1-4

The i860™ Assembler 1-5

The i860™ Linker 1-5

THE i860™ DEBUGGER 1-5

THE iPSC®/860 EXECUTION ENVIRONMENTS 1-6

Running on the i860™ Simulator 1-6

Running on RX Nodes 1-6

EXAMPLE DRIVER COMMAND LINES 1-7

CHAPTER 2
THE ICC DRIVER

INVOKING THE DRIVER 2-1

CONTROLLING THE DRIVER 2-3
- Specific Passes and Options 2-3
- Preprocess Only 2-4
- Preprocess and Compile Only 2-4
- Preprocess, Compile, and Assemble Only 2-5
- Add and Remove Preprocessor Macros 2-5

CONTROLLING THE COMPILATION STEP 2-6
- Specific Actions 2-6
- Special Semantics 2-8
- Location of Include Files 2-8
- Optimization Level 2-9
- Symbolic Debug Information 2-10
- Profiling Code 2-10
- Compatibility 2-10

CONTROLLING THE LINK STEP 2-10
- Specific Actions 2-11
- Linker Libraries 2-11

CONTROLLING THE DRIVER OUTPUT 2-11
- Executable for Simulator or RX Node 2-12
- Name of Executable File 2-12
- Verbose Mode 2-12
- Version and Copyright Statement 2-12

APPENDIX A
COMPILER ERROR MESSAGES

APPENDIX A
COMPILER ERROR MESSAGES

APPENDIX B
COMPILER INTERNAL STRUCTURE

SCANNER AND PARSER B-1

EXPANDER B-3

OPTIMIZER AND VECTORIZER B-3

Procedure Integration B-3

Internal Vectorization B-3

Global Optimizations B-4

Local Optimizations B-4

Flexible Memory Utilization B-5

SCHEDULER AND PIPELINER B-5

APPENDIX C
USING THE INLINER

COMPILER INLINE SWITCH C-2

CREATING A LIBRARY C-2

USING LIBRARIES C-3

RESTRICTIONS ON INLINING C-4

ERROR DETECTION DURING INLINING C-4

EFFICIENCY CONSIDERATIONS C-5

EXAMPLES C-6
Dhry C-6
Fibo C-6
Makefiles C-6

APPENDIX D
EXTENSIONS TO ANSI C

STANDARD LANGUAGE D-1
EXTENSIONS D-2
IMPLEMENTATION-DEFINED BEHAVIOR D-4
PORTING CONSIDERATIONS D-4

APPENDIX E
MANUAL PAGES

ICC E-2
IC E-10

LIST OF ILLUSTRATIONS

Figure 1-1. Traditional Software Development Cycle 1-2

Figure B-1. Compiler Structure B-2

Figure B-2. Parallel Activities of i860™ Microprocessor B-6

LIST OF TABLES

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

GETTING STARTED 1

This chapter introduces the iPSC®/860 software development environment (hardware and software), relates it to the traditional "software development cycle," and shows how to create executable files from C source code.

This chapter contains enough information to get you started using the driver to create iPSC/860-executable files from C source code that conforms to the draft ANSI C standard. For information on iPSC/860 extensions to the standard languages, refer to Appendix D.

THE SOFTWARE DEVELOPMENT CYCLE

Figure 1-1 shows the traditional software development cycle:

1. Write or edit the source code.
2. Compile the source code, producing assembly language code.
3. Assemble the assembly language code, producing object code.
4. Link the object code, producing executable code.
5. Run the executable code.
6. Debug the program (by hand, with the help of a debugger, or both).
7. Repeat steps 1 through 6 as needed.
8. Optimize the program.

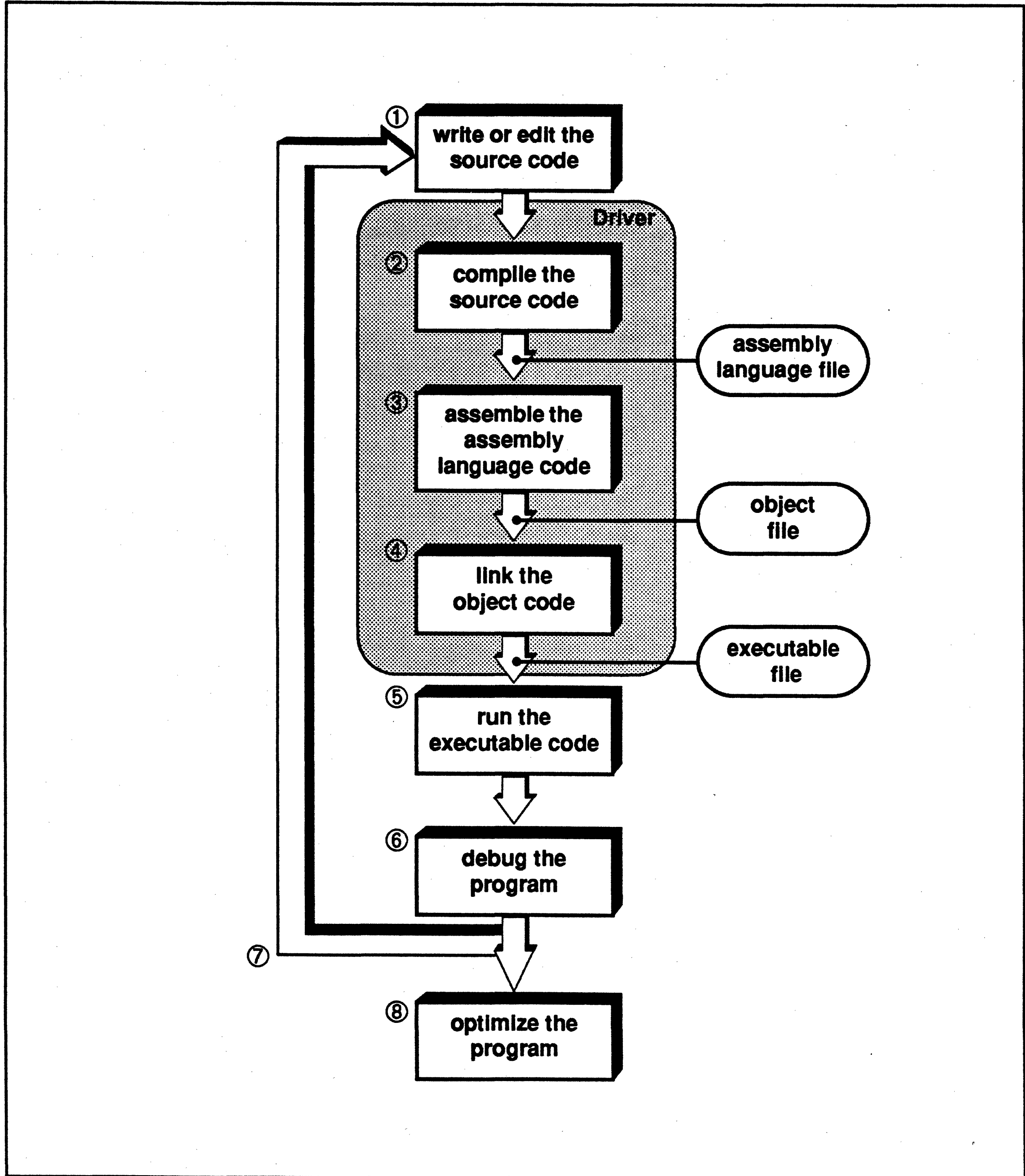


Figure 1-1. Traditional Software Development Cycle

THE iPSC®/860 SOFTWARE DEVELOPMENT ENVIRONMENT

The iPSC/860 software development environment consists of an iPSC/860 system and its supporting software.

The iPSC®/860 System

An iPSC/860 system consists of a *host* and two or more *nodes*:

- There are two kinds of hosts: *local* and *remote*. The local host is the System Resource Manager (SRM). A remote host is any workstation other than the SRM. The host runs the UNIX operating system, augmented with iPSC system extensions and networking software.
- There are two kinds of nodes: *compute* and *I/O*. Each node is a microprocessor/memory pair. Each node's physical memory is distinct from that of the host and other nodes. Each node runs the NX/2 operating system, uses message-passing to communicate with other nodes, and can access both the host file system and the iPSC/2 Concurrent File System™ (CFS).

A typical iPSC system application is developed on either a local or remote host and runs on a group of allocated nodes called a *cube*.

The iPSC®/860 Supporting Software

The iPSC/860 supporting software consists of

- **The software development tools** (compiler, assembler, linker, simulator, and debugger). This manual describes only the iPSC/860 C compiler and driver. For information on the other software development tools, refer to the following manuals:

  i860™ 64-Bit Microprocessor Assembler and Linker Reference Manual
  i860™ 64-Bit Microprocessor Simulator and Debugger Reference Manual

- **The runtime environment** (iPSC/860 system commands and calls). For information on the iPSC/860 runtime environment, refer to the following manuals:

  iPSC®/2 and iPSC®/860 Programmer's Reference Manual
  iPSC®/2 and iPSC®/860 User's Guide

THE DRIVER

The iPSC/860 C driver provides an interface to the compiler, assembler, and linker (steps 2, 3, and 4 of the traditional software development cycle).

The driver makes it easy to produce iPSC/860-executable files from C source code. For example:

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

By default, the driver creates an executable file for execution on the i860™ simulator (**sim860**). The section "Running on RX Nodes" (on page 1-6) tells how to create an executable file for execution on an iPSC/860 node running the NX/2 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 E contains a manual page for **icc**.

NOTE

If desired, you can invoke the iPSC/860 compiler, assembler, and linker directly (as indicated in the next three sections). However, doing so means that 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.

The Compiler

The **ic** command invokes the C compiler directly (step 2 of the traditional software development cycle). For example, the following command line compiles the C source code in the file *myprog.c* and leaves an assembly language version of the program in the file *myprog.s*:

```
ic -astype 1 myprog.c
```

For more information on the **ic** compiler, refer to the **ic** manual page in Appendix E.

The i860™ Assembler

The **as860** command invokes the i860 assembler to assemble the output of the **ic** compiler (step 3 of the traditional software development cycle). For example, the following command line assembles the file *myprog.s* (the default output of the **ic** command), 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 *i860™ 64-Bit Microprocessor Assembler and Linker Reference Manual.*

The i860™ Linker

The **ld860** command invokes the i860 linker to link the output of the **as860** assembler (step 4 of the traditional software development cycle). For example, the following command line links the file *myprog.o* (the default output of the **as860** command), and leaves the resulting executable in the file *a.out*:

```
ld860 myprog.o
```

For more information on using the i860 linker, refer to the *i860™ 64-Bit Microprocessor Assembler and Linker Reference Manual.*

THE i860™ DEBUGGER

The **sim860** command with the **-d** switch invokes the i860 simulator to run, under control of the i860 debugger, the executable file created by the **ld860** linker (step 6 of the traditional software development cycle) or the **icc** driver. For example, the following command line tells the simulator to start an interactive debugging session, accepting debugger commands from the standard input device:

```
sim860 -d a.out
```

Other command line switches let you further configure the debugging session. For complete information on invoking and controlling the i860 debugger, refer to the *i860™ 64-Bit Microprocessor Simulator and Debugger Reference Manual.*

NOTE

Do not use the **-node** switch to create executable programs that you want to debug with the i860 debugger. Programs created with the **-node** switch run only on RX nodes.

THE iPSC®/860 EXECUTION ENVIRONMENTS

The iPSC/860 software tools let you create executable files for execution in either of two possible environments (step 5 of the traditional software development cycle):

- The **sim860** simulator
- RX (i860-based) nodes

Running on the i860™ Simulator

The **sim860** command invokes the i860 simulator to run the executable file created by the **ld860** linker. For example, the following command line causes the program in the file *a.out* to execute under control of the i860 simulator:

```
sim860 a.out
```

For complete information on using the i860 simulator, refer to the *i860™ 64-Bit Microprocessor Simulator and Debugger Reference Manual.*

Running on RX Nodes

By default, the **icc** driver creates a file for execution by the i860 simulator (**sim860**). To create an executable file for execution on an iPSC/860 node running the NX/2 operating system, use the driver's **-node** switch. For example:

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

Creating a node-executable file using the compiler, assembler, and linker directly is a bit more complex. For example, the above **icc** command line generates **ic**, **as860**, and **ld860** command lines similar to the following:

```
ic myprog.c -astype 1 -x 127 1 \
   -stdinc /vol/pgi/i860/include:/vol/pgi/i860/include-ipsc \
   -def i860 -def __i860 -def __NX -def __NODE \
   -predicate #machine(i860) #lint(off) #system(nx) #cpu(i860) \
   -ieee 1 -opt 1 -x 121 1 -asm /usr/tmp/AAAa05259

as860 -o myprog.o /usr/tmp/AAAa05259

ld860 -e ld$start -T 0x0 -d 0x10000000 \
   /vol/pgi/i860/lib-coff/crt0.o myprog.o \
   /vol/pgi/i860/lib-coff/libnode.a \
   /vol/pgi/i860/lib-coff/iclib.a
```

No matter how you create a node-executable file, you next use the **getcube** command to get a cube on which to run the program and the **load** command to load and run the executable file. The program begins executing on each node as soon as it is loaded. For example, the following command lines allocate the largest possible cube consisting entirely of RX nodes, and load the executable file *a.out* on each node of the allocated cube:

```
getcube -trx
load    a.out
```

For complete information on running programs on the iPSC/860, refer to the *iPSC®/2 and iPSC®/860 Programmer's Reference Manual* and the *iPSC®/2 and iPSC®/860 User's Guide.*

EXAMPLE DRIVER COMMAND LINES

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

- Compile and link for an RX node (**-node**), leaving executable in a file called *x* (**-o x**):

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

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

```
icc  -node -o x x.c -lm
```

- Same as above, but include debugging information (**-g**):

```
icc  -g -node -o x x.c
```

- Compile and link for debugging on **sim860** simulator (no **-node** switch):

```
icc  -g -o x x.c
```

- 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)*

THE ICC DRIVER 2

This chapter describes **icc**, the driver for compiling (using **ic**), assembling (using **as860**), and linking (using **ld860**) C source code for execution on the iPSC®/860 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:

- Files whose names end with ".c" are considered to be C programs. They are preprocessed, compiled, and assembled. The resulting object file is placed in the current directory. If linking is not suppressed (the default), the object file is deleted following the link step.
- Files whose names end with ".s" are considered to be i860 assembly language files. They are assembled and the resulting object file is placed in the current directory. If linking is not suppressed, the object file is deleted following the link step.
- Files whose names end with ".o" are considered to be object files. They are passed directly to the linker (if linking is not suppressed).

- Files whose names end with ".a" are considered to be **ar** libraries. No action is performed on these files if linking is suppressed.

All other files are assumed to be object files; they are passed to the linker (if linking is not suppressed) with a warning message.

Table 2-1. Summary of icc Driver Switches

Switch	Description
-c	Skip link step; compile and assemble only
-D*name*[=*def*]	Define preprocessor *name* to be *def*
-E	Preprocess only (to *stdout*)
-f	Ignored; for compatibility with other C compilers
-g	Generate symbolic debug information
-I*directory*	Add *directory* to include file search path
-K*flag*	Request special compiler semantics (`ieee`, `noieee`)
-L*directory*	Change default library search directory
-l*library*	Load **lib***library***.a** from standard library directory
-M*flag*	Request special compiler actions (`debug`, `coff`, `frame`, `noframe`, `inline`, `keepasm`, `list`, `nodepchk`, `nostartup`, `nostddef`, `nostdinc`, `nostdlib`, `quad`, `reentrant`, `vect`)
-node	Create executable program for RX node
-O[*level*]	Set optimization *level* (`0`, `1`, `2`, `3`, `4`)
-o*file*	Use *file* as name of executable program
-P	Preprocess only (to *file*.i for each *file*.c)
-p	Generate code for function-level profiling
-S	Skip assemble and link step; compile only
-s	Strip symbol table information
-U*name*	Remove initial definition of *name* in preprocessor
-u*name*	Generate undefined reference
-V	Display each tool's version and copyright info
-v	Turn on driver and tool verbose modes
-W*pass,option*[,*option*...]	Pass *options* to *pass* (`0`, `a`, `1`)
-Y*pass,directory*	Look in *directory* for *pass* (`0`, `a`, `1`, `S`, `L`, `U`, `I`)

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
-c	Skip link step; compile and assemble only
-D	Define (create) preprocessor macro
-U	Undefine (remove) preprocessor macro

Specific Passes and Options

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

-W*pass*, *option*[, *option*...]

where:

pass Is one of the following:

0	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	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*. The standard include directory is set to a default value by the driver and can be overridden by this option.
L	Set the compiler's primary library search path to *directory*.
U	Set the compiler's secondary library search path to *directory*.

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	After preprocessing each *file*.c, send the result to standard output (*stdout*).
-P	After preprocessing each *file*.c, send the result to a file named *file*.i.

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:

-S

After compiling each *file*.c, the output 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.

Add and Remove Preprocessor Macros

The following command line switches let you add (define) and remove (undefine) C 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`.

-U*name* Undefine any initial definition of *name* in the preprocessor. The only names predefined by the preprocessor itself are the standard ANSI C predefined macros. However, the driver can predefine other names (using the **-Mnostddef** option).

Because all **-D** options are processed before all **-U** options, the **-U** option overrides the **-D** option.

CONTROLLING THE COMPILATION STEP

The following switches let you control the compilation step:

-M	Request special compiler actions
-K	Request special compiler semantics
-I	Add a directory to include file search path
-O	Set optimization level
-g	Generate symbolic debug information
-p	Generate code for function-level profiling
-f	Ignored; for compatibility with other C compilers

Specific Actions

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

```
-Mflag
```

where *flag* is one of the following:

`debug` — Put information needed for level 0 symbolic debugging into the output file.

`coff` — Generate Common Object File Format (COFF) format object and executable files (default). This option causes the compiler to emit Intel-style i860 assembly language and causes the COFF assembler and linker to be invoked.

`frame` — Include the frame pointer (default).

`noframe` — Omit the frame pointer unless **varargs** is used or **alloca**() is called. Using this switch can improve execution time and decrease code, but makes it impossible to get a call stack traceback when using a debugger.

`inline=option[,option,...]`
Pass *options* to the function inliner. Refer to Appendix C for 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 switch is used mainly for compiler performance analysis and debugging.

`list[=name]`
: Create a source listing in the listing file with filename *name*. If *name* is not specified, the listing file will have the same name as the source file except that ".c" suffix will be replaced by a ".lst" suffix. If *name* is specified, the listing file will have that name. No extension is appended.

`nodepchk`
: Ignore unknown potential data dependencies. 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. This switch must not be used if such data dependencies do exist.

`nostartup`
: Do not link in the usual start-up routine. This routine contains the entry point for the program. The start-up file is *$(IPSC_XDEV)/i860/lib-coff/crt0.o*.

`nostddef`
: Do not predefine any macros to the preprocessor when compiling a C program. The normal predefined macros are: **-Di860** and **-Dunix**.

`nostdinc`
: Do not search in the standard location (*$(IPSC_XDEV)/i860/include-ipsc*) for include files when those files are not located elsewhere.

`nostdlib`
: Do not link in the standard libraries when linking a program.

`quad`
: Force top-level objects (e.g., local arrays, structures, etc.) of size greater than or equal to 16 bytes to be quad-aligned. Note that this does not affect items within a top-level object; such items are quad aligned only if appropriate padding is inserted. Common blocks are always quad-aligned.

`reentrant`
: Disable compiler optimizations that may prevent code from being reentrant.

`vect=[option[,option,...]]`

Pass the specified options to the internal vectorizer, where *option* may be:

`noassoc`	Prevents the vectorizer from unrolling scalar reductions. Unrolling scalar reductions improves code quality but causes slightly different answers due to associativity in addition (common example is dot product).
`recog`	Causes the vectorizer to recognize idioms and to perform strip-mining, streaming, and some invariant vector motion.
`unroll`	Causes the vectorizer to unroll small loops to improve pipelining opportunities.

Specifying `vect` with no *option*, turns on all vectorizer options.

An unrecognized *flag* is passed directly to the compiler, which often removes the need for the **-W0** switch.

Special Semantics

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

`-K`*`flag`*

where *flag* is one of the following:

`ieee`	Perform **float** and **double** divides in conformance with the IEEE 754 standard. This is default.
`noieee`	Perform **float** and **double** divides using an inline divide algorithm for performance. This algorithm produces results that differ from the IEEE result by no more than three units in the last place.

Location of Include Files

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

`-I`*`directory`*

where *directory* is the pathname of the directory to be added.

For include files surrounded by angle brackets (<..>), each **-I** directory is searched, followed by the standard area. 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 area.

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.

The default optimization level is determined as follows:

0	If **-O** is not specified *and* **-g** is specified
1	If **-O** is not specified *and* **-g** is *not* specified
2	If a *level* is not supplied with **-O**

Symbolic Debug Information

The following command line switch tells the compiler to generate symbolic debug information:

```
-g
```

This switch also sets the optimization level to 0 (unless the **-O** switch appears on the command line after the **-g** switch) and sets the **-Mframe** switch.

Symbolic debugging may give confusing results with optimization levels other than 0. The code produced for level 0 is significantly slower than code generated for other optimization levels.

Profiling Code

The following command line switch tells the compiler to generate code for function-level profiling:

```
-p
```

This switch also sets the **-Mdebug** switch.

Compatibility

The following option is provided for compatibility with other C compilers; it is ignored:

```
-f
```

CONTROLLING THE LINK STEP

The following switches let you control the link step:

-s	Strip symbol table information
-u	Generate undefined reference
-L	Change default library search directory
-l	Load a specific library from standard library directory

Specific Actions

The **icc** driver passes the following switches directly to the linker:

-s	Strip all symbols from the output object file
-u*name*	Generate an undefined reference for the symbol named *name*

Linker Libraries

The following switch tells the linker where to find linker libraries:

`-Ldirectory`

where *directory* is the pathname of the directory that the linker searches for libraries. The **icc** driver searches *directory* first (i.e., 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 standard library directory. The library name is constructed and the full library path is passed to the linker.

CONTROLLING THE DRIVER OUTPUT

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

-node	Create executable program for RX node
-o	Specify name of executable program
-v	Turn on driver and tool verbose modes
-V	Print each tool's version and copyright information

NOTE

The **-v** and **-V** switches do not perform the same function for **icc** that they do for **cc**.

Executable for Simulator or RX Node

By default, the **icc** driver creates a program for execution on the **sim860** simulator. However, the following command line switch lets you create a program for execution on an RX node.

-node

Name of Executable File

By default, the executable file is named **a.out**. However, the following command line switch lets you name the file anything you like:

-o*file*

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 command line that invokes each tool, and to turn on verbose mode (if available) for each tool:

-v *(lowercase letter "v")*

Version and Copyright Statement

The following command line switch causes each tool to display its version and copyright statement:

-V *(uppercase letter "V")*

COMPILER ERROR MESSAGES A

This appendix lists the error messages generated by the iPSC®/860 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 Customer Support:

1-800-421-2823 (Customer Support Hotline)
(44) 793 641 469 (in England)
Your Local Intel Sales Office (in Europe)
support@isc.intel.com (Internet address)

`F001 Source input file name not specified`

On the command line, you must specify the name of a source file. You may specify the name either before or after the switches.

`F002 Unable to open source input file: $`

The specified source file (or include file) is not in current working directory or is read-protected. Check to make sure that you spelled the file's name correctly.

`F003 Unable to open listing file`

You may not have write permission in the current working directory.

`F004 Unable to open object file`

You may not have write permission in the current working directory.

`F005 Unable to open temporary file`

The compiler creates temporary files in */usr/tmp* or */tmp*. This error occurs when neither of these directories is available.

`F006 Missing -exlib option`

Extractor program requires **-exlib** option to specify output library.

`F007 Source file too large to compile at this optimization level`

The symbol table overflowed, or the compiler working storage space was exhausted. If this error occurred at optimization level 2, reducing the level to 1 may work around the problem; otherwise consider splitting the source file into two files. There is no hard limit on how large a file the compiler can handle; however, if your file is less than 2,000 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 Function $ extracted. Size = $`

The extractor issues this informative message when it writes entry to inliner library.

```
I010 Function $ inlined
```

```
S011 Unrecognized command line switch: $
```

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

```
S012 Value required for command line switch: $
```

Some switches require a value to follow immediately. For example, **-opt 2**.

```
S013 Unrecognized value specified for command line switch: $
```

```
S014 Ambiguous command line switch: $
```

The abbreviation you used for a command line switch was too short.

```
W015 Can't inline $ - wrong number of arguments
```

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

An identifier cannot be more than 31 characters; characters after the 31st are ignored.

```
I017 Argument of inlined function not used
```

```
S018 Inline library not specified on command line (-inlib switch)
```

```
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 3,000 characters long. It may be possible to work around the problem by removing unneeded blank characters from some 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 not terminated or had no characters.

`S025 Illegal character: $`

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

`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 improperly formed; found during syntax error recovery.

`W031 Multi-character character constant`

You cannot 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 $

Source code processing 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.**"

W039 Use of undeclared variable $

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

S040 Illegal use of symbol, $

S041 $ is not an enumeration tag

An identifier was used as an enumeration tag before it was declared.

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`

`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 Illegal lvalue. Ampersand (&) ignored before array name.`

If *arr* is an array, then &*arr* is not a legal expression. Probably &*arr*[0] was intended.

`S058 Illegal lvalue`

The expression on the left side of an assignment statement or the operand of the 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 that 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 = 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 one wide character.

```
F079 Unable to access file $/TOC
```

```
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
```

```
S084 Unable to open file $ for inlining
```

```
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.

```
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
```

The data types of the left and right 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 performed the necessary type conversion as if you 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.

`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 one bit too large for the signed **short** (**char**) data type. Either the type **unsigned short** (**char**) should be used in place of signed **short** (**char**), or the equivalent negative number should be used in place of the positive constant.

`I117 Value missing from return statement`

This return statement does not return a function value.

`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** (**char**) value can cause this message to be issued. Note that such code is non-portable.

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 defined using the prototype form, but a declaration 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 followed by ... or other parameters

A function prototype of the form **(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`

`V144 Syntax requires semicolon, semicolon inserted`

`V145 Syntax requires no comma, comma deleted`

`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).

`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 Unreasonable include nesting`

The nesting depth of **#include** directives exceeded the maximum (currently 10).

`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

W251 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

COMPILER INTERNAL STRUCTURE B

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 consists of Intermediate Language Instructions (ILIs) representing a sequence of language statements in which 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 options selected by the user, a hierarchy of optimizations may be applied: global optimizations, local optimizations, vectorization, and software pipelining.

SCANNER AND PARSER

Each compiler has a Scanner and Parser that performs syntax and semantic analysis of its respective source language input. A set of ILMs is created and a symbol table and various data structures referring back to the original source code are maintained for diagnostics and symbolic debugging. Error detection and recovery is performed using an advanced multiple parse stack technology.

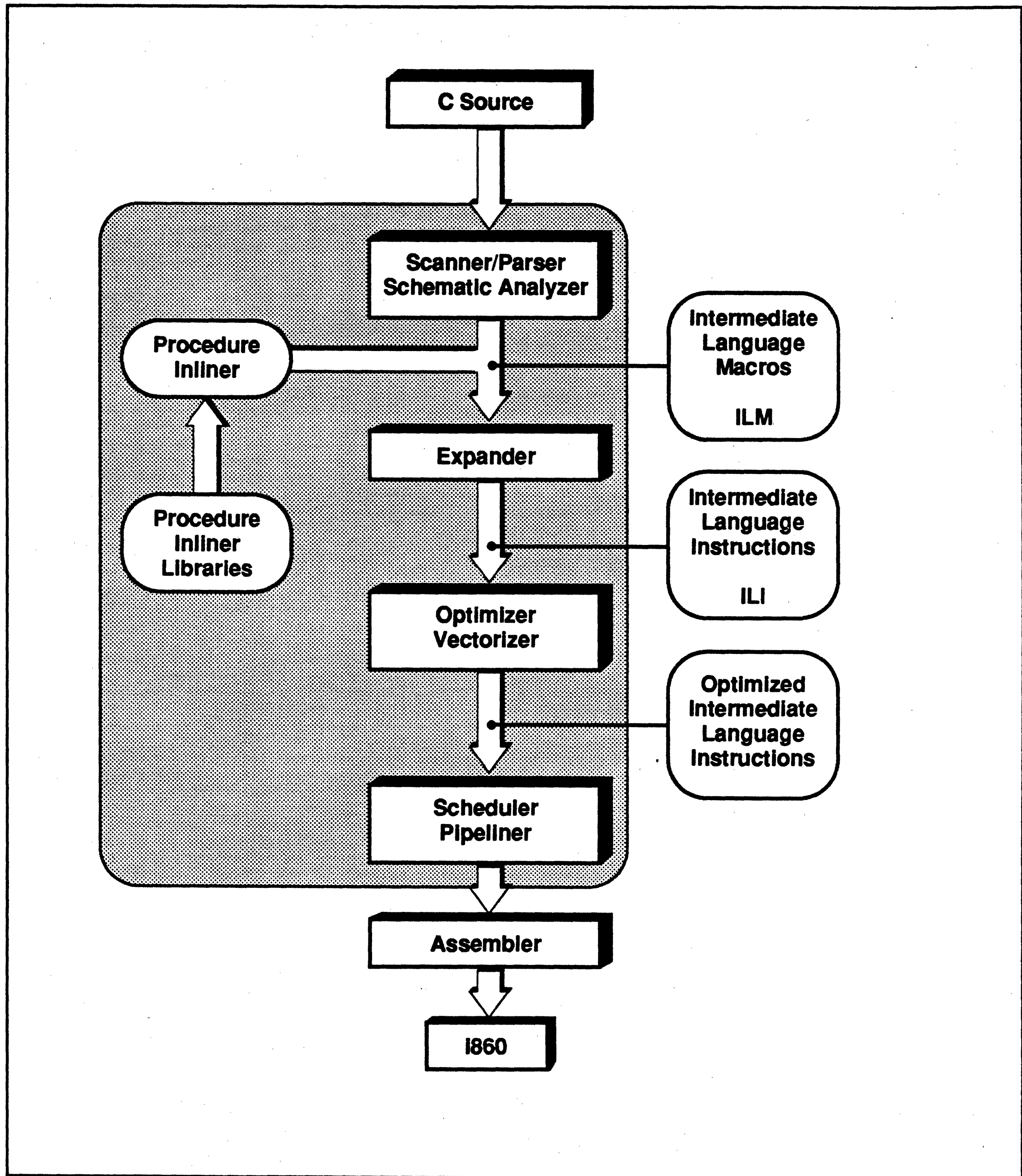


Figure B-1. Compiler Structure

EXPANDER

The Expander lowers the ILM set along with the semantic analysis information and generates a set of ILIs and associated data structures including extended basic block tables and information about referenced variables. Some optimizations such as constant folding, elimination of identity expressions, and branch folding occur at this point. 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 and vectorizer provides both a faster compile time and more efficient code generation than traditional source-to-source preprocessors. Advanced optimizations are used 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 which 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
- WHERE statement transformations using scatter/gather when appropriate
- Splitting out intrinsics
- 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.

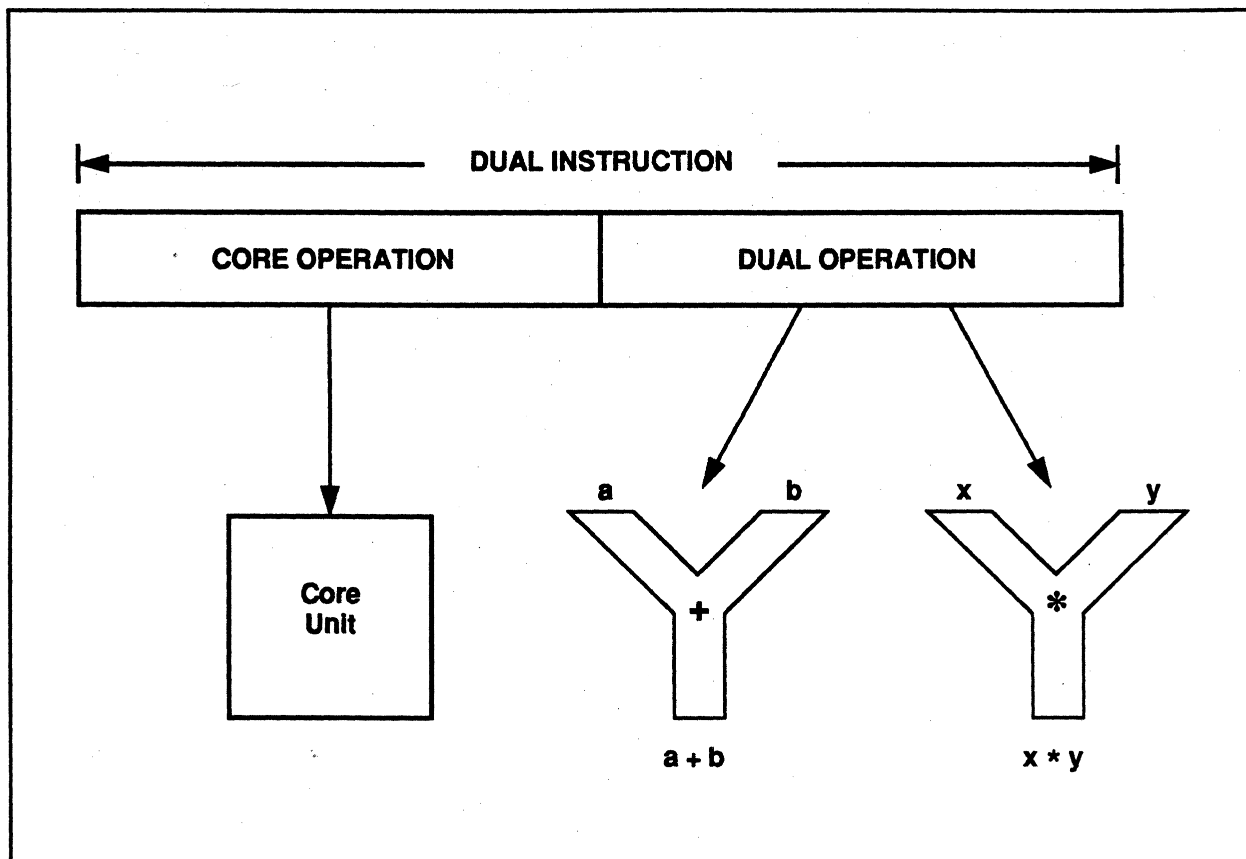


Figure B-2. Parallel Activities of i860™ Microprocessor

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 (regarding loops which are pipelinable, data dependence information, recurrences, array references).

The output of the Scheduler and Pipeliner is a list of assembly language instructions which are passed to an assembler to create the final object file.

USING THE INLINER C

This appendix describes the compiler's function inlining capability.

Function inlining is a compiler optimization under which the body of a function (or subroutine) 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. In this case there is a hidden pass, preceding the compilation pass, to extract functions that are candidates for inlining. The actual inlining of functions takes place as the source files are compiled.

- Use of **inliner libraries**. These libraries are created by previous invocations of **icc** using the **-Mextract** option, or can be part of a compiler release. There is no hidden extract pass, but the user must take care that the library is up to date, and that compilations that depend on the library are updated whenever necessary.

COMPILER INLINE SWITCH

Function inlining is requested using the **-Minline** switch:

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

where *option* is one of the following:

A size limit — This is an integer that represents an upper bound on function size. A function will not be inlined if it exceeds the size limit. This limit roughly corresponds to the number of executable statements in the function.

A function name — Any number of names can be specified. Any function that is named is inlined.

An inliner library name — Any number of libraries can be specified. Library names are distinguished from function names by the presence of a period somewhere in the name. A function is inlined if it is found in any of the libraries.

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 no inliner libraries are specified, a special pass is performed for all source files named on the compiler 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 one or more inliner libraries are specified, no initial extract pass is performed. 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 A LIBRARY

An inliner library is created or updated using the **-Mextract** switch:

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

where *option* is either:

- A size limit
- A function name

These have the same meaning as for the **-Minline** switch described above. If no *option* is specified with **-Mextract**, then an attempt is made to extract all subprograms of a reasonable size.

When the extract option is used, 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 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:

- Libraries can be copied or renamed
- Elements of libraries can be deleted or copied from one library to another
- The **ls** command can be used to learn the date of the most recent change of a library entry

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 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 (**-Mx,0,8**).

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.

EFFICIENCY CONSIDERATIONS

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

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

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

  However, this fragment will not vectorize well:

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

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

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

EXAMPLES

This section contains examples of using the inliner.

Dhry

Assume the program **dhry** consists of a single source file *dhry.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
```

EXTENSIONS TO ANSI C D

This appendix describes the language that ic accepts (draft 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 *Draft Proposed Standard for Programming Language C (X3J11/88-159)*, American National Standards Institute, December 7, 1988.

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 appendix 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 draft ANSI standard, supported by the **ic** 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 **#ident** directive is supported. The syntax is:

   ```
   #ident "string"
   ```

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

7. 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 a set of predefined predicates to the compiler. Only predefined predicates exist; the user may not create new predicates.

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

9. 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.

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

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

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

13. The new ANSI types **unsigned short int** and **unsigned char** are supported. The keyword **signed** is added as per the proposed 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 new ANSI type **long double** is supported; it is currently implemented the same as **double**.

14. The keywords **const** and **volatile** are supported as per the proposed 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.

15. 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.

16. 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.

17. Initialization of automatic aggregates is allowed as per the proposed 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.

18. Both signed and unsigned bit fields are supported. (ANSI)

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

IMPLEMENTATION-DEFINED BEHAVIOR

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

- If the pathname is enclosed in angle brackets, the compiler first searches the directories specified with the **-idir** 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 **ic** 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 (e.g., **static int**, not **int static**). The proposed 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 proposed 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 proposed 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.

MANUAL PAGES E

This appendix contains manual pages for the **icc** and **ic** commands.

ICC ICC

Driver for compiling, assembling, and linking C programs for the iPSC®/860 system.

Synopsis

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

Description

The **icc** command invokes the iPSC®/860 C compiler, assembler, and linker with switches derived from the driver's command line arguments.

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

- Files whose names end with ".c" are considered to be C programs. They are preprocessed, compiled, and assembled. The resulting object file is placed in the current directory.
- Files whose names end with ".s" are considered to be i860 assembly language files. They are assembled and the resulting object file is placed in the current directory.
- Files whose names end with ".o" are considered to be object files. They are passed directly to the linker if linking is requested.
- Files whose names end with ".a" are considered to be **ar** libraries. No action is performed on ".a" files unless linking is requested.
- All other files are taken as object files and passed to the linker (if linking is requested) with a warning message.

If a single C program is compiled and linked with one **icc** command, then the intermediate object and assembly files are deleted.

The compiler is targeted to the Intel i860 System Binary Standard (SBS) for the Intel APX system. The compiler creates object files and executables that will function for any system conforming to the SBS. In addition, the compiler can create object files that will work for any system conforming to the System V, Release 4 Application Binary Interface (ABI) for the i860 microprocessor.