i960® Processor Compiler User's Manual

Order Number: 651230-004

Revision	Revision History	Date
001	Initial Release	02/96
002	Revised for release 5.1	01/97
003	Revised for release 6.0	12/97
004	Revised for release 6.5	12/98

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:

Literature Distribution Center
Intel Corporation
PO Box 5937
Denver, CO 80217-9808

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office.

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice. Contact your local sales office to obtain the latest specifications before placing your order.

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 Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

Contents

Chapter 1 The CTOOLS Compilation System

Chapter 2 gcc960 Compiler Driver

Chapter 3 ic960 Compiler Driver

Chapter 4 Program-Wide Analysis and Optimization

Chapter 5 Profile Data Merging and Data Format (gmpf960)

Chapter 6 gcdm Decision Maker Option

Chapter 7 C Language Implementation

Chapter 8 C++ Language Implementation

Chapter 9 GCC960/ic960 Compatibility

Chapter 10 Position Independence and Reentrancy

Chapter 11 Initializing the Execution Environment

Chapter 12 Optimization

Chapter 13 Caveats

Figures

Tables

Examples

The CTOOLS Compilation System 1

This manual provides operating instructions and other information on the CTOOLS compilation system. This system consists of a compiler and two drivers that provide the user interface to the compiler, gcc960 and ic960. These two interface drivers allow backward compatibility with software developed using GNU/960 and CTOOLS960, respectively.

New Features

- Release 6.5 features support for 64-bit integers using *long long* type.

Features of the Compilation System

The compiler lets you use the following features to develop applications:

- Using either the gcc960 or ic960 compiler driver to invoke and control translation and linking. See Chapter 2 "gcc960 Compiler Driver" or Chapter 2 "ic960 Compiler Driver".
- Creating a run-time performance profile of your application. Optimizations based on this profile include inter-module optimizations and preferential use of fast memory regions for variables that are frequently accessed. For an overview of the program-wide optimization process, including profile-driven optimization, see Chapter 4 "Program-Wide Analysis and Optimization". For descriptions of other optimizations, refer to Chapter 12 "Optimization".

- Calling functions written in i960 processor assembly language, or including in-line assembly language in your C/C++ program. Chapter 7 “C Language Implementation”
- Stopping the compilation process to examine intermediate results after syntax checking, preprocessing, compilation, assembly, or incremental linking. (See Chapters 2, “gcc960 Compiler Driver” and , “ic960 Compiler Driver”.)
- Using a single command to compile, assemble, and link modules into a complete ROM-able or executable program. (See Chapters 2, “gcc960 Compiler Driver” and , “ic960 Compiler Driver”.)
- Using the CAVE pragma to compress functions, thus reducing code size. During program execution, these functions are decompressed when called. For more information on CAVE and the other pragmas, see Chapter 7 “C Language Implementation”.
- Creating blended code with the new `-mcore0-3` and `-Gcore0-3` options. With these options, you can generate code that is compatible with multiple i960 processor types. For more information, see Chapters 2, “gcc960 Compiler Driver” and , “ic960 Compiler Driver”.

Compatibility and Conformance to Standards

The compiler runs on a UNIX* or a Windows* 95/NT* host system and generates object code for any i960 commercial processor. The translation and code generation phases use the instruction set for the i960 processor that you specify.

The compiler's implementation of C conforms to the ANSI standard for the C language (X3.159-1989). One exception is static pointer initialization in applications using position-independent code or data (described in Chapter 9 “GCC960/ic960 Compatibility”). Additionally, the compiler allows use of in-line assembly language in the C source text.

The ANSI standard specifies that a conforming implementation of a C compiler must meet minimum requirements for certain translation limits. In all cases, the compiler exceeds ANSI limits. Table 1-1 lists the tested levels for each translation limit and compares them to ANSI minimum requirements. Available memory determines actual limits in a host system.

Table 1-1 Compiler Limits

Limit	ANSI Minimum	Tested Minimum
Control structure nesting levels	15	128
Conditional compilation nesting levels	6	32
Declarator modifiers	12	32
Declaration parenthesis nesting levels	31	64
Parenthesis nesting levels	32	128
Significant characters for internal identifier	31	128
Name length for external identifier	6	33
Identifiers in a single block	127	1024
Macros simultaneously defined	1024	4096
Parameters per function call	31	128
Characters in a logical line	509	4096
Characters in a string	509	4096
Bytes in an object	32767	65535
Include file nesting levels	8	32
Case labels in a switch	257	1024
Members in one structure or union	127	512
Enumeration constants in one enumeration	127	512
Structure nesting levels	15	64
External identifiers per file	511	2048
Parameters per macro	31	128

About this Manual

This manual contains the following chapters:

Table 1-2 Chapter Descriptions

Chapter Number	Title	Description
1.	The CTOOLS Compilation System	Introduces the compiler and provides information on using this manual.
2.	gcc960 Compiler Driver	Teaches you how to use the gcc960 command-line interface and provides a complete list of command line options.
3.	ic960 Compiler Driver	Teaches you how to use the ic960 command-line interface and provides a complete list of command line options.
4.	Program-wide Analysis and Optimization	Tells you how to use some of CTOOLS most powerful optimization features: • program-wide optimizations • run-time profiling
5.	Profile Data Merging and Data Format (gmpf960)	Explains how to use gmpf960 to merge the execution profile data you collected in Chapter 4 "Program-Wide Analysis and Optimization". You also learn how to use gmpf960 to create a report that shows how many times each basic block was "hit" or run during program execution.
6.	gcdm Decision Maker Option	Describes the `gcdm` option, which invokes the gcdm960 global optimization decision maker during the link process. The decision maker then invokes the compiler and linker as necessary to perform program-wide optimizations.
7.	Language Implementation	Describes data representation, register use, object file format use, and pragmas for modifying code generation.

Table 1-2 Chapter Descriptions

Chapter Number	Title	Description
8.	C++ Language Implementation	Describes the differences from the C Language Implementation in the areas of data representation, register use, and pragmas.
9.	gcc960 / ic960 Compatibility	Describes the incompatibilities between ic960 and gcc960.
10.	Position Independence and Reentrancy	Provides information on writing i960 processor applications that require position-independent or reentrant programs.
11.	Initializing the Execution Environment	Describes the initialization process for the i960 processor execution environment, including the startup assembly-language routine, configuration files, and associated options.
12.	Optimization	Describes the different ways in which the compiler can optimize your program and explains ways to control optimization.
13.	Caveats	This chapter provides useful programming tips on: • Aliasing assumptions • Alignment assumptions • Volatile object • Known problems • C version incompatibilities • Troubleshooting
14.	Messages	Describes the diagnostic messages that the compiler produces.

Audience Description

This manual assumes that you are familiar with the i960 processor architecture, C/C++ and assembly language programming, and your host computer's operating system.

Licensing and Copyrights

Refer to the *i960 Software Tools License Guide* for licensing and copyright information.

UNIX and Windows Conventions

This manual tells you how to use the compiler in both UNIX and Windows 95/NT systems. This manual uses the following conventions:

- Command-lines appear without a preceding prompt.
- Directory paths use the UNIX forward slash (`/`) rather than the Windows backslash (`\`) for pathnames.
- Environment variables are referenced using the UNIX dollar-sign (e.g., `$I960BASE`), not the Windows `%` character (e.g,.`%I960BASE%`)

NOTE. *In UNIX, only the dash (*`-`*) is accepted as a prefix for a command-line option. In Windows, both the (*`-`*) and the (*`/`*) are accepted as a prefix for a command-line option.*

Customer Service

If you need service or assistance with CTOOLS, see your *Getting Started with the i960 Processor Development Tools* manual.

Where Do You Go From Here?

If you installed the CTOOLS GNU interface, go to Chapter 2 “gcc960 Compiler Driver” for information on using the gcc960 compiler driver. If you installed the CTOOLS/960 interface, go to Chapter “ic960 Compiler Driver” for information on using this driver. Once you are familiar with the compiler driver interface, you are ready to read Chapters 4, “Program-Wide Analysis and Optimization” through 6, “gcdm Decision Maker Option”, where you learn how to use some of the more advanced features of the compilation system, including whole program optimizations, profiling, and using the gcdm global decision maker program.

gcc960 Compiler Driver 2

This file describes how to use the gcc960 driver program to control the compilation system. Topics include:

- "Controlling the Compilation System with gcc960"
- "gcc960 and Environment Variables"
- "gcc960 and File Use"
- ".gld Files"
- "gcc960 Options"
- "Option Arguments and Syntax"

Controlling the Compilation System with gcc960

gcc960-style translation and linking requires use of the gcc960 driver, preprocessor, compiler, assembler, and linker.

The gcc960 compiler driver (`gcc960.exe` in Windows, `gcc960` on UNIX) controls the preprocessor (`cpp.exe` in Windows, `cpp.960` on UNIX) and the compiler (`cc1.exe` in Windows, `cc1.960` on UNIX). Starting with CTOOLS release 6.0 gcc960 also controls the new C++ compiler (cc1plus.exe in Windows, cc1plus.960 on UNIX). It can also invoke the assembler, linker, and gcdm960 optimization decision maker. The command-line options and environment variables, described later in this file, allow you to control the compilation.

The drivers invoke the appropriate modules to compile a file based on filename extensions.

- Files with names ending with `.cc`, `.cpp`, and `.cxx` are taken as C++ source to be preprocessed and compiled. In UNIX, filenames ending with `.C` (uppercase) are treated as C++ source to be preprocessed and compiled.
- Files with names ending with `.ii` are taken as preprocessed C++ source to be compiled
- Files with names ending in `.c` are taken as C source to be preprocessed and compiled.
- Files with names ending in `.i` are taken as preprocessor output to be compiled.
- Compiler output files plus any input files with names ending in `.s` are assembled.
- Input files with names ending in `.S` (uppercase) are preprocessed and then assembled. (UNIX only.)
- The resulting object files, plus any other input files, are passed to the linker to produce an executable.
- Program-wide and profile-directed optimizations can be performed during the link step. For an overview of this capability, see "Program-Wide Analysis and Optimization".

Invoking the Compiler with gcc960

The gcc960 command-line syntax is:

`gcc960 [-`*`option`*`]... [`*`path/`*`]`*`filename`*` ... [@`*`response-file`*`]`

`gcc960` is the compiler driver executable filename.

`option` is a compiler option. Case is significant in options and their arguments. Multiple single-character options cannot be grouped: `-dr` is different from `-d -r`. When two or more options contradict each other, the right-most option in the command line takes precedence. For example, the following command line sets the value of the symbol `L` to `132`:

```
gcc960 -DL=80 -DL=132 proto.c
```

NOTE. *Note that the gcc960 compiler driver does not check the command line options for validity. Invalid options are ignored without producing a warning message.*

On UNIX, the compiler recognizes a letter preceded by a hyphen (`-`) as an option. In Windows, the compiler recognizes a letter preceded by either a hyphen (`-`) or a forward slash (`/`) as an option. For example, `-A` specifies the architecture option for UNIX or Windows. However, on a Windows system, you can also use `/A` to specify the architecture option.

`path` identifies the directory containing the file named by *`filename`*. Not specifying *`path`* for a *`filename`* causes gcc960 to search in the current directory. Each *`filename`* not in the current directory requires a separate *`path`* specification.

NOTE. *Although Windows file pathnames require backslashes (`\`), this manual shows paths using the forward slash required by UNIX (`/`).*

`filename` is the name of a source, preprocessed source, assembly-language, object module, or other file (e.g., linker directive file) to be processed by the compilation system. The gcc960 command line allows specification of more than one *`filename`*.

@response-file Open the named response file and read in its contents as if they had been typed on the command line. Response files are a convenient way to store commonly-used command line options, and a way to get around the 128-character line length limit in Windows

Response files can contain comments. Lines whose first non-whitespace character is # are treated as comment lines, and ignored.

gcc960 Sample Command Lines

This section provides examples of how the compiler is commonly invoked. All these examples assume that you have C source files named `t1.c` and `t2.c` or C++ source files name `t1.cc` and `t2.cc`. All examples assume that you are generating code for the i960 CA architecture.

Preprocessing a Source File

To preprocess a source file to stdout, use the command:

```
gcc960 -E t1.c
```

or

```
gcc960 -E t1.cc
```

`-E` informs the compiler to preprocess the source file.

Generating a Preprocessed Source File

To generate a preprocessed C/C++ source file use the following command. The command generates a preprocessed source file named `t1.i` or for C++ `t1.ii`.

```
gcc960 -E t1.c -o t1.i
```

or

```
gcc960 -E t1.cc -o t1.ii
```

`-E` instructs the gcc960 compiler to preprocess the source file.

`-o` *`filename`* instructs the gcc960 compiler to redirect output to *`filename`*.

Generating Assembly Code

This example generates assembly code for the i960 CA architecture. The command lines below each generate an assembly language file named `t1.s`.

```
gcc960 -S -ACA t1.c
```

or

```
gcc960 -Felf -S -ACA t1.cc
```

`-Felf` specifies ELF object module format, which is required for C++. The default object module format is b.out.

`-S` instructs the compiler to generate assembly code.

`-ACA` specifies the i960 CA architecture.

Generating an Object Module with Debug Information

To generate a object module with debug information, use the following command.

```
gcc960 -c -g -ACA t1.c
```

or

```
gcc960  -Felf -c -g -ACA t1.cc
```

`-g` instructs the compiler to generate debug information.

`-c` instructs the compiler to generate an object file.

Generating an Executable

To generate an absolute module (executable file) for a Cyclone board with a CA processor, use the following command.

```
gcc960 -ACA -Tmcycx -g -O t1.c t2.c -o test
```

or

```
gcc960  -Felf -ACA -Tmcycx -g -O t1.cc t2.cc -o test
```

The above command compiles the modules t1.c and t2.c and links them with appropriate libraries to generate an absolute module targeted for a Cyclone i960 Cx evaluation board.

`-Tmcycx` use the linker directive file for a Cyclone i960 Cx evaluation board.

`-O` causes the compiler to perform some basic optimizations on the generated code.

`-o test` instructs the compiler to name the generated executable test.

gcc960 Linker Options

When you do not specify a target with the `T`*`target`* option, gcc960 does not attempt to link programs for a specific target board. Unless otherwise specified source files with recognized extensions (e.g., `.cc`, `.s`) are compiled and/or assembled, and the following linker command is issued:

```
gld960 -AKB $G960BASE/lib/crt960.o file.o... -lqf -lc -lm
```

To link for a different target, you can change the crt (startup) file and specify board and monitor support libraries.

To link for another environment, the options `crt` and `nostdlib` prevent gcc960 from including the default startup files or libraries in the link, allowing them to be fully specified by the user. For example:

```
gcc960 -crt -nostdlib mycrt.o file.o... -lc -lmylib
```

You can invoke gcc960 to create object files in either the b.out, COFF or ELF object module format. The compilation system accepts the `Fcoff` option to generate COFF and the `Felf` option to generate ELF; these options override the gcc960 driver's default format option, which is `Fbout`.

.

NOTE. *ELF is the only object format supported when using C++*

Table 2-1 lists the linker options that gcc960 passes directly to the linker.

Table 2-1 Linker Options Accepted by gcc960

Option	Name	Description
e	Entry point	defines an entry point other than the default for beginning execution of the program.
gcdm	Decision Maker	invokes gcdm960 decision maker.
l	Archive file	specifies an archive file as input.
L	Library search	adds directories to search for libraries, configuration files, and startup object files.
r	Relocation	retains relocation information in the output object file.
s	Strip	strips line-number entries and symbol-table entries from the linker's COFF output file.
u	Unresolved Symbol	introduces an unresolved symbol, causing the linker to search symbol tables for resolution of the reference.
X | x	Compress	X removes all symbols from the output symbol table; x removes only local symbols.
y	Trace symbol	traces a symbol; indicates object files where it appears and provides other information about the symbol.
z	Time stamp	suppresses COFF time stamp in linker output file.

gcc960 and Predefined Macros

Predefined macros within a program can act as constants during execution or as values in conditional-compilation statements. Predefined macros include ANSI C standard macros and macros specific to the i960 processor architecture. The U (Undefine) option removes i960 processor-specific macros but not ANSI C standard macros.

The following macros are available in accordance with the ANSI C standard for C, as described in the book, *C: A Reference Manual*:

```
__DATE__    __FILE__    __LINE__    __TIME__    __STDC__
```

The following macros are predefined by the compilation system when invoked with the gcc960 driver program:

`__GCC960_VER` is defined to a decimal number that can be used to check the version number of the compiler. The number is expressed in decimal as *MmmPPPP*, where *M* is the major version number, *mm* is the minor version number, and *PPPP* is an internal version number that is used to track the patch level. So, for example, R6.0 patch level 4032 would have `__GCC960_VER` defined to be 6004032.

`__i960` indicates the i960 processor environment. The compiler defines `__i960` automatically. This macro can be used to identify the parts of a program specific to the i960 processor.

`__i960`*xx* indicates the i960 processor instruction set in use. The compiler automatically defines the `__i960`*xx* macro. The *xx* is `SA`, `SB`, `KA`, `KB`, `CA`, `CF`, `JA`, `JF`, `JD`, `JT`, `HA`, `HD`, `HT`, `RD`, `RP`, `RM`, `RN`, or `VH`. Definition of *xx* depends on the specific i960 processor instruction set specified by the `A` (Architecture) option.

`__PIC` indicates that the generated code is position-independent. The `mpic` (Generate-for-position-independent-code) option causes the `__PIC` macro to be defined.

`__PID` indicates that the generated data is position-independent. The `mpid` (Generate-for-position-independent-data) option causes the `__PID` macro to be defined.

`__i960_ABI__`

indicates that the generated code is 80960 ABI-Conformant. The `mabi` option causes this macro to be defined.

`__i960_BIG_ENDIAN__`

indicates that the generated code is arranged for big-endian address space. The `G` (Big-endian) option causes this macro to be defined.

`__STRICT_ANSI__`

indicates that C constructs not conforming to the ANSI standard should be flagged. The `ansi` (ANSI) option causes this macro to be defined.

`__CHAR_UNSIGNED__`

indicates that the plain `char` type are treated like the `unsigned char` type. This is the default.

gcc960 and Environment Variables

Environment variables specify default directories for input files, temporary files, libraries, the assembler, and the linker. The compilation system uses the following environment variables to set defaults:

Table 2-2 gcc960 Interface Environment Variables

Name	Purpose
G960AS	Specifies an alternate pathname for the assembler. Default is G960BASE/bin/gas960 (G960BASE\bin\gas960.exe in Windows).
G960BASE	Specifies top-level directory containing the bin, include, and lib subdirectories. G960BASE is necessary for every phase of compilation and linking. The compiler driver uses G960BASE/lib to invoke the preprocessor and compiler. The driver uses G960BASE/bin to invoke the assembler and linker. The preprocessor uses G960BASE/include to find include files. The linker uses G960BASE/lib to find libraries, startup modules, and configuration files. G960BASE also sets defaults for other environment variables in this list. Use these other environment variables to override the paths from G960BASE.
G960BIN	Specifies an alternate pathname for binary files, such as the assembler and linker. If set, G960BIN overrides G960BASE/bin.
G960CC1	Specifies an alternate pathname for the C compiler. The default is G960BASE/lib/cc1.960. (G960BASE\lib\cc1.exe in Windows.)
G960CC1PLUS	Specifies an alternate name for the C++ compiler when using the gcc960 driver. The default pathname is G960BASE/lib/cc1plus.960 (G960BASE\lib\cc1plus.exe in Windows).
G960CPP	Specifies an alternate pathname for the C preprocessor. The default is G960BASE/lib/cpp.960. (G960BASE\lib\cpp.exe in Windows.)
G960INC	Specifies the include file directory. The default is G960BASE/include.
G960LD	Specifies an alternate linker pathname. The default is G960BASE/bin/gld960 (G960BASE\bin\gld960.exe in Windows.)
G960LIB	Specify library search path(s). The default is G960BASE/lib.
G960PDB	Specifies the program database directory for whole-program and profiling optimizations.
G960TMP, TMP, or TMPDIR	Specifies the directory used for temporary work files. Set it to the name of your temporary file directory.
I960ERR	Windows variable that enables you to redirect errors to stderr rather than stdout (the default). To use this capability, set I960ERR to any string, as in: set I960ERR="Enable stderr"

gcc960 and File Use

The compiler, assembler, and linker all use filenames specified on the gcc960 command line to find and create input and output files. In addition, translation and linking require temporary work files.

Input Files

The gcc960 command line allows filename inputs that support specification of assembly-language files, preprocessed source files, C/C++ source files, object files, and libraries. The compiler driver determines the type of each input file by the filename extension, as follows:

filename`.c`	indicates a C source file that can contain macros and preprocessor directives.
filename`.cc, .cpp, .cxx`	indicates a C++ source file that can contain macros and preprocessor directives.
filename`.C`	indicates a C++ source file that can contain macros and preprocessor directives (UNIX only).
filename`.i`	indicates a preprocessed C source file.
filename`.ii`	indicates a preprocessed C++ source file.
filename`.s`	indicates an assembly-language source file.
filename`.S`	indicates an assembly-language source file that can contain preprocessor macros and directives.

The driver passes any other filename to the linker. The linker then determines whether the file is an object file, library, or configuration file.

Input files not needed for processing are not processed. For example, if you specify an assembly-language (*filename*`.s`) file and also specify the `S` (Assembly) compile into assembly code option, gcc960 takes no action on the assembly-language file.

Include Files

The gcc960 command line allows insertion of text from include files using the `#include` preprocessor directive.

The `I`, `I-` and `I.` options affect the directories that are searched for the file specified in the `#include` directive. These options are described in detail in the *Option Arguments and Syntax* section. In the absence of the `I` option, gcc960 searches the current directory for include files followed by the `G960BASE/include` directory.

NOTE. *The include files `icache.h`, `dcache.h`, and `timer.h` used for on-chip cache and timer control are not supported with the `-ARP` option.*

Output Files

Specifying the options `-E`, `-S`, or `-c` causes the compilation system to produce output of the last phase that completes for each primary input file: preprocessed source file, assembly-language file, or an unlinked object file respectively. If no errors occur during processing, the output files created by these options are usable as input to a future gcc960 invocation. Table 2-3 lists the compilation phases and their inputs and outputs.

Specifying the `clist` option generates a listing. gcc960 produces a separate list file for each primary C/C++ source file. The list file is named by replacing the C or C++ filename extension with `.L`.

Specifying the `-M` option causes the preprocessor to output rules describing the dependencies of each source file, suitable for use with a "make" utility. The `clist` and `-M` options are described in detail in "Option Arguments and Syntax".

Table 2-3 Intermediate Inputs and Outputs

Last Phase Completed	Option	Inputs	Outputs
preprocessing	`M`, `E`	C/C++ source files	display on standard output
compilation	`S`	C/C++ source files preprocessed files	assembly-language file listing files
assembly	`c`	C/C++ source files preprocessed files assembly files	unlinked object files listing files
linking	(default)	C/C++ source files preprocessed files assembly files unlinked object files relinkable object files libraries configuration files	list files executable file relinkable object file

When specifying only one primary input file, the `o` (Output) option names a single output file. Specifying multiple primary input files, or not specifying an output filename, causes gcc960 to use the primary input filenames to derive corresponding output filenames with the form *`filename.e`*, where:

`filename`	is the primary input filename without its extension.
`e`	is a single-letter extension indicating the contents of a file, as follows:
`s`	indicates an assembly-language file from the `S` option.
`o`	indicates an object file from the `c` option.
`L`	indicates a listing file from the `clist` option.

Unless otherwise specified, the destination directory for any output file is the current working directory. If *`filename.e`* already exists in the destination directory, the compilation system overwrites the existing file.

The filename `a.out` is the default for the executable COFF object file from the linker, in the absence of an Output option. For ELF files, the default is `e.out` and for bout files, the default is `b.out`.

The following examples illustrate the creation and use of output filename extensions:

- The command `gcc960 -c -clist s proto.c proto1.i` produces the object files `proto.o` and `proto1.o` and the listing files `proto.L` and `proto1.L`.
- The command `gcc960 -c -o proto_v1.o -clist s proto.c` produces the object file `proto_v1.o` and the listing file `proto.L`.
- The command `gcc960 -ACA -Tmcycx proto.c` produces the executable file `b.out`.

.gld Files

The `.gld` files provide a convenient mechanism for specifying default options to the compiler and linker. It also provides a mechanism for specifying the startup file and the libraries to be linked in. These files are meant to be used with the gcc960 interface to the tools (GLD is an acronym for gcc960 linker directive file even though it can be used to pass options to the compiler as well).

By default, the installation program places several `.gld` files in the directory `$G960BASE/lib.` These files have been written for the Cyclone evaluation boards. To illustrate, the sample `.gld` file given below is written for the Cyclone i960 Cx processor-based evaluation board.

Example 2-1 Sample .gld File

```
gcc:-ACA

crt:%{!crt:%[~]/lib/%{mpid:%{G:crt960_e.o}%{!G:crt960_p.o}}
%{!mpid:%{G:crt960_b.o}%{!G:crt960.o}}}

ld:%{!Ttext:-Ttext 0xA0008000}%{*: -defsym
_heap_size=0x20000;_heap_base=(_end+0xf)&~0xf;_heap_end=_heap_base+
_heap_size-1;_stackbase=(_heap_end+0x40)&~0x3f -defsym
fpem_CA_AC=0x100}

lib:%{!nostdlib:-lmn -lll}
```

In the `.gld` file, you can place any options that the tools accept on the command line. The `.gld` file in Example 2-1, includes options for the gcc960 compiler driver and linker.

The command in the `gcc:` section defines the architecture setting for the gcc960 compiler driver. This setting is used throughout the compilation process. The options following `gcc:` are treated in the same fashion as if they were specified on the gcc960 invocation line.

The commands in the `crt:`, `ld:`, and `lib:` sections are written conditionally so that they interact with gcc960 command line switches. For example, the `lib:` section indicates that the linker should be involved with the `-lmn` and `-lll` options, unless the gcc960 `-nostlib` option appears on the command line. These sections determine the startup code, linker options and the libraries that are passed to the linker.

The `crt:` section is used to specify the startup code. In the example given above, if the `-crt` option has not been specified on the compile line, then the compiler driver uses the following for the startup code.

`[G960BASE]/lib/crt960_e.o`	if both `-mpid` and `-G` options are on
`[G960BASE]/lib/crt960_p.o`	if `-mpid` option is on `-G` is off
`[G960BASE]/lib/crt960_b.o`	if `-mpid` option is off and `-G` is on
`[G960BASE]/lib/crt960.o`	if both `-mpid` and `-G` options are off

The `ld:` section contains options that are passed to the linker. This example includes commands to place the `.text` section at address `0xA0008000`, and defines symbols used to specify the heap and stack locations.

The `lib:` section in the above example is used to specify that the compiler driver should pass the options `-lmn` and `-lll` to the linker if the `-nostdlib` option is off. This causes the linker to include the monitor and the low-level libraries in its search path to look up unresolved symbols.

For more information on the linker directives used in this sample file, see the *i960 Processor Software Utilities User’s Guide*.

gcc960 Options

This section describes the gcc960 compiler driver options that allow control of various aspects of compilation:

Input processing and output	The `c`, `E`, and `S` are the Stop-after options. They stop the translation and linking process after the preprocessing, syntax checking, compilation, or assembly phase. A Stop-after option causes the compilation system to save the intermediate output of the last phase to execute. The `C` (Keep-comments) and `M` (Mix) options affect the contents of the output file. The `o` (Output) option allows specification of the output filename.
Specifying included source text	The `i` (Preinclude) and `I` (Searchinclude) options prepend and find include files of C/C++ source text.
Defining macros	The `D` (Define) and `U` (Undefine) options allow specification of macros for conditional compilation.

Control contents of generated object code
The `A` (Architecture), `Fcoff`/`Felf`/`Fbout` (Object-format), `F` (Fine-tune), `f` (Optimize), `g` (Debug), `G` (Generate), and `O` (Optimization-level) options control the instruction set, object format, debug information, and optimization level.

Whole-program optimizations
The `fdb` (Program Database), `fprof` (Instrumentation), and `gcdm` (Decision Maker) options allow for creation and use of information necessary for advanced optimizations involving multiple modules and/or execution profiles. See "Program-Wide Analysis and Optimization" for an overview of whole-program and profile-driven optimization.

Provide Information on the compiling process
The `a` (ANSI) option affects messages the compiler produces about C/C++ syntax and semantics. The `v` (Verbose), `V` (Version), and `v960` (Version-exit) options display information about preprocessor, compiler, assembler, and linker options. The Version option displays the versions of each compilation component and the host operating system. The `W` (Warnings) option allows fine control of the level of warnings emitted.

Option Arguments and Syntax

Some compiler driver options take arguments. Case is significant in options and arguments. A few options allow whitespace between the option and its argument; this whitespace is shown in Table 2-4.

The options and arguments have default settings. In most cases, the option is "off," that is, not in effect. Default settings of options and arguments are summarized in Table 2-4 and further discussed in the detailed description of the option. Some option defaults are affected by environment variables, which are described in the *Getting Started* manual.

This file uses the following notation:

`[item]`	Square brackets indicate that the enclosed item is optional.
`. . .`	Horizontal ellipses indicate that you can use multiple instances of the preceding item.

Table 2-4 gcc960 Option Summary (Sheet 1 of 3)

Option	Name	Purpose	Default
A*arch*	Architecture	Select the instruction set.	AKB
ansi	ANSI	Detect non-ANSI source.	off
C	Comments	Keep comments in preprocessor output.	off
c	Create Object	Stop after creation of object file.	off
clist *arg* ...	Create listing	Create a listing.	off
crt	Startup	Do not use standard startup file.	Use default
D *macro* [=*value*]	Define	Define *macro*. (default is one)	*macro* undefined
d*arg*	Definitions	Control macro processing.	off
E	Preprocess	Preprocess source; terminate.	Do not stop
Fbout | Fcoff | Felf	Format	Generate b.out, COFF or ELF object format.	Fbout
fdb	Database	Build program database directory (PDB).	No database
fprof	Instrument	Compile with instrumentation; build PDB.	No instrumentation
f[no-]*arg*	Fine-Tune	Enable or disable an option.	Varies with option
G	Big-endian	Generate big-endian code.	off
g[*level*]	Debug	Include debug information in objects.	No debug info
gcdm,*arg...*	Decision-maker	Invoke gcdm960 decision-maker.	off
h[elp]	Help	Display invocation help; terminate.	off
I *directory*	Searchinclude	Search *directory* for include files.	off
I- | I.	I-dash, I-dot	Control include-file search order.	off

Table 2-4 gcc960 Option Summary (Sheet 2 of 3)

Option	Name	Purpose	Default
ic960	iC-960	Accept iC-960 source dialect.	off
imacros *filename*	Macros File	Specify macros file for preinclusion.	off
include *filename*	Preinclude	Prepend text to source files.	off
L *directory*	Library Directory	Specify directory for library search.	off
M | MD | MM | MMD	Make	Generate make tool output.	off
m*string*	Machine	Machine-specific options.	Varies with option
nostdinc	No Standard Include	Exclude standard include (header) files.	off
nostdlib	No Standard Libraries	Excludes standard libraries.	off
O [*level*]	Optimize	Specify optimization level.	O0
o *filename*	Output	Name output file.	Varies with object format
P	Preprocess Output	Preprocessor output control.	off
pedantic [-errors]	Pedantic	Controls ANSI error and warning generation.	off
S	Assembly	Stop after assembly-language output.	off
save-temps	Save Intermediate	Save intermediate files.	Do not save
T*file*.gld	Target	Specify configuration file.	off
traditional	Traditional	Allow traditional C.	off
trigraphs	Trigraphs	Support ANSI trigraphs.	off
U *macro*	Undefine	Undefine *macro*.	off
V	Version	Display version information.	No display
v960	Version-exit	Display version information and exit.	off
v	Verbose	Display invocation information.	No display
W [no-]*arg*	Warnings	Enable/disable a warning.	Varies

Table 2-4 gcc960 Option Summary (Sheet 3 of 3)

Option	Name	Purpose	Default
w	No Warnings	Inhibits warnings.	off
Z*directory*	Program database	Specify location of program database directory (PDB).	G960PDB specifies PDB

A (Architecture)

Selects instruction set.

`A`*`architecture`*

`architecture` is one of:

`SA`, `SB`, `KA`, `KB`, `CA`, `CF`, `JA`, `JD`, `JF`, `JT`, `HA`, `HD`, `HT`, `RD`, `RP`, `RM`, `RN`, or `VH`.

Default

By default, the compiler uses the i960 KB architecture.

Discussion

Use the `A` (Architecture) option to specify the target instruction set. See also the `-mcore0`, `-mcore1`, `-mcore2`, and `-mcore3` options that let you generate code that is compatible with multiple i960 processor types.

Note that with release 5.1 and later using the `-ARP` or `-ARD` options generates code that is compatible with current and proposed future variations on the i960 Rx architecture.

You can use predefined macros in your source text to conditionally compile code for the selected architecture. The compiler defines a preprocessor macro indicating the selected architecture. The preprocessor macro takes the form:

`_ _i960`*`xx`*

`xx` is `SA`, `SB`, `KA`, `KB`, `CA`, `CF`, `JA`, `JD`, `JF`, `JT`, `HA`, `HD`, `HT`, `RD`, `RP`, `RM`, `RN`, or `VH`. The compiler selects the value of *`xx`* according to the architecture you specify.

The _ _i960 macro is defined for all architecture selections. Use _ _i960 to identify parts of your program specific to the i960 architecture but not necessarily specific to a particular processor.

In addition, for compatibility with earlier releases, macros of the forms: i960, _ _i960_ _, _ _i960*xx*_ _ and _ _i960_*xx*_ _ are defined.

When you link object modules compiled with incompatible architectures, the linker displays the following warning message:

```
file: architecture i960:XX incompatible with output
i960:YY
```

file is the first file containing incompatible instructions the linker encounters.

XX is one of the two-letter architecture abbreviations.

YY is one of the two-letter architecture abbreviations.

ansi (ANSI)

Disable non-ANSI features.(C-specific option)

Disables features of gcc960 that are incompatible with ANSI C, such as the asm, inline and typeof keywords, and nonstandard macros such as I80960. ansi also enables the ANSI trigraph feature.

See the table shown under the traditional option for a summary of the macros defined when the ansi or traditional options are used.

The alternate keywords _ _asm_ _, _ _inline_ _ and _ _typeof_ _ continue to function even if you specify ansi. You would not want to use them in an ANSI C program, of course, but it can be useful to put them in header files that might be included in compilations done with ansi.

ansi does not cause non-ANSI programs to be rejected with errors. For that, the pedantic-errors option is required in addition to ansi.

The macro `_ _STRICT_ANSI_ _` is predefined when the `ansi` option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

C (Comments)

Keep comments.

Directs the compiler not to discard comments, and to pass them through to the preprocessor output file. Comments in arguments of a macro call are copied to the output before expansion of the macro call. Used with the `E` option.

c (Create Object)

Stop after creation of object file.

Directs the compilation system to stop after creating the object file(s). Object files are named by replacing `.c`, `.cc`, `.cpp`, `.cxx`,`.i`, `.ii`, `.S`, or `.s` with `.o` at the end of the input filenames. If you specify an object file as input, the compiler does nothing with the file.

clist (Listing)

Creates a listing.

clist *arg...*

Generates a listing of the types described below. The list file has the name *filename*.L where filename is the name of the original C/C++ source file. Multiple arguments are allowed. *arg* is one of the following letters:

s	lists the primary source text, that is, source text from files named on the command line.
i	adds source text from included files to the primary source text listing.
o	adds the assembly language generated by the compiler to the listing file.
m	adds expanded preprocessor lines to the primary source text listing.
c	adds conditionally noncompiled source text to the primary source text listing.

crt (Startup)

Omit standard startup file.

Do not use the standard startup file when linking. A replacement crt file should come first in the list of object files. For all i960 processor types except the Rx, the standard startup file is crt960.o. For i960 Rx processors, the standard startup file is crtrp.o.

D (Define)

Defines a macro.

D *macro*[=*value*]

With no =*value*, defines *macro* as 1. (This is exactly the same as D *macro*=1.)

`D` *`macro`*`=`*`value`*

Defines *`macro`* as *`value.`*

d (Definitions)

Control macro processing.

- `dD` Tells the preprocessor to pass all macro definitions into the output, in their proper sequence in the rest of the output.
- `dM` Tells the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing.
- `dN` Like `dD` except that the macro arguments and contents are omitted. Only `#define` *`macro`* is included in the output.

These should be used only with `-E`, and they affect preprocessor output.

E (Preprocess)

Run only the C/C++ preprocessor.

Directs compilation system to preprocess all the C/C++ source files specified and send the results to standard output.

Fbout | Fcoff | Felf (Format)

Specifies the object file format.

`Fbout`	specifies the b.out object format. This is the default. You can add the `g` option to specify the style of symbolic-debug symbols created. Note that you cannot use this option with the `-ARP` or `-ARD` architecture setting or with C++ modules.
`Fcoff`	specifies the COFF object format, and causes the assembler to be invoked as gas960c, rather than gas960. You can add the `g` option to specify the style of symbolic-debug symbols created. The compiler does not support using the object module format with C++.
`Felf`	specifies the ELF object format, and causes the assembler to be invoked as gas960e, rather than gas960. If you add the `g` option, the DWARF style of symbolic-debug symbols is used.

fdb (Database)

Builds optimization database.

All modules subject to program-wide optimization must be initially compiled with the `fdb` option. This option causes the insertion of program database information in the object modules, and it requires a minimum module-local optimization level of `O1` (although higher module-local optimization levels are allowed).

This option does not otherwise change the code or data generated for the object modules. It simply makes optimization information collected during the initial compilation available to gcdm.

Before using the `fdb` option, you should read "Program-Wide Analysis and Optimization", and "gcdm Decision Maker Option".

If you intend to use execution profiles when optimizing your application, you should read "Profile Data Merging and Data Format (gmpf960)".

fprof (Instrument)

Instruments code for profile creation.

This option inserts execution profile instrumentation code into the code generated during compilation, so that when the linked program is executed, a profile can be collected.

Before using the `fprof` option, read "Program-Wide Analysis and Optimization" through "gcdm Decision Maker Option" for general strategies on using CTOOLS profiling and other optimization features.

This option enables the `fdb` option, which instructs the compiler to insert program database information into the object modules and create the program database. `fprof` also requires a minimum module-local optimization level of O1 (although higher module-local optimization levels are allowed).

When you use the `fprof` option, a special profiling library required for profile collection (`libqf`) is linked automatically. If your target environment does not support file I/O, you must explicitly link an alternate profiling library (`libq`). The profiling libraries provided are described in Chapter 2 of the *i960 Processor Library Supplement*.

Note that compiling with the `fprof` option creates object modules useful only for collecting a profile. If you compile with `fprof` and later do not want a profile, you must then use substitutions to replace every instrumented module in `prog`, or you must recompile the modules without the `fprof` option. See "Program-Wide Analysis and Optimization" for more information on this subject.

f (Fine-Tune)

Enable or disable specific options.

In most cases, you will want to optimize code automatically by using the various `O` optimizations. (See the section on the `-O` option.) In some cases, however, you may want to enable or disable specific features for a given optimization level. For example, at optimization level `O0`, you cannot enable instruction scheduling with `fschedule-insns`. As with any optimization process, you should first compile without the option and then recompile with the desired option enabled/disabled. You can then compare the generated assembly code and see if adding/removing the option produced the desired result.

Before using any of these options, read “Program-Wide Analysis and Optimization” through “gcdm Decision Maker Option” for an overview using the compilation system’s performance features.

`f[no-]access-control`

Enable/Disable all access checking. This is normally used to work around access control bugs. `faccess-control` is the default. This is C++ specific option.

`f[no-]asm`

Do [not] recognize `asm`, `inline` or `typeof` as a keyword. These words can then be used as identifiers. You can use `_ _asm_ _`, `_ _inline_ _` and `_ _typeof_ _` instead. This option provides compatibility with strict ANSI standards. Do not use this option with C++ files. See also the `-ansi` option.

`f[no-]bbr`
Enable/disable basic block rearrangment. This option is normally used in a second-pass recompilation, but it can also be used in single-pass compilation.

`f[no-]coalesce`
Coalesces memory references into a single larger memory reference, thus taking better advantage of the i960 processor’s burst bus. The compiler only coalesces memory references that can be proven to be contiguous and whose base address can be proven to be aligned properly. `fcoalesce` enables `fshadow-mem`.

`f[no-]coerce`
Enable/disable byte/short optimization.

`f[no-]cond-mismatch`
Allow/do not allow conditional expressions with mismatched types in the second and third arguments of the `?:` operator. The value of such an expression is void.

`f[no-]condxform`
Performs a special conditional transformation that allows the use of the i960 Jx, Hx, and Rx processors’ sel<cc>, addo<cc>, and subo<cc> instructions. You cannot use this optimization unless the `AJx`, `AHx`, or `ARx` option is specified.

`f[no-]conserve-space`
Allocate uninitialized global variables into the common segment, as C does. This saves space in the executable at the cost of not diagnosing duplicate definitions. `fno-conserve-space` is the default. This is a C++ specific option.

`f[no-]constprop`
Performs constant propagation and folding. This optimization replaces uses of variables known to have a constant value with the constant value, allowing other optimizations to see these constants and possibly generate more optimized code.

`f[no-]copyprop`
Performs copy propagation. This optimization replaces uses of registers that are destinations of register to register copies with the source register (when possible). This allows unnecessary copies to be deleted later in the compilation.

`f[no-]cse-follow-jumps`
During common subexpression elimination (CSE), scan through jump instructions in only certain cases. This is not as powerful as completely global CSE, but allows for faster compilation.

`f[no-]cse-skip-blocks`
Enable/disable a limited form of global CSE.

`f[no]dollars-in-identifiers`
Accept “$” in identifiers. ANSI C forbids “$” in identifiers. `fno-dollars-in-identifiers` is the default for C and `fdollars-in-identifiers` is the default for C++.

`f[no-]expensive-optimizations`
Perform/skip a number of minor optimizations that are relatively expensive. This option is enabled with optimization level `O2`.

`f[no-]fancy-errors`
Display/do not display C/C++ source line and caret (^) with error messages.

`f[no-]float-store` Store/do not store floating-point variables in registers, and do not perform common sub-expression elimination on floating point expressions.

`f[no-]force-addr` Force/do not force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code.

`f[no-]for-scope` Limit the scope of variables declared in a `for-init` statement to the for loop itself, as specified by the draft C++ standard. When you specify `-fno-for-scope`, the scope of variables declared in a for-init-statement extends to the end of the enclosing scope, as was the case in old (traditional) implementations of C++. `for-scope` is the default. This is a C++ specific option

`f[no-]inline-functions` Inline/do not inline all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth inlining in this way. When all calls to a given function are inlined, and the function is declared static, then the function is normally not output as assembler code in its own right.

`fint-alias-ptr` indicates to the compiler that pointer objects may be referenced as 32-bit integers and vice versa.

`fint-alias-real` indicates to the compiler that `float`, `double`, and `long double` objects (or parts thereof) may be referenced as 32-bit integers and vice versa.

`fint-alias-short` indicates to the compiler that four-byte integer objects may be referenced as two-byte objects and vice versa.

The aliasing options listed above tell the compiler not to use certain kinds of type information when disambiguating memory references, even though ANSI C section 3.3 “Disambiguation Constraints,” allows this.

The rules enforced by the aliasing options are transitive. For example, when user code accesses parts of `double` objects as `short`, then `fint-alias-real` and `fint-alias-short` should both be used.

The rules are also applied recursively to `structs` and unions. That is to say, when `fint-alias-ptr` is in use, then a union that has a member of pointer type is assumed to be aliased by 32-bit integers or by `structs` or unions containing 32-bit integers.

Note that ANSI C 3.3 requires the compiler to assume that `char` references alias all types, so code using `char` pointers is already correct and using these options is not necessary.

Using all three aliasing options effectively disallows all use of type information in memory disambiguation. This is bad both for compiler performance and the efficiency of generated code.

`f[no-]enum-int-equiv` Allow implicit conversion of integer to enumeration types. Normally the compiler allows conversion of enum to int, but not vice versa. `fno-enum-int-equiv` is the default. This is a C++ specific option.

`f[no-]huge-objects`

The implementation of virtual function calls assumes that the size of an object can be represented with a short integer. Use this flag to support virtual function calls for objects that cannot be represented by a short integer. Use this flag only if the compiler requests you to do so. Note that the C++ library sources need to be recompiled with `fhuge-objects` if you plan to link with the C++ libraries. `fno-huge-objects` is the default. This is a C++-specific option.

`f[no-]keep-inline-functions`

Even when all calls to a given function are inlined, a separate run-time callable version of the function is still output.

`f[no-]marry_mem`

Rejoin multi-word moves split apart by `fsplit_mem` (where possible).

`f[no-]memoize-lookups`
`f[no-]save-memoized`

Use heuristics to compile faster. These heuristics are not enabled by default, since they are only effective for certain input files. Other input files compile more slowly. You may use either option to compile using heuristics. These are C++ specific options.

`fmix-asm`

Intermix C/C++ code as comments within the assembly code.

`f[no-]rerun-cse-after-loop`

Re-run common subexpression elimination after loop optimizations have been performed.

`f[no-]sblock`

Enable/disable superblock formation. This option is normally used in a second-pass recompilation, but it can also be used in a single-pass compilation.

`fsigned-char |`
`fno-signed-char`

Make the type `char` be signed, like `signed char` (`fsigned-char`), or make the type `char` be unsigned, like `unsigned char` (`fno-signed-char`). `fsigned-char` is equivalent to `fno-unsigned-char`.

By default, `char` variables are treated as `unsigned`.

`f[no-]schedule-insns`

Attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This allows other instructions to be issued until the result of a previously issued instruction is required.

This option makes debugging more difficult, since the code for multiple C/C++ statements may become intermixed, causing execution to make numerous jumps while single-stepping.

`f[no-]schedule-insns2`

Similar to `fschedule-insns`, but it requests an additional pass of instruction scheduling after register allocation has been done.

`f[no-]shadow-globals`

Shadow memory locations with global register variables where possible. Memory locations that are known not to change are temporarily allocated to registers. This option makes debugging more difficult, since objects allocated in memory may not always be up-to-date.

`f[no-]shadow-mem`

Shadow memory locations with register variables where possible. Memory references whose addresses are known to be the same are temporarily allocated to registers. This option makes debugging more difficult, since objects allocated in memory may not always be up-to-date. `fshadow-mem` is similar to `fshadow-globals`, but its analysis is considerably more sophisticated. In most cases, `fshadow-mem` allows more optimization than `fshadow-globals`, but compile time is slower.

`f[no-]space-opt`

Optimize to reduce the size of the generated code.

`f[no-]split_mem`

Split all multi-word moves into sequences of single word moves to improve copy propagation.

`f[no-]strict-prototype`

Treat a function declaration with no arguments, such as "`int foo ();`", to mean that the function `foo` takes no arguments. `fstrict-prototype` is the default. This is a C++ specific option.

`f[no-]this-is-variable`

Permit assignment to “this”. `fno-this-is-variable` is the default. This is a C++ specific option.

`funsigned-char |`
`unsigned char`

Make the type `char` be unsigned, like `(funsigned-char)`, or make the type `char` be signed, like `signed char (fno-unsigned-char)`. `funsigned-char` is equivalent to `fnosigned-char`.

By default, `char` variables are treated as `unsigned`.

`f[no-]strength-reduce`

Perform loop strength reduction and eliminate induction variables. See the Glossary for more information.

`fsyntax-only`

Check the syntax of C/C++ source file(s), without generating an object file.

`f[no-]thread-jumps`

Test whether a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or to a point immediately following it, depending on whether the condition is known to be true or false.

`f[no-]unroll-all-loops`

Perform the optimization of loop unrolling on all loops. This is not recommended as it increases code size and usually degrades runtime performance. `funroll-all-loops` enables both `fstrength-reduce` and `frerun-cse-after-loop`.

`f[no-]unroll-loops`

Break up a loop into several iterations of the loop body. This typically improves performance, since the loop’s exit condition is not checked for each iteration. In a few cases, however, the increased code size may decrease performance.

This option uses several decision criteria determine how far to unroll a loop. For example, when the loop body is small and there are relatively few iterations, it may choose to completely unroll the loop. For loops with larger bodies and more iterations, it may partially unroll the loop and change the increment counter accordingly. `funroll-loops` enables both `fstrength-reduce` and `frerun-cse-after-loop`.

`fvirtual-opt`

Optimizes the dispatch of virtual functions. This optimization can be used only in a 2-pass scheme. By default, this optimization is not enabled. This optimization can be used only when certain conditions are met. See “Optimizing Virtual Function Dispatch” in Chapter 12 for more details. This is a C++-specific option.

`f[no-]volatile`

Consider/do not consider all memory references through pointers to be volatile.

`f[no-]volatile-global`

Consider/do not consider all references to global variables to be volatile.

`f[no-]writable-strings`	Store/do not store string constants in the writable data segment and make them unique. This is for compatibility with old programs that assume they can write into string constants.

G (Big-endian)

Generate big-endian code.

Compile for a target that uses big-endian memory. This option requires that `Fcoff` or `Felf` be in effect. This option is also passed to gas960c/gas960e and gld960. When `G` is specified, the preprocessor symbol `__i960_BIG_ENDIAN__` is defined.

g (Debug)

Specifies debug information.

`g [`*`level`*`]`

where *`level`* specifies the amount of debug information. Note that the meaning of level varies depending on the object format in use, as described below.

Using `g0` disables debug information. (This is the same as not using the `g` option.)

For b.out and COFF, debug level settings of `g`, `g1`, `g2`, and `g3` all have the same effect: they specify “normal” debug information.

When the default object-file format (b.out) is selected, DBX-style symbolic debug directives suitable for use only with gdb960 are output.

For ELF/DWARF, debug level settings of `g`, `g1`, and `g2` all have the same effect: they specify all DWARF debug information except preprocessor macros.

For ELF/DWARF, a debug level setting of `g3` specifies all DWARF debug information, including preprocessor macros in the debug information. If your debugger (like gdb960) does not make use of preprocessor macro information, you can save space in your object files by dropping to ELF/DWARF debug level 2.

The `g` (Debug) option does not inhibit optimization. When you specify the `g` option but do not specify the `O` (Optimize) option, the optimization level defaults to `O0`.

Specifying an optimization level higher than `O0` can inhibit the effectiveness of the symbolic debug information. For example, if you set a breakpoint on a source line that has been removed during optimization, the breakpoint is never hit. Or if you try to print the value of a variable that has been optimized away, an erroneous value is displayed. In general, as the optimization level increases, the reliability of the symbolic debug information decreases.

When you are using the ELF object module format (`Felf`), `g` causes the compiler to produce DWARF debug information. This debug information format is richer than that of other supported OMFs, and allows more reliable debugging under optimization. However, even with DWARF, there are situations where debugging behavior does not agree with the debugging behavior of unoptimized code.

gcdm,*arg*[,*arg*]... (Decision Maker)

Invoke gcdm960 optimization decision maker.

```
gcdm,arg[,arg]...
```

The gcdm option provides a high level of automation for whole-program or profile-driven optimization processes. The compiler driver and the linker both use the gcdm option and its arguments.

The gcdm option is flexible and powerful, and therefore requires a certain level of understanding in order to use it effectively. For these reasons, it is documented in a separate file ("gcdm Decision Maker Option") in this manual. Before using the gcdm option, you should read "Program-Wide Analysis and Optimization", and become familiar with the information in "Profile Data Merging and Data Format (gmpf960)".

I (Searchinclude)

Specifies include file directory.

`I`*`directory`*

Adds *`directory`* to the end of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. When you use more than one `I` option, the directories are scanned in left-to-right order; the standard system directories come after.

I- | I. (Include-dash, Include-dot)

Controls search order and paths.

`I- | I.`

Any directories specified with `I` options before the `I-` option are searched only for `#include "`*`file`*`"`; they are not searched for `#include <`*`file`*`>`.

When additional directories are specified with `I` options after the `I-`, these directories are searched for all `#include` directives. (Ordinarily all `I` directories are searched this way.)

The `I-` option inhibits the use of the current directory as the first search directory for `#include "`*`file`*`"`. The current directory is searched for `#include "`*`file`*`"` only when it is requested explicitly with `I.` (`I`"dot"). It is not searched automatically with `I-`. Specifying both `I-` and `I.` allows you to control which directories are searched before the current one and which are searched after.

ic960 (iC-960 Compatibility)

Accept iC-960 source dialect.

Accept the same C dialect as ic960 R3.0 or later. Note that this does not make the generated code compatible. To make the generated code compatible, the `mic3.0-compat` option is necessary. This is a C-specific option.

imacros (Macros File)

Specifies macros file.

```
imacros file
```

Process *`file`* as input, discarding the resulting output, before processing the regular input file. Because the output generated from *`file`* is discarded, the only effect of `imacros` *`file`* is to make the macros defined in *`file`* available for use in the main input. Any `D` and `U` options on the command line are always processed before `imacros` *`file`*, regardless of the order in that they are written. All the `include` and `imacros` options are processed in the order in that they are written. All `imacros` options are processed before all `include` options.

include (Preinclude File)

Specifies file for preinclusion.

```
include file
```

Process *`file`* as input before processing the regular input file. In effect, the contents of *`file`* are compiled first. Any `D` and `U` options on the command line are always processed before `include` *`file`*, regardless of the order in that they are written. All the `include` and `imacros` options are processed in the order in that they are written. All `imacros` options are processed before all `include` options.

j (Errata)

Specifies processor errata.

```
j num
```

Use the `j` (Errata) option to cause the compilation system to generate code with workarounds for specified processor errata. A *`num`* argument of 1 generates code to work around the Cx processors' DMA errata.

L (Library Directory)

Specifies directory for library search.

`L` ***`directory`***

Adds ***`directory`*** to the list of directories to be searched for libraries. See the *i960 Processor Software Utilities User's Guide* for a complete explanation of the directory search order.

l (Library)

Specifies library for linking.

`l`***`library`***

Search a standard list of directories for a library file named `lib`*`library`*`.a`. The linker uses this file as if it had been specified precisely by name.

Several standard directories are searched, plus any that you specify with `L`.

Normally the files found this way are library files — archive files whose members are object files. The linker handles an archive file by scanning through it for members that define symbols that so far have been referenced but not defined. However, when the file found is an ordinary object file, it is linked in the usual fashion. The only difference between using an `l` option and specifying a filename is that `l` searches several directories. Under normal operation, gcc960 supplies the options `lqf`, `lc`, and `lm` to the linker. For architectures without floating-point support, the option `lh` is also passed to the linker.

M | MD | MM | MMD (Make)

Generate make tool output.

`M`	Tells the preprocessor to output a rule suitable for a make tool describing the dependencies of each source file. For each source file, the preprocessor outputs one make rule whose target is the object filename for that source file and whose dependencies are all the files `#include`d in it. This rule can be a single line or can be continued with `\newline` if it is long. Using this option stops compilation after preprocessing.
`MM`	Like `M`, but the output mentions only the user-header files included with `#include "`*`file`*`"`. System header files included with `#include <`*`file`*`>` are omitted.

The `M` and `MM` options output the dependecy information to stdout.

The `MD` and `MMD` options behave in a fashion similar to the `M` and `MM` options respectively. However, the `MD` and `MMD` options write the dependency information to the file *filename.d* instead of to stdout where *filename* is the name of the C/C++ source file without the filename extension. These options cause a separate dependency file to be generated for each of the C/C++ source files.

These options stop compilation after preprocessing. The `M` and `MM` options also suppress the preprocessor output.

m (Machine-specific Options)

Various options.

`m`*`string`* Specifies a machine-specific option.

`mabi` Generate 80960 ABI-conformant code. This causes the `char` type to be signed, enums to be four bytes in size and signed, and changes default alignment rules for structs and unions. See “C Language Implementation” for more information.

`masm-compat` Generate special Intel pseudo-operations for long compare-and-branch operations. gas960, gas960c, or gas960e do not require these pseudo-ops in order to generate correct code, but the ASM960 R3.5 or earlier assembler generates out-of-range errors for these instructions when this option is not used. This should not be used with gas960, gas960c, or gas960e, because the split compare-and-branch instructions are slower and larger than the combined ones.

`mcave`

Generate all functions as CAVE secondary. When you select `mcave`, the compiler generates special CAVE entries for all functions in the compilation unit. This prepares the functions for link-time compression. The cave entries resemble the following:

```
.section .text
_foo:
     lda     L1,reg
     call    __dispatcher
     ret
.section cave
     .word  L2-L1,0
L1:
     function body
L2:
```

At runtime, the dispatcher decompresses the function bodies and transfers control to them. This mechanism saves runtime memory.

See the discussion of `#pragma cave` in “C Language Implementation” for information on this option.

`mcmpbr |`
`mno-cmpbr`

Generate/do not generate code that uses compare-and-branch instructions whenever possible.

`mcode-align |`
`mno-nocode-align`

Generate/do not generate alignment directives prior to labels that are not entered from above. `mcode-align` is the default when the Cx or Hx architecture is specified.

`mcore0|mcore1|`
`mcore2|mcore3|`

generate code that is compatible with multiple i960 processor types. Additionally, when you use an `-mcore` option, you can include another `-A` switch to generate code that is optimized for a particular architecture, but still compatible with a group of architectures. The table below lists the architectures that are supported by each `-mcore` option and the `-A` options that you can use with them.

Table 2-5 Mcore Supported Architectures

Option Name	Compatible Architectures	Can Be Used With
mcore0	Jx, Hx, Rx	`-AJA`, `-AJD`, `-AJF`, `-AJT`, `-AHA`, `-AHD`, `-AHT`, `-ARD`, `-ARP`, `-ARM`, `-ARN`, or `-AVH*`.
mcore1	Kx, Sx, Cx, Jx, Hx	Any architecture option except `-ARP`, `-ARD`, `-ARM`, or`-ARN`.
mcore2	Jx, Hx	`-AJA`, `-AJD`, `-AJF`, `-AJT`, `-AHA`, `-AHD`, `-AHT` or `-AVH*`.
mcore3	Cx, Jx, Hx	`-ACA`, `-ACF`, `-AJA`, `-AJD`, `-AJF`, `-AJT`, `-AHA`, `-AHD`, `-AHT` or `-AVH*`.

* Note that the big-endian mode is not supported for VH.

`mdouble4` Generate code so that the size and alignment of `double` is the same as `float`.

`mlong-double4` Generate code so that the size and alignment of `long double` is the same as `float`.

NOTE. *The* `mdouble4` *and* `mlong-double4` *options force floating-point arguments to be passed in single-precision format. When your source program explicitly calls functions (such as* `sin` *and* `printf`*) that require double-precision or extended-precision arguments, the arguments passed to these functions are incorrect.*

`mi960_align=`*`n`*

Aligns `struct` data on the byte boundary that is a multiple of *n*. (Legal values are 1, 2, 4, 8, 16.)

`mic-compat`
`mic2.0-compat`

Use ic960 R2.0's rules for size and alignment of types. This option also causes the compiler to use the ic960 compiler's rules for promotion of `char`, `unsigned char`, `short`, and `unsigned short` types at function call and return.

`mic3.0-compat`

Use ic960 R3.0's rules for size and alignment of types and other conventions. These are largely the same as gcc960's, but ic960 R3.0 selects the size of `enum`s based on their value. Additionally, ic960 R3.0 assumes that type char is signed by default, whereas gcc960 assumes it is unsigned. The `mic3.0-compat` option emulates ic960's behavior.

`mleaf-procedures |`
`mno-leaf-procedures`

Generate/do not generate output that contains leaf procedures: these are procedures that may be entered with the `bal` instruction rather than `call`. The linker automatically promotes `call` instructions into `bal` instructions when appropriate. This option makes debugging more difficult. `mleaf-procedures` is the default at `O2` or higher.

`mlong-calls`

Generate all call instructions as `calljx` instead of `callj`. This is used where the distance between the call site and the called function may exceed `callj`'s range. Using this option degrades code execution speed and increases code size.

`mpic`

Generate position-independent references to any objects in the text section. Such objects are functions, `const` file-scope variables, switch tables, and strings. Position independent code references are made relative to the current instruction pointer (IP).

`mpid`

Generate position-independent references to objects in the bss, common, and data sections. Such objects are non-`const` file-scope variables, and strings when the `fwritable-strings` option is used. Position independent data references are made relative to register `g12`. Register `g12` is not used for any other purpose.

`mpid-safe`

Reserve register g12 as the position independent data bias register, but do not generate code for position independent data.

`msoft-float`

Generates output containing library calls for architectures without on-chip floating point support (all except KB, SB). This is set automatically, based on the architecture option.

`mstrict-align |`
`mno-strict-align`

This option determines whether or not the compiler risks generating memory references that are not provably aligned. When `mstrict-align` is disabled, the compiler occasionally generates potentially unaligned references when it seems advantageous to do so. When `mstrict-align` is enabled, sequences of smaller memory references are used instead of larger ones that might not be correctly aligned. The default is on for i960 Cx and Jx processors.

`mstrict-ref-def`

Generate code so that an uninitialized file-scope variable definition causes space to be allocated in the `.bss` section instead of as a `.comm` symbol. This enforces a single unique definition of a variable.

`mtail-call |`
`mno-tail-call`

Generate output that converts (does not convert) `call` instructions immediately followed by `ret` instructions to branches to the call target. While generating faster code, this option makes debugging more difficult. `mtail-call` is the default at `O2` or higher.

`mwait=`*`n`*

Specifies the expected number of wait-states for the memory being used in the target. This can make a difference in which optimizations are cost-effective and in the instruction scheduling optimization. *`n`* must be in the range 0.32.

nostdinc (No Standard Header Files)

Do not use standard header files.

Do not search the standard system directories for header files. Only the directories specified with `I` options (and the current directory, when appropriate) are searched. Using `nostdinc` and `I-`, you can eliminate all directories from the search path except those you specify.

nostdlib (No Standard Libraries)

Do not use standard libraries.

Excludes standard libraries.

O (Optimize)

Specifies optimization level.

`O[`*`level`*`]`

The `O[`*`level`*`]` option specifies the level of optimization as described below.

`O0`	Turns optimization off, and additionally disables default optimizations that may interfere with debugging. This is the default.
`O` or `O1`	These options enable basic optimizations, including: advanced register allocation, common subexpression elimination, loop invariant code motion, expression simplification and instruction combination, jump

optimization, dead-code elimination, and i960 processor-specific peephole optimization. `O1` is equivalent to `O`. This is the default setting when you use the `fdb` (Program Database) or `fprof` (Instrument) option.

`O2`

This level includes the `O` or `O1` optimizations described above, and the following additional optimizations:

`fcopyprop`, `fcondxform`, `fcse-follow-jumps`, `fcse-skip-blocks`, `fexpensive-optimizations`, `frerun-cse-after-loop`, `fschedule-insns`, `fschedule-insns2`, `fshadow-globals`, `fstrength-reduce`.

The `O2` level enables strength-reduction, combination of more than one variable value into a single register, copy propagation, tail-call elimination, leaf-procedure optimization, and instruction reordering (scheduling) to make use of the particular i960 processor's pipeline and superscalar capabilities.

`O3`

This level includes the `O2` optimizations described above, and the following additional optimizations:

`fcoerce`, `fconstprop`, `finline-functions`, `fshadow-mem`, `funroll-loops`.

`O4`

This level includes the `O3` optimizations described above, and the following additional optimizations:

`fcoalesce`, `fmarry_mem`, `fsplit_mem`.

`O5`

This setting specifies program-wide optimization. Before using the `O5` option, you should read "Program-Wide Analysis and Optimization", and "gcdm Decision Maker Option".

Note that the `O5` level is not accepted directly by the gcc960 driver. It is accepted only in the `subst` argument of the `gcdm` option.

o (Output)

Specifies output filename.

`o` *`filename`*

Specifies output filename.

P (Preprocessor Output)

Preprocessor output control.

Inhibits generation of #-lines with line-number information in the output from the preprocessor. This is useful when running the preprocessor on non-C/C++ code that is intended for a program that might be confused by the #-lines.

pedantic[-errors] (Pedantic)

Controls ANSI messages

`pedantic` causes the compilation system to issue all the warnings specified by ANSI C (such as when text other than a comment follows `#else` or `#endif`) and to reject programs that use forbidden extensions.

Valid ANSI standard C programs should compile properly with or without this option (though a rare few require `ansi`). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected.

`pedantic` does not cause warning messages for use of the alternate keywords whose names begin and end with _ _ (double underscore).

`pedantic-errors` is the same as `pedantic`, except that it causes the compilation system to issue errors instead of warnings.

S (Assembly)

Create assembly output.

Compile into assembly code but do not assemble. The assembly output filename is made by replacing `.c`, `.cc`, `.cpp`, `.cxx`,`.i`, `or .ii`, with`.S`, or `.s` at the end of the input filename. Do nothing for assembly source files or object files specified as input.

save-temps (Save Intermediates)

Save intermediate files.

Store the usual “temporary” intermediate files permanently; place them in the current directory and name them based on the source file. Thus, compiling `foo.c` with `-c -save-temps` would produce files `foo.i` and `foo.s`, as well as `foo.o`.

stdlibcpp

Link Standard C++ libraries

Instructs the compiler to link in the standard C++ libraries when creating an absolute module. The standard C++ libraries are included in the search path ahead of the standard C libraries. The distribution includes only an implementation of the C++ iostream classes. Future releases will add more modules. Note that this option has no effect if `nostdlib` is specified.

T (Target)

Specifies `.gld` *file.*

T*string*	where *string* identifies a target-specific configuration file, *string*.gld.

Causes gcc960 to configure itself for a specific target board.

traditional (Traditional)

Allow traditional C.

Attempt to support some aspects of traditional C compilers, specifically:

- All `extern` declarations take effect globally even when they are written inside of a function definition. This includes implicit declarations of functions.
- The keywords `typeof`, `inline`, `signed`, `const`, and `volatile` are not recognized.

- Integer types `unsigned short` and `unsigned char` promote to `unsigned int`.
- All automatic variables not declared `register` are preserved by `longjmp`. Ordinarily, GNU C follows ANSI C: automatic variables not declared `volatile` may be clobbered.
- In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.
- In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.
- The predefined macro `_ _STDC_ _` is not defined when you use `traditional`, but `_ _GNUC_ _` is (since the GNU extensions that `_ _GNUC_ _` indicates are not affected by `traditional`). When you need to write header files that work differently depending on whether `traditional` is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other C compilers.
- For C++ programs, `traditional` has the same effect as `-fthis-is-variable` as well as all the effects described above.

The following table summarizes the macros defined when the `traditional` or `ansi` option is used.

	_ _STRICT_ANSI_ _	**_ _STDC_ _**	**_ _GNUC_ _**
traditional			X
ansi	X	X	X
none		X	X

trigraphs (Trigraphs)

Support ANSI C trigraphs.

Process ANSI standard trigraph sequences. These are three-character sequences, all starting with `??`, that are defined by ANSI C to stand for single characters. For example, `??/` stands for `\`, so `'??/n'` is a character constant for a newline.

The `ansi` option also enables trigraphs.

U (Undefine)

Undefines a preprocessor macro.

`U`*`macro`*

Undefines the named preprocessor macro.

V (Version)

Display tool version numbers.

v (Verbose)

Display tool version numbers and subprocess commands.

v960 (Version, exit)

Display tool version numbers and exit.

W (Warnings)

Enables / disables specific warnings.

`W[`*`string`*`]`

`W` With no arguments, this option prints extra warning messages for certain events, including:

`longjmp() warnings`

Warn when a nonvolatile automatic variable might be changed by a call to `longjmp()`. These warnings are possible only in an optimizing compilation.

The compiler sees only the calls to `setjmp()`. It cannot know where `longjmp()` is called; in fact a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp()` cannot actually be called at the place that would cause a problem.

`return` **and** `return(`*`value`*`)`

Warn when a function can return either with or without a value. (Falling off the end of the function body is considered returning with a value.)

`W` (continued)

null effect

Warn when an expression-statement contains no side effects.

no-op comparison

Warn when an unsigned value is compared against zero with **`<`** or **`<=`**.

between-ness comparison

Warn when a comparison like `x<=y<=z is` used; this is equivalent to `{(x<=y ? 1 : 0) <=z}`, which is a different interpretation from that of ordinary mathematical notation.

obsolete storage class specification

Warn when storage-class specifiers like `static` are not first in a declaration. According to the ANSI C standard, this usage is obsolescent.

partially bracketed initializer

Warn when an aggregate has a partially bracketed initializer.

`Wall`

Enable the following warning options: `W`, `Wchar-subscripts`, `Wcomment`, `Wformat`, `Wreturn-type`, `Wswitch`, `Wtrigraphs`, `Wuninitialized`, `Wunused`. There is no `Wno-all` option.

`Waggregate-return`

Warn when any functions that return structures or unions are defined or called.

`Wcast-align`

Warn whenever a pointer is cast such that the required alignment of the target is increased. For example, warn when a `char *` is cast to an `int *` on machines where integers can be accessed only at two- or four-byte boundaries.

`Wcast-qual` Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn when a `const char *` is cast to an ordinary `char *`.

`Wchar-subscripts` Warn when an array subscript has type `char`. This is a common cause of error, as programmers often forget that this type is signed on some machines.

`Wcomment` Warn whenever a comment-start sequence `/*` appears in a comment.

`Wconversion` Warn when a prototype causes a type conversion different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument, except when these are the same as the default promotion.

`Werror` Make all warnings into errors.

`Wformat` Check calls to `printf` and `scanf`, etc., to make sure that the arguments supplied have types appropriate to the specified format string.

`Wid-clash-`*`len`* Warn whenever two distinct identifiers match in the first *`len`* characters. This may help you prepare a program that compiles with certain obsolete compilers. There is no `[no-]` form of this option.

`Wimplicit` Warn when a function is used without being explicitly declared.

`Wmissing-braces` Warn when an initializer is not completely enclosed within braces.

`Wmissing-prototypes`
Warn when a global function is defined without a previous prototype declaration. This warning is issued even when the definition itself provides a prototype. The aim is to detect global functions that are not declared in header files.

`Wnested-externs`
Warn when an `extern` declaration is encountered within a function.

`Woverloaded-virtual`
Warn when a derived class function declaration may be an error in defining a virtual function. In a derived class, the definitions of virtual functions must match the type signature of a virtual function declared in the base class. With this option, the compiler warns when you define a function with the same name as a virtual function, but with a type signature that does not match any declarations from the base class. `Wno-overloaded-virtual` is the default. This is a C++-specific option.

`Wparentheses`
Warn when parentheses are suggested around an expression.

`Wpointer-arith`
Warn about anything that depends on the size of a function type or of `void`. gcc960 assigns these types a size of 1, for convenience in calculations with `void*` pointers and pointers to functions.

`Wredundant-decls`
Warn when anything is declared more than once in the same scope, even in cases where multiple declaration is valid and changes nothing.

`Wreorder`
Warn when the order of member initializers given in the code does not match the order in which they must be executed. `Wno-reorder` is the default. This is a C++-specific option.

`Wreturn-type` Warn whenever a function is defined whose return-type defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.

`Wswitch` Warn whenever a `switch` statement has an enumeral type index and lacks a `case` for one or more of the named codes of that enumeration.

`Wshadow` Warn whenever a local variable shadows another local variable.

`Wstrict-prototypes` Warn when a function is declared or defined without specifying the argument types. An old-style function definition is permitted without a warning when it is preceded by a declaration specifying the argument types.

`Wtraditional`

- Warn about certain constructs that behave differently in traditional and ANSI C:
  Macro arguments occurring within string constants in the macro body. These would substitute the argument in traditional C, but are part of the constant in ANSI C.
- A function declared external in one block and then used after the end of the block.
- A switch statement has an operand of type long.

`Wtrigraphs` Warn when any trigraphs are encountered (assuming they are enabled).

`Wuninitialized`

An automatic variable is used without first being initialized. These warnings are possible only in an optimizing compilation, because they require data flow information that is computed only when optimizing. When no `O` option is given, these warnings are not generated.

These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4, or 8 bytes. Also, they do not occur for structures, unions, or arrays, even when they are in registers.

There may be no warning about a variable that is used only to compute a value that itself is never used, because such computations can be deleted by data flow analysis before the warnings are printed.

`Wuninitialized` (continued)

These warnings are optional because gcc960 cannot foresee all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
    int x;
    switch (y)
      {
      case 1: x = 1;
       break;
      case 2: x = 4;
       break;
      case 3: x = 5;
      }
    foo (x);
}
```

When the value of *y* is always 1, 2 or 3, then *x* is always initialized, but gcc960 doesn't know this. Here is another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y =
new_y;
  ...
  if (change_y) y = save_y;
}
```

This has no bug because `save_y` is used only when it is set.

Some spurious warnings can be avoided if you declare as volatile all the functions you use that never return.

`Wunused` Warn whenever a local variable is unused aside from its declaration, and whenever a function is declared `static` but never defined.

`Wwrite-strings`

Give string constants the type `const char[`*`length`*`]` so that copying the address of one into a non-`const char*` pointer generates a warning.

w (Inhibit Warnings)

Inhibits all warnings.

Z (Specify PDB)

Specifies PDB directory.

Z*directory*

Specifies the name of the program database (PDB) directory.

Before using this option, you should read "Program-Wide Analysis and Optimization", "Profile Data Merging and Data Format (gmpf960)", and , "gcdm Decision Maker Option".

ic960 Compiler Driver 3

This chapter describes how to use the ic960 driver program to control the compilation system. Topics include:

- "Controlling the Compilation System with ic960"
- "ic960 and File Use"
- "ic960 Options"
- "Option Arguments and Syntax"

Controlling the Compilation System with ic960

The ic960 compiler driver (`ic960.exe` in Windows, `ic960` on UNIX) controls the preprocessor (`cpp.exe` in Windows, `cpp.960` on UNIX) and the compiler (`cc1.exe` in Windows, `cc1.960` on UNIX). Starting with CTOOLS release 6.0 ic960 also controls the new C++ compiler (cc1plus.exe in Windows, cc1plus.960 on UNIX). It can also invoke the assembler, linker, and gcdm960 optimization decision maker. The command-line options and environment variables, described later in this chapter, allow you to control the compilation.

The drivers invoke the appropriate modules to compile a file based on filename extensions.

- Files with names ending with `.cc`, `.cpp`, and `.cxx` are taken as C++ source to be preprocessed and compiled. In UNIX, filenames ending with `.C` (uppercase) are treated as C++ source to be preprocessed and compiled.
- Files with names ending with `.ii` are taken as preprocessed C++ source to be compiled

- Files with names ending in `.c` are taken as C source to be preprocessed and compiled.
- Files with names ending in `.i` are taken as preprocessor output to be compiled.
- Compiler output files plus any input files with names ending in `.s` are assembled.
- Input files with names ending in `.S` (uppercase) are preprocessed and then assembled. (UNIX only.)
- The resulting object files, plus any other input files, are passed to the linker to produce an executable.
- Program-wide and profile-directed optimizations can be performed during the link step. For an overview of this capability, see Chapter 4, "Program-Wide Analysis and Optimization".

Invoking the Compiler with ic960

The ic960 command-line syntax is:

```
ic960 [-option]... [path]filename ...
```

`ic960` is the compiler driver executable filename.

`option` is a compiler option. Case is significant in options and their arguments.

On UNIX, the compiler driver recognizes a letter preceded by a hyphen (`-`) as an option. In Windows, the driver recognizes a letter preceded by either a hyphen (`-`) or a forward slash (`/`) as an option.

For a complete description of the ic960 options, see the ic960 Option Reference section. You can also use linker invocation options in an ic960 command; see Table 3-1 for a summary of these options.

`path` identifies the directory containing the file named by *`filename`*. Not specifying *`path`* for a *`filename`* causes ic960 to search in the current directory. Each *`filename`* not in the current directory requires a separate specification of *`path`*.

NOTE. *Although Windows pathnames require backslashes (`\`), this manual shows paths using the forward slash required by UNIX (`/`).*

filename — is the name of a source, assembly-language, or object file to be processed by the compilation system. The command line allows specification of more than one `[`*`path/`*`]`*`filename`*.

Table 3-1 lists the linker options that ic960 passes directly to the linker. To pass other options to the linker, use the `Wl,`*`arg`* pass-through option.

ic960 Sample Command Lines

This section provides examples of `how` the compiler is commonly invoked. All these examples assume that you have C source files named `t1.c` and `t2.c` or C++ source files name `t1.cc` and `t2.cc`. All examples assume that you are generating code for the i960 CA architecture.

Preprocessing a Source File

To preprocess a source file to stdout, use the command:

```
ic960 -E t1.c
```

or

```
ic960 -E t1.cc
```

`-E` — informs the compiler to preprocess the source file.

Generating a Preprocessed C++ Source File

To generate a preprocessed C/C++ source file use the following command. The command generates a preprocessed source file named `t1.i` (for C) or `t1.ii` (for C++).

```
ic960 -P t1.c
```

or

```
ic960 -P t1.cc
```

`-P` instructs the ic960 compiler to preprocess the file and store the output in `<basename>.i` for C or `<basename>.ii` for C++.

Generating Assembly Code

This example generates assembly code for the i960 CA architecture. The command lines below each generate an assembly language file named `t1.s`.

```
ic960 -S -ACA t1.c
```

or

```
ic960 -Felf -S -ACA t1.cc
```

`-Felf` specifies ELF object module format, which is required for C++. The default object module format is b.out.

`-S` instructs the compiler to generate assembly code.

`-ACA` specifies the i960 CA architecture.

Generating an Object Module with Debug Information

To generate a object module with debug information, use the following command.

```
ic960 -c -g -ACA t1.c
```

or

```
ic960 -Felf -c -g -ACA t1.cc
```

`-g` instructs the compiler to generate debug information.

`-c` instructs the compiler to generate an object file.

Generating an Executable

To generate an absolute module (executable file) for a Cyclone board with a CA processor, use the following command.

```
ic960 -ACA -Tcycx -g -O1 t1.c t2.c -o test
```

or

```
ic960 -Felf -ACA -Tcycx -g -O1 t1.cc t2.cc -o test
```

The above command compiles the source files and links them with appropriate libraries to generate an absolute module targeted for a Cyclone i960 Cx board.

`-Tcycx` use the linker directive file for a Cyclone i960 Cx evaluation board.

`-O1` causes the compiler to perform some basic optimizations on the generated code.

`-o test` instructs the compiler to name the generated executable `test`.

ic960 Linker Options

When you do not specify a target with the `T`*`target`* option, ic960 does not attempt to link programs for a specific target board. Unless otherwise specified, source files with recognized extensions (e.g., `.cc`, `.s`) are compiled and/or assembled, and the following linker command is issued:

```
lnk960 -AKB file.o... -lqf
```

ic960 links in the profiling library (`-lqf`) by default. To avoid linking in the profiling library, invoke lnk960 directly to perform your final link. You can also link in your own libraries (lib1, lib2...) if needed.

```
lnk960 -AKB file.o... -llib1 -llib2
```

You can invoke ic960 to create object files in either the COFF or ELF object module format. The compilation system accepts the `Fcoff` option to generate COFF and the `Felf` option to generate ELF. ELF is the only supported format for C++.

`Fcoff` is the default. For more detailed information, see the following discussions of compiler invocation and options.

Table 3-1 Linker Options Accepted by ic960 (Sheet 1 of 2)

Option	Name	Description
l	Archive file	specifies an archive file as input.
x	Compress	removes local symbols from the output symbol table.

Table 3-1 Linker Options Accepted by ic960 (Sheet 2 of 2)

Option	Name	Description
`L`	Library search	adds directories to search for libraries, configuration files, and startup object files.
`m`	Map	creates a linker memory map file.
`r`	Relocation	retains relocation information in the output object file.
`s`	Strip	strips line-number entries and symbol-table entries from the linker's COFF output file.
`T`	Target	specifies the file describing the target environment.
`u`	Undefine	introduces an unresolved symbol, causing the linker to search symbol tables for resolution of the reference.
`gcdm`	Decision Maker	invokes gcdm960 decision maker.

For more information on the linker, see the *i960 Processor Software Utilities User's Guide*.

ic960 and Predefined Macros

Predefined macros within a program can act as constants during execution or as values in conditional-compilation statements. Predefined macros include ANSI C macros and macros specific to the i960 processor architecture. The `U` (Undefine) option can remove i960 processor-specific macros but not ANSI C macros.

The following macros are available in accordance with the ANSI standard for C, as described in the book, *C: A Reference Manual*:

```
__DATE__    __FILE__    __LINE__    __TIME__    __STDC__
```

The following macros are predefined by the compilation system when invoked with the ic960 driver program:

`__IC960` indicates the CTOOLS960 compilation system. The compiler defines `__IC960` automatically, when invoked with the ic960 driver.

`__IC960_VER` is defined to a decimal number that can be used to check the version number of the compiler. The number is expressed in decimal as *`MmmPPPP`*, where *`M`* is the major version number, *`mm`* is the minor version number, and *`PPPP`* is an internal version number that is used to track the patch level. So, for example, R6.5 patch level 4008 has `__IC960_VER` defined to be 6054008.

`__i960` indicates the i960 processor environment. The compiler defines `__i960` automatically. This macro can be used to identify the parts of a program specific to the i960 processor.

`__i960`*`xx`* indicates the i960 processor instruction set in use. The compiler automatically defines the `__i960`*`xx`* macro. The *`xx`* is `SA`, `SB`, `KA`, `KB`, `CA`, `CF`, `JA`, `JD`, `JF`, `JT`, `HA`, `HD`, `HT`, `RD`, `RP`, `RM`, `RN`, or `VH`. Definition of *`xx`* depends on the specific i960 processor instruction set specified by the `A` (Architecture) option or the `I960ARCH` environment variable.

`__PIC` indicates that the generated code is position-independent. The `G pc` (Generate-for-position- independent-code) option causes the `__PIC` macro to be defined.

`__PID` indicates that the generated data is position-independent. The `G pd` **(**Generate-for-position- independent-data) option causes the `__PID` macro to be defined.

`__i960_ABI__` indicates that the generated code is 80960 ABI-Conformant. The `Gabi` option causes this macro to be defined.

`__i960_BIG_ENDIAN` indicates that the generated code is arranged for big-endian address space. The `G be` (Generate-big endian) option causes this macro to be defined.

`__STRICT_ANSI__`
`__STRICT_ANSI` indicates that C constructs not conforming to the ANSI standard should be flagged. The `a` (ANSI) option causes these macros to be defined.

`__SIGNED_CHARS__` indicates that the plain `char` type are treated like the `signed char` type. This is the default.

`__CHAR_UNSIGNED__` indicates that the plain `char` type are treated like the `unsigned char` type. The `G cu` (Generate-char-unsigned) option causes this macro to be defined instead of `__SIGNED_CHARS__`.

ic960 and Environment Variables

Environment variables specify default directories for input files, temporary files, libraries, the assembler, and the linker. In addition, the `I960ARCH` environment variable specifies the default architecture. The compilation system uses the following environment variables to set defaults:

`I960ARCH` specifies an architecture other than the i960 KB processor for code generation. The possible definitions for `I960ARCH` are `CA`, `CF`, `HA`, `HD`, `HT`, `KA`, `KB`, `RD`, `RP`, `SA`, `SB`, `JA`, `JD`, `JF`, `JT`, `RM`, `RN`, or `VH`. The `A` (Architecture) option overrides the architecture specified in `I960ARCH`. In the absence of `I960ARCH` and the Architecture option, the compiler selects the i960 KB processor architecture.

`I960BASE` contains the pathname of the top-level directory containing the files and directories needed by the compiler. This environment variable is necessary for every phase of compilation. The driver uses `I960BASE` to find the preprocessor, compiler, assembler, linker, and include files.

To invoke the preprocessor and compiler, the ic960 driver looks in the `lib` directory under `I960BASE`.

To invoke the assembler and linker, the driver looks in the `bin` directory under the directory specified by `I960BASE`.

To find include files, the driver looks in the `include` directory under the directory specified by `I960BASE`.

The linker looks for libraries, startup modules, and configuration files in the `lib` directory under the directory specified by `I960BASE`.

`I960AS`
specifies a non-default pathname for the assembler. The pathname must include the name of the executable. In the absence of `I960AS`, ic960 looks for the assembler in `bin` under the directory specified by `I960BASE`.

`I960CC1PLUS`
Specifies an alternate name for the C++ compiler when using the ic960 driver. The default pathname is `I960BASE/lib/cc1plus.960` (`I960BASE\lib\cc1plus.exe` in Windows).

`I960CPP`
specifies an alternate name for the preprocessor. The default pathname is `I960BASE/lib/cpp.960` (`I960BASE\lib\cpp.exe` in Windows).

`I960CC1`
specifies an alternate name for the compiler. The default pathname is `I960BASE/lib/cc1.960` (`I960BASE\lib\cc1.exe` in Windows).

`I960DM`
specifies an alternate name for the gcdm960 optimization decision maker.

`I960ERR`
The assembler, linker, and other tools can redirect errors to the standard error stream (`stderr`). To use this capability, set the Windows environment variable `I960ERR` to any string, as in:

```
SET I960ERR="Enable stderr"
```

Leaving `I960ERR` unset directs error output to the standard output stream (`stdout`).

`I960INC` specifies a non-default pathname for the directory containing include files. In the absence of `I960INC`, the driver looks for include files in the `include` directory in the directory specified under `I960BASE`.

`I960LIB`, `I960LLIB` contain additional pathnames of libraries. Definition of `I960LIB` causes the linker to search for libraries in the directory specified by `I960LIB`. In the absence of `I960LIB`, the linker searches the `lib` directory in the directory specified by `I960BASE`. Definition of `I960LLIB` causes the linker to search the directory specified by `I960LLIB` before searching the `lib` directory in the directory specified by `I960BASE`. For a complete description of the search algorithm used by the linker, see the *i960 Processor Software Utilities User's Guide*.

`I960LD` contains an alternate pathname of the linker. The path must include the name of the executable. In the absence of `I960LD`, ic960 looks for the linker in the `bin` directory under the directory specified by `I960BASE`.

`I960PDB` defines the location of the program database for use with profile-driven optimizations. The `Yd` (Program Database) option overrides this environment variable and allows specification of an alternate database directory.

`TEMP, TMP, TMPDIR, G960TMP` contain the pathname of the directory used for compiler temporary work files. In the absence of these variables, the compiler attempts to write temporary work files to the current working directory in Windows, and to `/tmp` or `/usr/tmp` on UNIX.

ic960 and File Use

The compiler, assembler, and linker all use filenames specified on the ic960 command line to find and create input and output files. In addition, translation and linking require temporary work files. Environment variables allow specification of default directories for work files.

Input Files

The ic960 command line allows filename inputs that support specification of assembly-language files, preprocessed source files, C/C++ source files, object files, and libraries. The compiler driver determines the type of each input file by the filename extension, as follows:

`*filename*.c` indicates a C source file that can contain macros and preprocessor directives.

filename`.cc, .cpp,`indicates a C++ source file that can contain

`.cxx` macros and preprocessor directives.

filename`.C` indicates a C++ source file that can contain macros and preprocessor directives (UNIX only).

filename`.i` indicates a preprocessed C source file.

filename`.ii` indicates a preprocessed C++ source file.

filename`.s` indicates an assembly-language source file.

The driver passes any other filename to the linker. The linker then determines whether the file is an object file, library, or configuration file.

Input files not needed for processing are not processed. For example, if you specify an assembly-language (*filename*`.s`) file and also specify the `S` (Save assembly) stop-after option, ic960 takes no action on the assembly-language file because processing stops after compilation and before assembly.

Include Files

The ic960 command line allows insertion of text from include files. Both the `i` (Preinclude) option and the `#include` preprocessor directive cause text insertion.

The `#include` preprocessor directive causes a search of the directory or directories indicated by the `I` (Searchinclude) option. In the absence of the `I` option, ic960 searches the current directory, the directory defined by the `I960INC` environment variable, or the `I960BASE/include` directory.

NOTE. *The include files `icache.h`, `dcache.h`, and `timer.h` used for on-chip cache and timer control are not supported with the `-ARP` option.*

Temporary Files

The compiler, assembler, and linker automatically create and delete temporary work files. You need not remove temporary work files unless your host system loses power or some other abnormal termination prevents the compilation system from cleaning up its work files.

The compiler selects a directory for temporary work files as follows:

`G960TMP`, `TEMP`, `TMPDIR`, `TMP`, `.\` (Windows), `/tmp` (UNIX), `/usr/tmp` (UNIX).

Output Files

Specifying a Stop-after option (`-n`, `-Q`, `-E`, `-P`, `-S`, or `-c`) causes the compilation system to produce a separate output file representing the output of the last phase that completes for each primary input file. An output file can be a preprocessed source file, an assembly-language file, a listing file, a map file, or an unlinked object file. If no errors occur during processing, the output files created by the stop-after option are usable as input to a future ic960 invocation. Table 3-2 lists the compilation phases and their inputs and outputs.

Specifying the `Z` (Listname) option allows specification of a list file filename; ic960 places all listings in the single file specified. If you do not use `Z`, ic960 produces a separate list file for each primary C/C++ source file. Each filename has the form *`file`*`.L`, where *`file`* is the same name as the C/C++ source file.

Table 3-2 Intermediate Inputs and Outputs

Last Phase Completed	Stop-after Option	Inputs	Outputs
preprocessing	P, E, or Q	C/C++ source files	preprocessed files or display on standard output
syntax checking	n	C/C++ source files preprocessed files	syntax error list listing files
compilation	S	C/C++ source files preprocessed files	assembly-language file listing files
assembly	c	C/C++ source files preprocessed files assembly files	unlinked object files listing files
linking	(default)	C/C++ source files preprocessed files assembly files unlinked object files relinkable object files libraries configuration files	list files executable file map file relinkable object file

When specifying only one primary input file, the `o` (Output) option names a single output file besides the listing file. Specifying multiple primary input files, or not specifying an output filename, causes ic960 to use the primary input filenames to derive corresponding default output filenames with the form *`filename.e`*, where:

`filename` is the primary input filename without its extension.

`e` is a single-letter extension indicating the contents of a file, as follows:

`i` indicates a preprocessed C source file from the `P` (Preprocess-files) stop-after option.
`ii` indicates a preprocessed C++ source file from the `P` (Preprocess-files) stop-after option.
`s` indicates an assembly-language file

from the `S` (Save assembly) stop-after option.

`o` indicates an object file from the `c` (Create-object) stop-after option.

`L` indicates a listing file from the `z` (List) option.

Unless otherwise specified, the destination directory for any output file is the current working directory. If *`filename.e`* already exists in the destination directory, the compilation system overwrites the existing file.

The filename `a.out` is the default for the executable COFF object file from the linker, produced in the absence of the stop-after options and the Output option. For ELF files, the default is `e.out`.

Creating a linker configuration file containing information for preparing an absolutely relocated module, a module for incremental linking, or code ready for programming into read-only memory (ROM) allows for additional file types. For more information on linker configuration, see the *i960 Processor Software Utilities User's Guide*.

The following examples illustrate the creation and use of output filename extensions:

- The command `ic960 -c -zs proto.c proto1.i` produces the object files `proto.o` and `proto1.o` and the listing files `proto.L` and `proto1.L`.
- The command `ic960 -c -o proto_v1.o -zs  proto.c` produces the object file `proto.o` and the listing file `proto.L`.
- The command `ic960 -ACA -Tcycx proto.c` produces the executable file `a.out`.

ic960 Options

This section describes the ic960 compiler driver options that allow control of various aspects of compilation:

Input processing and output

The `c`, `E`, `n`, `P`, `Q`, and `S` are the Stop-after options.

They stop the translation and linking process after the preprocessing, syntax checking, compilation, or assembly phase. A Stop-after option causes the compilation system to save the intermediate output of the last phase to execute.

The `C` (Keep-comments) and `M` (Mix) options affect the contents of the output file. The `o` (Output) option allows specification of the output filename.

Specifying included source text

The `i` (Preinclude) and `I` (Searchinclude) options prepend and find include files of C/C++ source text.

Defining macros

The `D` (Define) and `U` (Undefine) options allow specification of macros for conditional compilation.

Control contents of generated object code

The `A` (Architecture), `Fcoff/Felf` Object-format), `F` (Fine-tune), `f` (Optimize), `g` (Debug), `G` (Generate), and `O` (Optimization-level) options control the instruction set, object format, debug information, and optimization level.

Assembler and linker support

The `W` (Pass) option relays options to the preprocessor, compiler, assembler, and linker. In addition, ic960 recognizes some options as linker options rather than compiler options. Table 3-1 lists the options that are relayed to the linker without the Pass option. For more detailed information on linker options, see the *i960 Processor Software Utilities User's Guide*.

Whole-program optimizations	The `fdb` (Program Database), `fprof` (Instrumentation), and `gcdm` (Decision Maker) options allow for creation and use of information necessary for advanced optimizations involving multiple modules and optional execution profiles. See Chapter 4, "Program-Wide Analysis and Optimization" for an overview of whole-program and profile-driven optimization.
Provide Information on the compiling process	The `w` (Diagnostic) and `a` (ANSI) options affect messages the compiler produces about C syntax and semantics. The `z` (List) and `Z` (Listname) options specify the contents and name of the listing file. The `v` (Verbose), `V` (Version), and `v960` (Version-exit) options display information about preprocessor, compiler, assembler, and linker options. The Version option displays the versions of each compilation component and the host operating system. The `W` (Warnings) option allows fine control of the level of warnings emitted.

Option Arguments and Syntax

Some compiler driver options take arguments. Whitespace is optional between an option and its argument. Case is significant in options and arguments.

The options and arguments have default settings. In most cases, the option is "off," that is, not in effect. Default settings of options and arguments are summarized in Table 3-3 and further discussed in the detailed description of the option. Some option defaults are affected by environment variables, as noted in the option descriptions.

This chapter uses the following notation:

[*item*] Square brackets indicate that the enclosed item is optional.

`. . .` Horizontal ellipses indicate that you can use multiple instances of the preceding item.

If two or more options contradict each other, the right-most option in the command line takes precedence. For example, the following command line sets the value of the symbol `L` to `132`:

```
ic960 -DL=80 -DL=132 proto.c
```

Table 3-3 ic960 Option Summary (Sheet 1 of 2)

Option	Name	Purpose	Default
A *arch*	Architecture	Select the instruction set.	AKB
a	ANSI	Warn about non-ANSI source.	Do not warn
b *size*	Limit-optimizations	Limit optimization of functions with more than *size* asm instructions.	b 2500
C	Keep-comments	Keep comments in preprocessor output.	Strip comments
c	Create-object	Stop after creation of object file.	Do not stop
D *symbol* [=*value*]	Define	Define *symbol*.	*symbol*=1
E	Preprocess - stdout	Write preprocessed source to stdout; terminate.	Do not stop
Fcoff | Felf	Object-format	Generate COFF or ELF object format.	Fcoff
fdb	Database	Build program database (PDB).	No database
fprof	Instrument	Compile with instrumentation; build PDB.	No instrument-ation
F [no]*arg*	Fine-tune	Adjust optimizations.	
f [no]*arg*	Additional fine-tune	Enable or disable an optimization.	
G *arg* [,*arg*]...	Generate	Control code generation options.	G cs,dc
g [*level*]	Debug	Include debug information in objects.	No debug info
gcdm	Decision-maker	Invoke gcdm960 decision-maker.	Do not invoke gcdm960
h	Help	Display invocation help; terminate.	No help text
I *dir*	Searchinclude	Search *dir* for include files.	
i *filename*	Preinclude	Prepend text to source files.	

Table 3-3 ic960 Option Summary (Sheet 2 of 2)

Option	Name	Purpose	Default
J *arg* [,*arg*]...	Miscellaneous	Selects miscellaneous controls.	J nogd
j *num*	Errata	Specify processor errata.	
M	Mix	Mix C/C++ text with assembly output.	No C text
n	Syntax only	Check syntax; list errors; terminate.	Do not stop
O *level*	Optimize	Specify optimization level (0, 1, 2, or 5).	O1
o *filename*	Output	Name output file.	*filename*=a.out
P	Preprocess - file	Write preprocessed source text to files; terminate.	Do not stop
Q	Dependencies	Print include-file dependencies; terminate.	No print
S	Save-assembly	Save assembly-language output.	Do not save
U *symbol*	Undefine	Undefine symbol.	
V	Version	Display version information.	No display
v960	Version-exit	Display version information and exit.	
v	Verbose	Display invocation information.	No display
W *phase* *arg* [,*arg*]...	Pass	Pass arguments to preprocessor, compiler, assembler, or linker.	
W [no-]*arg*	Warnings	Enable/disable a warning.	
w *level*	Diagnostic-level	Control diagnostic messages.	*level*=1
Y d,*dirname*	Program database	Specify location of program database.	I960PDB specifies location
Z *filename*	Listname	Name listing file.	Compiler generates name
z *arg*	List	Produce listing file.	No listing

A (Architecture)

Selects the instruction set.

`A`*`architecture`*

`architecture` is one of:

`CA`, `CF`, `KA`, `KB`, `RD`, `RP`, `SA`, `SB`, `HA`, `HD`, `HT`, `JA`, `JD`, `JF`, `JT`, `RM`, `RN`, or `VH`.

Default

By default, the compiler uses the i960 KB architecture. The `I960ARCH` environment variable can override the default architecture.

Discussion

Use the `A` (Architecture) option to specify the target instruction set. This option overrides the environment variable `I960ARCH`. See also the `-Gcore0`, `-Gcore1`, `-Gcore2`, and `-Gcore3` options that let you generate code that is compatible with multiple i960 processor types.

NOTE. *Starting with release 6.0, using the `-ARP` or `-ARD` option generates code that is compatible with current and proposed future variations on the i960 Rx architecture.*

You can use predefined macros in your source text to conditionally compile code for the selected architecture. The compiler defines a preprocessor macro indicating the selected architecture. The preprocessor macro takes the form:

`__i960`*`xx`*

`xx` is `CA`, `CF`, `KA`, `KB`, `RD`, `RP`, `SA`, `SB`, `HA`, `HD`, `HT`, `JA`, `JD`, `JF`, `JT`, `RM`, `RN`, or `VH`. The compiler selects the value of *`xx`* according to the architecture you specify.

In addition to `__i960xx`, the `__i960` macro is defined for all architecture selections. Use `__i960` to identify parts of your program specific to the i960 architecture but not necessarily specific to a particular processor.

In addition, for compatibility with earlier releases, macros of the forms: `i960`, `_ _i960_ _`, `_ _i960xx_ _` and `_ _i960_xx_ _` are defined.

If you link object modules compiled with incompatible architectures, the linker displays the following warning message:

```
file: architecture i960:XX incompatible with output
i960:YY
```

file is the first file containing incompatible instructions the linker encounters.

XX is one of the two-letter architecture abbreviations.

YY is one of the two-letter architecture abbreviations.

Example

The following example selects the i960 KA instruction set:

```
ic960 -AKA proto.c
```

a (ANSI)

Flags non-standard constructs.

```
a
```

Default

The compiler accepts constructs that are legal under Kernighan and Ritchie's definition of the C language but that do not comply with the ANSI standard.

Discussion

Use the ANSI option to flag old-style C constructs that are legal according to Kernighan and Ritchie's definition in *The C Programming Language*, but are not legal according to the ANSI standard. When the ANSI option is in effect, the compiler prints warning messages for each occurrence. This is a C-specific option.

NOTE. *When this option is in effect, if your program contains in-line assembly-language (*`asm`*) statements, the compiler treats the statement as a regular function call and produces code for the call. For example, if your program contains the following line:*

```
asm("flushreg");
```

The compiler produces the following code:

```
   callj  _asm
LFC0.$:
   asciz "flushreg"
...
```

The linker may then generate an error for an undefined extern for the `_asm` *call.*

To use `asm` *statements and functions with the* `a` *option, use the* `__asm` *keyword.*

Specifying the `a` (ANSI) option can override the `w` (Diagnostic-level) option, as follows:

`-a -w2` has the same effect as `-a -w1`; that is, errors and major warnings appear.

`-a -w1` errors and major warnings appear.

`-a -w0` errors and all warnings appear.

Example

The following example causes the compiler to issue an error message when it encounters a non-standard C construct. Because of the `c` (Create-object) option, the compiler stops after creating an object file:

```
ic960 -c -a proto.c
```

Related Topic

`W` (Warnings)   `w` (Diagnostic-level)

b (Limit-optimizations)

Limits optimizations.

b*size*

size is a positive decimal integer.

Default

Having more than 2500 intermediate language statements in a function causes the compiler to disable some global optimizations.

Discussion

As function size increases, the compiler slows. The `b` (Limit-optimizations) option allows you to alter the threshold at which optimizations are scaled back when functions are too large to compile quickly.

Example

In the following example, the `b` (Limit-optimizations) option forces suppression of global optimization for functions in `proto.c` larger than 2000 intermediate language statements.

```
ic960 -b2000 -S proto.c
```

Related Topic

`O` (Optimize)

C (Keep-comments)

Keeps comments in preprocessor output.

```
-E -C
-P -C
```

Default

All comments are stripped away.

Discussion

Use the `C` (Keep-comments) option to preserve comments normally stripped by the preprocessor. This option modifies the `E` and `P` Stop-after options. Using the `C` (Keep-comments) option alone neither generates a preprocessor listing nor stops the processing after the preprocessor phase.

Example

The following example uses the `C` (Keep-comments) option to modify the `P` (Preprocess - file) option. The output is a newly created file named `proto.i`, containing the comments as they appear in the original C source text.

```
ic960 -P -C proto.c
```

Related Topics

`E` (Preprocess - stdout)`P` (Preprocess - file)

c (Create-object)

Create object file; terminate.

```
c
```

Default

Create an executable file after the link phase of the compilation process.

Discussion

If you specify `c` (Create-object) the compilation process terminates after the assembler generates an object file. If you do not specify the `o` (Output) option, the compiler writes the object file to *`filename`*`.o`, where *`filename`* is the source filename.

Examples

1. The following example produces the object file `proto.o` but no executable file:

   ```
   ic960 -c proto.c
   ```

2. The following example produces the object files `proto.o`, `t1.o`, and `proto1.o` in the current directory but creates no executable file:

   ```
   ic960 -c proto.c t1.s proto1.i
   ```

Related Topics

`o` (Output)	Stop-after options

D (Define)

Define a symbol.

`D` *`symbol`*`[=`*`value`*`]`

`symbol`	is a symbolic name.
`value`	is a value. The value can be any string.

Default

If you define *`symbol`* without specifying *`value`*, the preprocessor assigns the value 1 to *`symbol`*.

Discussion

Use the `D` (Define) option to create a symbol with a given *`value`*. You can use the `D` (Define) option more than once in an invocation.

You can use the `D` (Define) option with conditional compilation to create macros to select source text during preprocessing. A macro defined in the invocation command remains in effect for each module compiled, unless you remove the macro with the `#undef` preprocessor directive or the `U` (Undefine) option. The compilation system processes all the `U` (Undefine) options in a command-line only after processing all the `D` (Define) options.

Example

The following example invokes the preprocessor with `D LONGPATH`, so that `PATHLENGTH` is defined with the value 128 in the source file. Since the macro `LONGPATH` is defined without a value, it defaults to 1:

```
ic960 -c -D LONGPATH proto.c
```

The source text is:

```
#ifdef LONGPATH
#define PATHLENGTH 128
#else
#define PATHLENGTH 45
#endif
```

Related Topics

`#define`
`#undef`

`U (Undefine)`

E (Preprocess - stdout)

Preprocess; write output to screen; terminate.

```
E
```

Default

After the link phase of the compilation process is complete, an executable file is produced.

Discussion

If you specify E, the compilation process terminates after preprocessing and the compiler writes preprocessor output with line number directives to standard output.

Example

The following example runs only the preprocessor phase, sending the preprocessed source text to the screen:

```
ic960 -E proto.c
```

Related Topic

Stop-after options

Fcoff | Felf (Format)

Specifies object format.

`Fcoff`	specifies the COFF object format, and causes the assembler to be invoked as asm960. You can add the g option to specify the style of symbolic-debug symbols created.
`Felf`	specifies the ELF object format, and causes the assembler to be invoked as gas960e, rather than asm960. If you add the g option, the DWARF style of symbolic-debug symbols is used. ELF is the only supported format for C++.

NOTE. *Unlike gcc960, ic960 does not support the b.out object module format.*

F (Fine-tune)

Adjust optimizations.

`F` *`arg`*`[,`*`arg`*`]...`

`arg` is any of:

`F[no]ai` enables/disables procedure in-lining using heuristics at optimization level 2.

`F[no]ca` enables/disables code alignment; generate (do not generate) alignment directives prior to labels that are not entered from above.

`F[no]cb` enables/disables use of compare and branch instructions.

`F[no]lp` enables/disable code generation of functions using the `bal` calling sequence at optimization level 1 or 2. `nolp` is the default at optimization level 1, and `lp` is the default at optimization level 2.

`F[no]pf` This option is obsolete. It is recognized but has no effect.

`F[no]sa` determines whether or not the compiler risks generating memory references that are not provably aligned. If `Fnosa` is selected, the compiler occasionally generates potentially unaligned references when it seems advantageous to do so. When `Fsa` is enabled, sequences of smaller memory references are used instead of larger ones that might not be correctly aligned.

`sb | nosb` enables/disables superblock formation. Suppressing this optimization may reduce your application's code size.

`tce | notce`	enables/disables conversion of tail calls into branch instructions at optimization level 1 or 2. `notce` is the default at optimization level 1, and `tce` is the default at optimization level 2.

Default

The set of optimizations performed is determined by the argument of the `O` (Optimize) option.

Discussion

Use the `F` (Fine-tune) option to fine-tune how your code is optimized. For general purposes, the optimization level specified with the `O` (Optimize) option is sufficient. The optimizations performed at each level balance considerations of code quality, ease of debugging, and compilation time. However, circumstances can call for use of, or disabling of, some specific optimizations.

Example

To disable heuristic function in-lining and leaf procedure generation when compiling at optimization level 2, enter the following:

```
ic960 -F noai,nolp -O2 proto.c
```

fdb (Database)

Builds optimization database.

All modules subject to program-wide optimization must be initially compiled with the `fdb` option. This option causes the insertion of program database information in the object modules, and it implies a minimum module-local optimization level of `O1` (although higher module-local optimization levels are allowed).

This option does not otherwise change the code or data generated for the object modules in any way. It simply makes information collected during initial module compilation available to the global decision maker (gcdm). Before using the `fdb` option, you should read Chapter 4, "Program-Wide Analysis and Optimization", and Chapter 6, "gcdm Decision Maker Option".

If you intend to use execution profiles when optimizing your application, you should read Chapter 5, "Profile Data Merging and Data Format (gmpf960)".

fprof (Instrument)

Instruments code for profile creation.

This compiler driver option inserts execution profile instrumentation code into the generated code during compilation, so that when the linked program is executed, a profile can be collected.

This option implies the `fdb` option (described previously) that causes the insertion of program database information in the object modules and the creation of the program database. Since `fprof` implies `fdb`, `fprof` also implies a minimum module-local optimization level of O1 (although high module-local optimization levels are allowed).

When you compile with the `fprof` option, a special profiling library required for profile collection (`libqf`) is linked automatically. If your target environment does not support file I/O, you must explicitly link an alternate profiling library (`libq`). The profiling libraries provided are identified in Chapter 2 of the *i960 Processor Library Supplement*.

Note that when the `fprof` option is used in this manner, the generated object module contains code is unsuitable for linking into programs that are not supposed to collect profile information. To solve this problem, and avoid having inappropriate instrumentation in widely-used library modules for example, use `+fprof` with the `gcdm,subst` option instead.

Before using the `fprof` option, you should read Chapter 4, “Program-Wide Analysis and Optimization”, Chapter 5, “Profile Data Merging and Data Format (gmpf960)”, and Chapter 6, “gcdm Decision Maker Option”.

f (Additional Fine-tune)

Additional optimization adjustments.

`f [no-]`*`arg`*

`arg` is any one of the optimizations listed below. This option takes only one argument; use a separate `f` option to enable/disable an optimization.

The `f [no-]`*`arg`* option is supported to allow access to optimization controls that are supported by the gcc960 compiler driver.

Note that most of these options are controlled automatically by the various `O` optimization levels. Therefore, some of them may be ignored for certain compilations. For example, at optimization level `O0`, you cannot enable instruction scheduling with `fschedule-insns`. To check whether one of these options has the desired effect, compare the generated assembly code with and without the option.

`[no-]access-control` Enable all access checking. This is normally used to work around access control bugs. `Faccess-control` is the default. This is C++ specific option.

`bbr` Enable basic block rearrangement.

`coalesce` Coalesce adjacent memory references into a single reference of a larger size, to take advantage of the processor’s burst bus. Only memory references that can be proven to be contiguous and whose base address can be proven to be aligned properly are coalesced. This option implies `fshadow-mem`.

`coerce` Enable byte/short optimization.

`cond-mismatch` Allow type mismatch in operands of the `?:` operator.

`condxform` Enable 80960 conditional instructions.

`[no-]conserve-space` Allocate uninitialized global variables into the common segment, as C does. This saves space in the executable at the cost of not diagnosing duplicate definitions. `Fno-conserve-space` is the default. This is a C++ specific option.

`constprop` Enable constant propagation and folding.

`copyprop` Enable copy propagation.

`cse-follow-jumps` Enable a limited form of global CSE.

`cse-skip-blocks` Enable a limited form of global CSE.

`[no]dollars-in-identifiers` Accept “$” in identifiers. ANSI C and C++ forbid “$” in identifiers. `Fno-dollars-in-identifiers` is the default when `ansi` is specified.

`[no-]enum-int-equiv` Allow implicit conversion of integer to enumeration types. Normally the compiler allows conversion of enum to int, but not vice versa. `Fno-enum-int-equiv` is the default. This is a C++ specific option.

`expensive-optimizations` Enable some minor optimizations.

`float-store` Do not store floating-point variables in registers, and do not perform common sub-expression elimination on floating-point expressions.

`[no-]for-scope`
Limit the scope of variables declared in a `for-init` statement to the for loop itself, as specified by the draft C++ standard. When you specify `-fno-for-scope`, the scope of variables declared in a for-init-statement extends to the end of the enclosing scope, as was the case in old versions of gcc960, and other (traditional) implementations of C++. `ffor-scope` is the default. This is a C++ specific option

`force-addr`
Place address constants in registers before use.

`[no-]huge-objects`
The implementation of virtual function calls assumes that the size of an object can be represented with a short integer. Use this flag to support virtual function calls for objects that exceed the size that can be represented by a short integer. Use this flag only if the compiler requests you to do so. Note that the C++ library sources need to be recompiled with `Fhuge-objects` if you plan to link with the C++ libraries. `Fno-huge-objects` is the default.

`fint-alias-ptr`
Indicates to the compiler that pointer objects may be referenced as 32-bit integers and vice versa.

`fint-alias-real`
Indicates to the compiler that `float`, `double`, and `long double` objects (or parts thereof) may be referenced as 32-bit integers and vice versa.

`fint-alias-short`

Indicates to the compiler that four-byte integer objects may be referenced as two-byte integer objects and vice versa.

The aliasing options listed above tell the compiler not to use certain kinds of type information when disambiguating memory references, even though it could do so according to ANSI C section 3.3 (disambiguation constraints).

The rules enforced by the aliasing options are transitive. For example, if user code accesses parts of `double` objects as `short`, then `fint-alias-real` and `fint-alias-short` should both be used.

The rules are also applied recursively to `structs` and unions. That is to say, if `fint-alias-ptr` is in use, then a union that has a member of pointer type is assumed to be aliased by 32-bit integers or by structures or unions containing

Note that ANSI C 3.3 requires the compiler to assume that `char` references alias all types, so code using `char` pointers for this sort of thing is already correct and using these options is not necessary.

Using all three aliasing options effectively disallows all use of type information in memory disambiguation. This is bad both for compiler performance and the efficiency of generated code.

`keep-inline-functions`

Emit out-of-line code for inlined functions.

`marry_mem`

Rejoin multi-word moves split by `fsplit_mem`.

`F[no-]memoize-lookups`
`F[no-]save-memoized` — Use heuristics to compile faster. These heuristics are not enabled by default, since they are only effective for certain input files. Other input files compile more slowly. You may use either option to compile using heuristics. These are C++ specific options.

`rerun-cse-after-loop` — Reiterate CSE after loop optimization.

`sblock` — Enable/disable superblock formation. This option is normally used in a second-pass recompilation, but it can also be used in a single-pass compilation.

`schedule-insns` — Perform pre-register-allocation scheduling.

`schedule-insns2` — Perform post-register-allocation scheduling.

`shadow-globals` — Shadow memory locations in registers.

`shadow-mem` — Like `shadow-globals`, but more thorough.

`space-opt` — Optimize for code size.

`split_mem` — Split multi-word moves for copy propagation.

`strength-reduce` — Enable loop strength reduction.

`F[no-]strict-prototype` — Treat a function declaration with no arguments, such as "`int foo ();`", to mean that the function `foo` takes no arguments. `Fstrict-prototype` is the default. This is a C++ specific option.

`[no-]this-is-variable` — Permit assignment to "this". `Fno-this-is-variable` is the default. This is a C++ specific option.

`thread-jumps` — Enable an advanced branch optimization.

`unroll-all-loops` — Unroll all loops.

`unroll-loops` — Unroll loops where deemed beneficial.

`virtual-opt`	Optimizes the dispatch of virtual functions. This optimization can be used only in a 2-pass scheme. By default, this optimization is not enabled. This optimization can be used only when certain conditions are met. See "Optimizing Virtual Function Dispatch" in Chapter 12 for more details. This is a C++-specific option.
`volatile`	Treat indirect memory references as volatile.
`volatile-global`	Treat all memory references as volatile.
`writable-strings`	Place string literals in `.data` section.

Default

The set of optimizations performed is determined by the argument of the `O` (Optimize) option.

G (Generate)

Select code generation options.

`G` *`arg`*`[,`*`arg`*`]...`

`arg` is one of the following:

`abi`	Generate 80960 ABI-conformant code. This causes the `char` type to be signed, enums to be four bytes in size and signed, and changes default alignment rules for structs and unions. See Chapter 7, "C Language Implementation"for more information.
`ac=`*`n`*	Aligns `struct` data types on the byte boundary specified by *`n`*. *`n`* can be 1, 2, 4, 8, or 16.

`bc` Generates code that is backwardly-compatible with releases of ic960 before Release 3.0.

`be` Generates objects that execute in a big-endian memory environment.

`cave` Generate all functions as CAVE secondary functions.

`core0|core1|` generate code that is compatible with multiple

`core2|core3 |` i960 processor types. Additionally, when you use a `-Gcore` option, you can include another `-A` switch to generate code that is optimized for a particular architecture, but still compatible with a group of architectures. The table below lists the architectures that are supported by a `-Gcore` option and the `-A` options that you can use with them.

Table 3-4 Gcore Supported Architectures

Option Name	Compatible Architectures	Can Be Used With
`Gcore0`	Jx, Hx, Rx	`-AJA`, `-AJD`, `-AJF`, `-AJT`, `-AHA`, `-AHD`, `-AHT`, `-ARD`, `-ARP`, `-ARM`, `-ARN`, or `-AVH*`.
`Gcore1`	Kx, Sx, Cx, Jx, Hx	Any architecture option except `-ARP` `-ARD`, `-ARM`, or`-ARN`.
`Gcore2`	Jx, Hx	`-AJA`, `-AJD`, `-AJF`, `-AJT`, `-AHA`, `-AHD`, `-AHT`, or `-AVH*`.
`Gcore3`	Cx, Jx, Hx	`-ACA`, `-ACF`, `-AJA`, `-AJD`, `-AJF`, `-JT`, `-AHA`, `-AHD`, `-AHT`, or `-AVH*`.

*Note that the big-endian mode is not supported for VH.

`cs` or `cu` Treats `char` data types as `signed` or `unsigned`, respectively. `cs` is the default.

`dc` Specifies the relaxed ref-def external linkage model. This is the default.

`ds` Specifies the strict ref-def external linkage model.

`pc` Generates position-independent code.

`pd` Generates position-independent data.

`pr` Reserves register `g12` containing the position-independent data (PID) bias value.

`wait=`*`n`* Specifies wait-state for memory accesses. *`n`* is in the range 0 through 32, inclusive.

`xc` Specifies that all external calls in the module use the extended-call mechanism.

Discussion

You can select multiple arguments either by specifying all of them, separated by commas, as the argument of a single `G` (Generate) option, or by specifying each as the argument of a separate `G` (Generate) option. If you specify conflicting arguments, the last one takes precedence.

Alignment Argument (ac): If you select `ac=`*`n`*, the compiler aligns `struct` data types on *`n`*-byte boundaries. This is equivalent to an initial `#pragma align(`*`n`*`)` and does not override any subsequent `#pragma align(`*`n`*`)` directives. Alignment values can only be `1`, `2`, `4`, `8`, or `16`. Chapter 7, "Position Independence and Reentrancy" describes alignment in more detail.

Backward-compatible Argument (bc): If you select `bc`, the compiler generates object modules that can be linked with object modules translated by ic960 Release 2.0. This option resolves the following compatibility issues:

- The default alignment of individual `struct` data types for ic960 Release 2.0 can differ from the default structure alignment for Release 3.0 and later releases. The Release 3.0 ic960 derives the default alignment of a `struct` data type from its size, by rounding up from the size to the next power of 2 (to a maximum of 16). In code translated by ic960 releases before 3.0, the alignment of the `struct` defaults to the alignment of the largest member of the `struct`. You must compile all modules of a program with the same alignment.

- For enum data types, the compiler selects a basic integral representation type, choosing the narrowest type capable of representing all of the enumeration values. The compiler can represent the `enum` type as `signed char`, `unsigned char`, `short`, `unsigned short`, or `int`, depending upon the range of enumeration values. Before Release 3.0, the compiler used only signed types to represent `enum` data types. For example, a maximum enumeration value between 128 and 255 inclusive, now represented as an `unsigned char`, was represented as a `short` in Release 2.0.
- The values of upper, unused bits of prototyped parameters and return values smaller than 32 bits for ic960 Release 2.0 can differ from the corresponding bit values for Releases 3.0 and later. The calling convention for Release 3.0 does not extend the unused bits. The called function must extend into the unused bits of prototyped parameters and the function using a return value must extend into unused bits of the return value. In code translated by ic960 releases that preceded 3.0, the calling conventions extend into unused bits when passing prototyped parameters and returning values smaller than 32 bits.
- With this release of the compiler, the recipient of a narrow integral value must assume that the high-order bits of the register containing the value do not contain the appropriate zero- or sign-extension of the value passed. It is the recipient function's responsibility to clean the upper bits of a parameter or return value if necessary. Using the Backward Compatible (`bc`) argument causes the compiler to use the rules of prior releases. Before this release of the compiler, narrow integral values were always sign- or zero-extended by the originator.
- The Release 2.0 compiler, when used to compile for an i960 KB or SB processor, returns `long double` (80-bit) floating-point numbers in the `fp0` floating-point register.
- The Release 3.0 compiler, when used to compile for any i960 processor, returns `long double` floating-point numbers in the `g0`, `g1`, and `g2` global registers. When Release 3.0 is used to compile for a processor without a floating-point unit (e.g., the KA, SA, CA, or CF processor), the compiler generates calls to the accelerated floating-point library ("libh"). (Release 2.0 generated calls to the

FPAL floating-point-arithmetic library, but FPAL is no longer supported.) You must recompile any KA, SA, CA, or CF module that was compiled with ic960 R2.0 floating-point operations.

Big-endian Argument (be): If you select `be`, you inform the compiler that the memory system of the entire program is in big-endian format. Only the i960 Cx, Hx, and Jx processors support big- and little-endian format. Do not use this argument with other i960 architectures.

The compiler automatically passes the `G` (Generate big-endian) option to the assembler or linker if they are to be run.

Compression Assisted Virtual Execution (CAVE): If you select `cave`, the compiler generates special CAVE entries for all functions in the compilation unit. This prepares the functions for link-time compression. The CAVE entries resemble the following:

```
.section .text
_foo:
     lda     L1,reg
     call    __dispatcher
     ret
.section cave
     .word  L2-L1,0
L1:
     function body
L2:
```

At runtime, the dispatcher decompresses the function bodies and transfers control to them. This mechanism saves runtime memory. (See the discussion of `#pragma cave` in Chapter 7, "C Language Implementation" for more information.)

Signed and Unsigned Character Arguments (cs and cu): If you select `cs`, declarations of `char` are treated as `signed char`. (This is the default.)

If you select `cu`, declarations of `char` are treated as `unsigned char`.

Relaxed and Strict Linkage Definition Arguments (`dc` and `ds`): In the default relaxed ref-def external linkage model (i.e., the `dc` argument), any variable declared with the `extern` keyword is a reference to a variable and does not define storage. Somewhere in all the modules, a definition at file-scope must exist. You can have multiple definitions. All definitions are combined into a single storage location by the linker. Storage is allocated for initialized variables in the `.data` section with the appropriate initializer. Uninitialized definitions are allocated to the common sections using the `.comm` assembly language directive. At link time one of the following happens:

- If a variable is defined with an initializer in one module, and without an initializer in all other modules, the linker allocates space for the object in the `.data` section.
- If no definitions of a variable are initialized, all common references are combined and allocated to the `.bss` section. With the relaxed ref-def model, you cannot relocate uninitialized variables to named sections at specific memory locations using the linker configuration language.

In the strict ref-def model (i.e., using the `ds` argument), only one definition is allowed and all others must be declared with the keyword `extern`. You cannot have more than one definition of an object with external linkage. Storage is allocated to uninitialized file-scope variables in the `.bss` section. Initialized variables are allocated in the `.data` section with the appropriate initializer. Using the strict ref-def model, you can relocate uninitialized variables to named sections at specific memory locations using the linker configuration language. For more detailed information about using the linker, see the *i960 Processor Software Utilities User's Guide*.

Position Independence Arguments (pc, pd, and pr): If you select `pc`, the compiler generates position-independent code and predefines the `__PIC` macro.

NOTE. *Applications built using the* `pc` *option cannot be linked with assembly sources that contain* `callx` *or* `balx` *instructions, since these instructions are not position-independent.*

If you select `pd`, the compiler generates position-independent data and predefines the `__PID` macro. Register `g12` contains the bias value for the data sections; its contents cannot be modified, even during the saving or restoring process.

If you select pr, the compiler reserves register g12. Use this option for position-dependent modules to be combined with position-independent data modules. See Chapter 10, "Position Independence and Reentrancy" for more information on this subject.

Extended Call Argument (xc): Use the Extended Call argument when your code calls external functions outside the range of the call or bal opcodes. When you use this argument, the compiler emits the calljx pseudo-opcode, which the linker translates to either of the MEM format opcodes callx or balx. The linker decides which translation to perform based on the symbol table entry for the defined function. The extended call opcodes can address the entire 2^{32} address range. The extended call instructions occupy two words of code space. The single word CTRL format call instructions occupy one word.

The compiler emits the CTRL format callj pseudo-opcode when calling any function defined outside the current compilation module.

Examples

1. The following example aligns structures on 8-byte boundaries:

   ```
   ic960 -Gac=8 proto.c
   ```

2. The following example generates a module that can be linked with code resulting from an ic960 Release 2.0 translation:

   ```
   ic960 -Gbc proto.c
   ```

3. The following example generates code in which variables declared as char are treated as unsigned char:

   ```
   ic960 -Gcu proto.c
   ```

4. The following example generates position-independent code and data:

   ```
   ic960 -Gpc,pd proto.c
   ```

Related Topics

A (Architecture)	__PIC	#pragma align
I960ARCH	__PID	#pragma i960_align
__i960*xx*		

g (Debug)

Include debug information in object module.

```
g [level]
```

where *level* specifies the amount of debug information. Note that the meaning of level varies depending on the object format in use, as described below.

Using g0 disables debug information. (This is the same as not using the g option.)

For COFF, debug level settings of g, g1, g2, and g3 all have the same effect: they specify "normal" debug information.

For ELF/DWARF, debug level settings of g, g1, and g2 all have the same effect: they specify all DWARF debug information except preprocessor macros. A debug level setting of g3 specifies all DWARF debug information, including preprocessor macros in the debug information. If your debugger (like gdb960) does not make use of preprocessor macro information, you can save space in your object files by dropping to ELF/DWARF debug level 2.

The g (Debug) option does not inhibit optimization. If you specify the g option but do not specify the O (Optimize) option, the optimization level defaults to O0.

Specifying an optimization level higher than O0 can inhibit the effectiveness of the symbolic debug information. For example, if you set a breakpoint on a source line for which the code has been optimized away, the breakpoint is never hit. Or if you try to print the value of a variable that has been optimized away, an erroneous value is displayed. In general, as the optimization level increases, the reliability of the symbolic debug information decreases.

If you are using the ELF object module format (`Felf`), then `g` causes the compiler to produce DWARF debug information. This debug information format is richer than that of other supported OMFs, and allows more reliable debugging under optimization. However, even with DWARF, there are situations where debugging behavior does not agree with the debugging behavior of unoptimized code.

gcdm (Decision Maker)

Invoke gcdm960 decision-maker.

```
gcdm,arg[,arg]...
```

The `gcdm` option provides a high level of automation for the whole-program or profile-driven optimization process. The compiler driver and the linker both use the `gcdm` option and its arguments.

The `gcdm` option is flexible and powerful, and therefore requires a certain level of understanding in order to use it effectively. For these reasons, it is documented in a separate chapter (Chapter 6, "gcdm Decision Maker Option").

Before using the `gcdm` option, you should read Chapter 4, "Program-Wide Analysis and Optimization", and become familiar with the information in Chapter 5, "Profile Data Merging and Data Format (gmpf960)".

h (Help)

Display invocation help; terminate.

```
h
```

Discussion

This option causes the compiler to display brief descriptions of each option on the standard output device and then terminate.

I (Searchinclude)

Search alternate #include directory.

```
I dir
```

`dir` is a directory containing files to be included.

Default

If you use `#include "filename"` to specify a filename that is not an absolute pathname, the compiler searches directories in the following order:

1. the directory containing the primary C/C++ source file (the primary directory).
2. if `I960INC` is defined, the directory specified by `I960INC.`
3. if `I960INC` is not defined, the `include` directory located under the directory specified by `I960BASE.`

For a *filename* included with `#include <filename>`, the compiler searches directories in the following order:

1. if `I960INC` is defined, the directory specified by `I960INC`.
2. if `I960INC` is not defined, the `include` directory located under the directory specified by `I960BASE`.

Discussion

Use `I` (Searchinclude) to specify additional directories for the preprocessor to search to find files specified with `#include`. The preprocessor searches Searchinclude directories before the directory specified by `I960INC` or `I960BASE`. If you use quotation marks (`#include "filename"`), the preprocessor searches the primary directory first. If you use angle brackets (`#include <filename>`), the preprocessor does not search the primary directory.

Examples

1. In the following example, the preprocessor searches:
   - — `/usr/home/src` (the directory containing `proto.c`)
   - — `/usr/home/include` (the Searchinclude directory)
   - — `/usr/home/testinclude` (the directory specified by `I960INC`)

   The environment variable definitions are:
   - — `I960BASE` is set to `/usr/local/i960`
   - — `I960INC` is set to `/usr/home/testinclude`

   The command-line is:
   - — `ic960 -I /usr/home/include /usr/home/src/proto.c`

   The source text contains:
   - — `#include "proto.h"`
2. In the following example, the preprocessor searches:
   - — `/usr/home/include` (the Searchinclude directory)
   - — `/usr/local/i960` (the directory specified by `I960BASE`)

   The `I960BASE` environment variable is set to `/usr/local/i960`

   The command-line is:
   - — `ic960 -I /usr/home/include /usr/home/src/proto.c`

   The source text contains:
   - — `#include <proto.h>`

If the preprocessor does not find `proto.h`, for either of these examples, the compiler displays the following error message:

```
ic960 ERROR:  "/usr/home/src/proto.c", line 1 --
proto.h: No such file or directory
```

Related Topics

`#include`
`I960BASE`
`I960INC`
`i` (Preinclude)
Stop-after options

i (Preinclude)

Prepend text file to primary source files.

`i` ***`filename`***

`filename` is the name of a C/C++ source text file.

Discussion

Use the `i` (Preinclude) option to prepend the text of a C/C++ source file or include file to each C/C++ source file specified on the command line. This option has the same effect as placing an `#include` directive at line zero of each C/C++ source file.

The compiler searches for ***`filename`*** in the same way as for a file specified with `#include` using quotation marks. For a description of include file searching rules, see the `I` (Searchinclude) option description. The compiler issues an error if the file is not found.

Example

The following example prepends the file `globals.h` to the file `proto.c`:

```
ic960 -i globals.h proto.c
```

Related Topics

`#include`	`I960INC`	Stop-after options
`I960BASE`	`I` (Searchinclude)	

J (Miscellaneous)

Selects miscellaneous controls.

```
J arg[,arg]...
```

Discussion

Use the `J` (Miscellaneous) option to specify miscellaneous controls. Two such controls are `gd` (issue gcc960-style diagnostics) and `nogd` (issue ic960-style diagnostics). gcc960-style diagnostics are more compact, and do not include column position indicators.

Default

`nogd` (issue ic960-style diagnostics).

j (Errata)

Specifies processor errata.

j *num* Discussion

Use the `j` (Errata) option to cause the compilation system to generate code with workarounds for specified processor errata. A *`num`* argument of 1 generates code to work around the Cx processors' DMA errata.

M (Mix)

Mixes C/C++ source text with assembly language output.

```
-S -M
```

Default

Assembly language output does not contain interleaved C/C++ source as comments.

Discussion

Use the `M` (Mix) option to modify the `S` (Save-assembly) option to put C/C++ source text as comments into the assembly language output file. Using the `M` (Mix) option without the `S` (Save-assembly) option has no effect.

Note that if you use the `O` (Optimize) option with the `M` (Mix) option, the C/C++ source text comments can be mismatched to the assembly language text, since optimization can reorder and eliminate assembly language instructions.

Example

The following example produces the assembly language file `proto.s` containing C source text as comments:

```
ic960 -S -M proto.c
```

Related Topics

`O` (Optimize) `S` (Save-assembly)

n (Check-syntax)

Check syntax; terminate.

```
n
```

Default

After the link phase of the compilation process is complete, an executable file is produced.

Discussion

If you specify `n` (Check Syntax Only) the compilation process terminates after performing syntax and semantic checking. The compiler generates diagnostic messages but produces no output.

Example

The following example runs a syntax check only on the file `proto.c`, generating no output file:

```
ic960 -n proto.c
```

O (Optimize)

Optimize.

```
O[level]
```

The `O[`*`level`*`]` option specifies the level of optimization as described below.

`O0` Disables optimizations, including those that may interfere with debugging. This is the optimization level if you use the `g` (Debug) option.

`O1` Enables basic optimizations, including: advanced register allocation, common subexpression elimination, loop invariant code motion, expression simplification and instruction combination, jump optimization, dead-code elimination, and i960 processor-specific peephole optimization. This is the default setting if you do not use the `g` (Debug) option or when you use the `fdb` (Program Database) or `fprof` (Instrument) options.

`O2` This level includes the `O1` optimizations described above, tail-call elimination, leaf-procedure optimization, and the following optimizations:

`fcoalesce`, `fcoerce`, `fcondxform`, `fconstprop`, `fcopyprop`, `fcse-follow-jumps`, `fcse-skip-blocks`, `fexpensive-optimizations`, `finline-functions`, `fmarry_mem`, `frerun-cse-after-loop`, `fschedule-insns`, `fschedule-insns2`, `fshadow-globals`, `fshadow-mem`, `fsplit_mem`, `fstrength-reduce`, `funroll-loops`.

`O5` This setting specifies program-wide optimization. Before using the `O5` option, you should read Chapter 4, “Program-Wide Analysis and Optimization”, and Chapter 6, “gcdm Decision Maker Option”.

Note that the `O5` level is not accepted directly by the ic960 driver. It is accepted only in the `subst` argument of the `gcdm` option.

o (Output)

Name output file.

```
o filename
```

`filename` is the name of the file to receive the final output of the compilation.

Default

If the linker is to be invoked, the default name of the linker's output is `a.out` for COFF and `e.out` for ELF. Otherwise, each output filename is determined by replacing the filename extension of each input file. Output filenames' extensions depend on the Stop-after option in effect, as follows:

- `P` (Preprocess-file): *`filename`*`.i (C)` *`filename`*`.ii (C++)`
- `S` (Save-assembly): *`filename`*`.s`
- `c` (Create-object): *`filename`*`.o`

Discussion

Use the `o` (Output) option to direct the final output of a compiler invocation to a specific file. The final output can be any of the following:

For `E`, `Q`, and `n`, the output goes to `stdout`.

- If you specify the `P` (Preprocess - file) option, the final output is the result of preprocessing.
- If you specify the `S` (Save-assembly) option, the final output is the assembly language text generated by the compiler.
- If you specify the `c` (Create-object) option, the final output is the object module generated by the assembler.
- Otherwise, the final output is the result of linking.

The compiler issues an error message if you use the `o` (Output) option and do not invoke the linker when processing more than one input file.

Related Topic

Stop-after options

P (Preprocess-file)

Preprocess; write output to file; terminate.

```
P
```

Default

After the link phase of the compilation process is complete, the compilation system produces an executable file.

Discussion

If you specify the `P`, (Preprocess-file) option, the compilation process terminates after preprocessing and the compiler writes preprocessor output without line number directives to a file. If you do not specify a filename with the `o` (Output) option, the file is ***`filename`***`.i` (for C) or ***`filename`***`.ii` (for C++), where ***`filename`*** is the source filename without its extension.

Example

The following example puts the preprocessed source for `proto.c` in the file `proto.i` and the preprocessed source for `proto1.c` in the file `proto1.i`:

```
ic960 -P proto.c proto1.c
```

Related Topics

`o` (Output)

Stop-after options

Q (Dependencies)

Print include-file dependencies; terminate.

```
Q
```

Discussion

If you specify `Q` (Dependencies), the compilation process terminates after preprocessing and the compiler writes a list of dependency lines to standard output. The dependency lines take the form *`object`*`:` *`subfile`* where *`object`* is an object filename derived from the name of a primary C/C++ source file and *`subfile`* is the name of a file needed to create the object file. The preprocessor generates one line for each *`subfile`* on which the object file depends, including the primary C/C++ source file. Preprocessor directives for conditional compilation affect the output of the dependency lines.

Example

The following example generates a file dependency list for `dtest.c`. File `dtest.c` includes files `dinc.h`, `d2.h`, and `d3.h`, as follows:

```
#include "dinc.h"
#include "d2.h"
#include "d3.h"
```

File `dinc.h` includes file `dad.h`, as follows:

```
#include "dad.h"
```

The files `d2.h` and `d3.h` do not include any files. The following command compiles `dtest.c` with `Q`, resulting in file dependency lines:

```
ic960 -Q dtest.c
dtest.o: dtest.c
dtest.o: dinc.h
dtest.o: dad.h
dtest.o: d2.h
dtest.o: d3.h
```

Related Topics

`#include`	`o` (Output)	Stop-after options

S (Save-assembly)

Compile; save assembly language output; terminate.

`S`

Default

After the link phase of the compilation process is complete, the compiler produces an executable COFF file. (Assembly language output is not saved.)

Discussion

If you specify `S` (Save-assembly), the compilation process terminates after the compiler generates assembly code and writes the output to a file. If you do not specify a filename with the `o` (Output) option, the compiler writes the assembly language output to `*filename*.s`, where `*filename*` is the source filename without its extension.

Use the `M` (Mix) option to create a mixture of assembly language source code and corresponding C/C++ source code.

Examples

1. The following example creates the assembly language output from `proto.c` into `proto.s`:

   ```
   ic960 -S proto.c
   ```

2. The following example creates `proto.s`, the assembly language file for `proto.c`, and `t1.s`, the assembly language file for `t1.c`, in the current directory:

   ```
   ic960 -S proto.c -t1.c
   ```

Related Topics

`M` (Mix)`o` (Output)

Stop-after Options (n | Q | P | E | S | c)

Stop after the specified compilation phase.

```
n | Q | P | E | S | c
```

Default

After the link phase of the compilation process is complete, the compilation system produces an executable file.

You can use the `o` (Output) option to specify a name for the executable file. The default output filename is `a.out` (COFF) or `e.out` (ELF).

Discussion

Use one of the Stop-after options to halt the compilation process before linking and to write the intermediate output to a file or standard output. You can also use the `o` (Output) option to specify a filename for the output file.

Table 3-3 summarizes the processing and output other than listing the files that result from each Stop-after option.

If you specify `n` (Syntax-checking), the compilation process terminates after syntax and semantic checking are performed. The compiler generates diagnostic messages but produces no output.

If you specify `Q` (Dependencies), the compilation process terminates after preprocessing and the compiler writes a list of dependency lines to standard output. The dependency lines take the form *`object`*`:` *`subfile`* where *`object`* is an object filename derived from the name of a primary C/C++ source file and *`subfile`* is the name of a file needed to create the object file. The preprocessor generates one line for each *`subfile`* on which the object file depends, including the primary C/C++ source file. Preprocessor directives for conditional compilation affect the output of the dependency lines.

Table 3-5 Stop-after Option Phases and Output

Name	Option	Processing Phases	Output
Syntax-check	n	preprocessing, syntax-checking	a list of diagnostic messages, written to standard error
Dependencies	Q	preprocessing	a list of file-dependence lines, written to standard output
Preprocess - stdout	E	preprocessing	preprocessed source text with line number directives, written to standard output
Preprocess - file	P	preprocessing	preprocessed source text without line number directives, written to files
Save-assembly	S	preprocessing, compilation	assembly language, written to files
Create-object	c	preprocessing, compilation, and assembly	object modules, written to files

If you specify `E` (Preprocess - stdout), the compilation process terminates after preprocessing and the compiler writes preprocessor output with line number directives to standard output. The `o` (Output) option does not affect output from `E`.

If you specify `P` (Preprocess - file) the compilation process terminates after preprocessing and the compiler writes preprocessor output without line number directives to a file. If you do not specify a filename with the `o`

(Output) option, the compiler writes preprocessor output to `filename.i` (for C) or `filename.ii` (for C++), where *`filename`* is the source filename without its extension.

If you specify `S` (Save-assembly), the compilation process terminates after the compiler generates assembly code and writes the output to a file. If you do not specify a filename with the `o` (Output) option, the compiler writes the assembly language output to *`filename`*`.s`, where *`filename`* is the source filename without its extension. If you also specify the `M` (Mix) option, the assembly language output file also contains interleaved C/C++ source lines.

If you specify `c` (Create-object), the compilation process terminates after the assembler generates an object file. If you do not specify the Output option, the compiler writes the object file to *`filename`*`.o`, where *`filename`* is the source filename without its extension.

Examples

1. The following example puts the preprocessed source for `proto.c` in the file `proto.i` and the preprocessed source for `proto1.c` in the file `proto1.i`:

   ```
   ic960 -P proto.c proto1.c
   ```

2. The following example runs only the preprocessor phase, sending the preprocessed source text to the screen:

   ```
   ic960 -E proto.c
   ```

3. The following example runs a syntax check only on the file `proto.c`, generating no output file:

   ```
   ic960 -n proto.c
   ```

4. The following example puts the assembly language output from `proto.c` into `proto.s`:

   ```
   ic960 -S proto.c
   ```

5. The following example puts `proto.s`, the assembly language file for `proto.c`, and `t1.s`, the assembly language file for `t1.c`, in the current directory:

   ```
   ic960 -S proto.c -t1.c
   ```

6. The following example produces the object file `proto.o` but no executable file:

   ```
   ic960 -c proto.c
   ```

7. The following example produces the object files `proto.o`, `t1.o`, and `proto1.o` in the current directory but creates no executable file:

   ```
   ic960 -c proto.c t1.s proto1.i
   ```

8. The following example lists file dependencies for `dtest.c`:

   The `dtest.c` file includes the `dinc.h`, `d2.h`, and `d3.h` files, as follows:

   ```
   #include "dinc.h"
   #include "d2.h"
   #include "d3.h"
   ```

   The `dinc.h` file includes the `dad.h` file, as `#include "dad.h"`.

   The `d2.h` and `d3.h` files do not include any files. The following command compiles `dtest.c` with `-Q`, resulting in the following lines:

   ```
   ic960 -Q dtest.c
   dtest.o: dtest.c
   dtest.o: dinc.h
   dtest.o: dad.h
   dtest.o: d2.h
   dtest.o: d3.h
   ```

Related Topics

`C` (Keep-comments)	`o` (Output)	`z` (List)
`M` (Mix)	`V` (Verbose)	

U (Undefine)

Undefine symbol.

`U` *`symbol`*

`symbol`	is a symbolic name.

Default

No symbols are undefined.

Discussion

Use the `U` (Undefine) option to remove preprocessor macro symbols. Examples of symbols you can undefine include:

- the `__IC960`, `__i960` and `__i960`*`xx`* macros, where *`xx`* is `CA`, `CF`, `KA`, `KB`, `SA`, `SB`, `JA`, `JD`, `JF`, `RM`, `RN`, or `VH`.
- the `__PIC` and `__PID` macros
- symbols you have defined on the command line
- the symbol for big-endian code generation, `__i960_BIG_ENDIAN`

The compiler processes all the `U` (Undefine) options in a command line only after processing all the `D` (Define) options.

You cannot undefine or redefine the following predefined ANSI C macros:

`__DATE__` is the calendar date of the translation.

`__FILE__` is the name of the current source file.

`__LINE__` is the line number of the current source program line.

`__TIME__` is the calendar time of the translation.

`__STDC__` indicates that the compiler conforms to ANSI C.

Example

The following examples both undefine the symbol `__i960KA`:

```
ic960 -AKA -U__i960KA proto.c
ic960 -AKA -U__i960KA -D__i960KA=2 proto.c
```

Related Topics

A (Architecture)
D (Define)
#define
__i960*xx*
__i960
__PIC
__PID
#undef

V (Version)

Display version information.

```
V
```

Default

The compiler does not display version information.

Discussion

Use the V (Version) option to display to standard error the name and version, as shown below.

```
ic960 Version x.y.nnnn
```

x.y identifies the major release of the compiler

nnnn identifies the product's patch level

Version information differs for each host system and for each release.

Related Topic

v (Verbose)

v (Verbose)

Display invocation information.

```
v
```

Default

The compilation system does not display individual phase invocation information.

Discussion

Use the `v` (Verbose) option to display the standard errors from invocations of the driver program, preprocessor, compiler, assembler, and linker. These invocations are command lines generated by the driver program from the files and `W` (Pass) options you specify in the `ic960` command.

For example, if you specify the `v` (Verbose) option, the driver program passes it to the linker, even if you do not specifically use the `W` (Pass) option. The linker displays on standard output the files linked according to the following categories:

- input object files
- startup file
- high-level libraries
- low-level libraries

Example

The following command-line requests verbose invocation information:

```
ic960 -v -T cycx -ACA -o hello.out hello.c
```

Related Topics

`I960AS`	`I960LD`	`W` (Pass)
`I960BASE`	Stop-after options	`V` (Version)

v960 (Version, terminate)

Display version information and terminate.

```
v960
```

Default

The compilation system does not display version information.

Discussion

Use the `v960` (Version, terminate) option to display version information. This is the only thing the driver program does before terminating.

W (Pass)

Pass arguments to phases.

`W` ***`phase`***`,`***`arg`***`[,`***`arg`***`]. . .`

`phase` is a letter identifying the phase to receive the arguments, as follows:

`a` indicates the assembler.

`c` indicates the compiler.

`l` indicates the linker.

`p` indicates the preprocessor.

`arg` is a string to be passed to and interpreted by the phase. Each ***`arg`*** is passed as a separate argument. If an ***`arg`*** string contains whitespace, you must enclose the string in quotation marks.

Discussion

Use the `W` (Pass) option to specify options for the preprocessor, compiler, assembler, or linker. The driver program does not interpret the argument strings; only the receiving phase interprets them.

Related Topic

Stop-after options

W (Warnings)

Enable or disable a warning.

`W [no-]`*`arg`*

The `W [no-]`*`arg`* option allows more fine-grained control over diagnostics than `w` *`level`*.

`arg` is any of:

`aggregate-return`	warn if any functions return structures or unions.
`all`	enable several useful warnings. Has no `Wno-all` form.
`cast-align`	warn if a pointer cast may not have the required alignment.
`cast-qual`	warn if a pointer cast removes a type qualifier.
`char-subscripts`	warn if an array variable has type `char`.
`comment`	warn whenever `/*` occurs in a comment.
`conversion`	warn if a prototyped parameter causes a different conversion from the conversion that would take place if the parameter were not prototyped.
`error`	treat all warnings as errors.
`format`	check arguments of `printf`-family arguments at compile time.
`id-clash-`*`n`*	warn if two identifiers match in the first *`n`* characters.
`implicit`	warn if a function is used before it is declared.
`missing-braces`	warn if an aggregate initializer is not fully enclosed in braces.

`missing-prototypes` warn if a function is defined before it is prototyped.

`nested-externs` warn if an `extern` declaration is detected inside a function.

`overloaded-virtual` Warn when a derived class function declaration may be an error in defining a virtual function. In a derived class, the definitions of virtual functions must match the type signature of a virtual function declared in the base class. With this option, the compiler warns when you define a function with the same name as a virtual function, but with a type signature that does not match any declarations from the base class.

`Wno-overloaded-virtual` is the default. This is a C++-specific option.

`parentheses` warn if parentheses are suggested around an expression.

`pointer-arith` warn if the size a function type or type `void` is used.

`redundant-decls` warn if an object is declared twice in the same scope.

`reorder` Warn when the order of member initializers given in the code does not match the order in which they must be executed. `Wno-reorder` is the default. This is a C++-specific option.

`return-type` warn if any function implicitly returns `int`, and if any non-void function does not return a value.

`shadow` warn if a local variable shadows another local variable.

`strict-prototypes` warn if a function is declared without a prototype.

`switch` warn if a switch statement on an enumeration type does not have a case for each enumerator.

`traditional`	warn about contructs that behave differently in traditional C and ANSI C.
`trigraphs`	warn if any trigraphs are detected.
`uninitialized`	warn if use of an uninitialized local variable is detected.
`unused`	warn about objects that are never used.
`write-strings`	warn if string constants are used in a writable context.

w (Diagnostic-level)

Controls listing or display of diagnostic messages.

`w` *`level`*

`level`	is the level of diagnostic messages to be listed or displayed; can be `0`, `1`, or `2`.

Default

The compiler displays error and major warning messages; that is, *`level`* is `1`.

Discussion

Use the `w` (Diagnostic-level) option to suppress the warning messages that highlight legal but questionable uses of C. Unlike errors, uses of C that result in warning messages do not prevent the compiler from completing the translation and linking process.

To choose the level of diagnostic messages, use one of the following for the *`level`* argument:

`0`	to enable all warning and error messages
`1`	to enable major warning and error messages, suppressing only minor warning messages

`2`	to enable only error messages, suppressing warning messages

The `a` (ANSI) option always overrides the `w2` option, forcing the compiler to list or display warning messages.

The `W` (Warnings) option can be used to enable/disable specific warnings that would otherwise fall under the control of the `w` (Diagnostic-level) option. This is a C++-specific option.

Example

The following example displays warning and error diagnostic messages:

```
ic960 -c -w1 proto.c
```

Related Topics

`a` (ANSI)   Stop-after options   `W` (Warnings)

Yd (Program database)

Specifies location of program database.

`Yd,`*`PDB_directory`*

`PDB_directory`	specifies the directory containing the program database (PDB).

Default

The environment variable `I960PDB` specifies the location of the program database.

Discussion

When linking an instrumented program to generate profile information, during the Decision-making step, and during Profile-driven Recompilation, the location of the program database (PDB) must be specified. You can use the `Yd` (Program database) option to override `I960PDB` or to indicate where the PDB is located if `I960PDB` is not defined.

The PDB is a directory that the compilation system uses to store various files that it generates to contain information about the profile-driven compilation of a program. It must be specified either via the `Yd,`*`PDB_directory`* option, or with the `I960PDB` environment variable.

Z (Listname)

Names listing file.

`Z` *`filename`*

`filename` is the name of the listing file to be created.

Default

The compiler generates listing filenames from the primary source filenames.

Discussion

Use the `Z` (Listname) option to name the listing file. If you specify more than one source file on the command line, the compiler concatenates the listings for all the source text files into the single *`filename`* listing file. Using the `Z` (Listname) option without the `z` (List) option generates a listing file containing only primary source text.

Example

The following example produces the listing file `list.t` containing a source text listing for the file `proto.c`:

```
ic960 -c -Z list.t proto.c
```

Related Topics

Stop-after options`z` (List)

z (List)

Produce listing file.

`z` *`arg...`*

`arg`	is one of the following:	
	`s`	lists the primary source text, that is, source text from files named on the command line.
	`i`	adds included source text to the primary source text listing.
	`o`	adds the assembly language generated by the compiler to the listing file.
	`m`	adds expanded preprocessor lines to the primary source text listing.
	`c`	adds conditionally noncompiled source text to the primary source text listing.

Default

The compiler does not produce any listing files.

Discussion

Use the `z` (List) option to generate a listing file for each primary source file and to specify the listing file contents. The *`arg`* applies to all listing files produced. A listing file contains, at a minimum, the source text from the primary source file and diagnostic messages according to the diagnostic level. You can add other listing information by specifying one or more *`arg`* arguments instead of or in addition to `s`. Using the `i`, `o`, `m`, or `c` argument implies the `s` argument.

Unless you specifically name the listing filename with the `Z` (Listname) option, the compiler derives a listing filename from each primary source filename, as follows:

`base``.L`

base is a primary source filename, without its extension.

Example

The following example produces the listing file `complex.L` and object file `complex.o` in the current working directory for the source file `complex.c`. The listing file contains primary source listing, included source text, assembly language, source text that is conditionally compiled out, and expanded macros.

```
ic960 -c -z cosmi /complex.c
Include  Line
 Level  Number  Source Lines
======= ======  ============
          # Command line  (ic960): ic960 -c -z cosmi complex.c
         # Command line  (cc1): /ffs/p1/dev/src/gcc960/timc.sun4/cc1
.960 -ic960 -ffancy-errors -sinfo /usr/tmp/ica29412.sin -fno-builtin
-quiet -Fcoff -mkb -mic3.0-compat -fsigned-char -w1 -O1
-fno-inline-functions
-clist siomc -dcmd "ic960 -c -z cosmi complex.c" -dumpbase complex
-outz complex.L -tmpz /usr/tmp/ica29412.ltm /usr/tmp/ica29412.i -o
/usr/tmp/ica29412.s
          .file   "complex.c"
           gcc2_compiled.:
           ___gnu_compiled_c:
    0*      1   #include "complex.h"
    1*      1
    1       2   /* Define a struct for complex numbers
    1       3      with some macros */
    1       4
    1       5   #if !defined(complex_h)
    1       6
    1       7   struct complex {
    1       8       double x;
    1       9       double i;
    1      10   };
    1      11
    1      12   #define INIT_COMPLEX(num, real, imag) \
    1      13     num.x =real;  num.i =imag;
    1      14
    1      15   #define ADD_COMPLEX(res, op1, op2) \
    1      16     res.x =op1.x+op2.x; \
    1      17     res.i =op1.i+op2.i;
    1      18
    1      19   #endif   /* !defined(complex_h) */
    0       2
    0       3   extern void write_complex(struct complex num);
    0       4
    0       5   main()
    0       6   {

                .text
                   .align   4
                   .def _main; .val _main;  .scl 2; .type 0x40;
.endef
                .globl      _main
                 #  Function 'main'
                 #  Registers used: g0 g1 g2 g3 g4 g5 g6 g7 fp r4*
                 #    r5* r6* r7*
              _main:
                 lda   48(sp),sp
                 #Prologue stats:
                 #  Total Frame Size: 48 bytes
                 #  Local Variable Size: 48 bytes
                 #  Register Save Size: 0 regs, 0 bytes
                 #End Prologue#
    0       7  register struct complex x,y,z;
    0       8
    0       9  INIT_COMPLEX (x, 10.31, 4.25);
        +++++   x .x =  10.31 ;   x .i =  4.25 ; ;
                # lda1.031000000000000004974e1,r4
                lda0x51eb851f,r4
                lda0x40249eb8,r5
                movlr4,r6
                stlr6,64(fp)
                # lda4.250000000000000000000e0,r4
                mov0,r4
                lda0x40110000,r5
                movlr4,r6
                stlr6,72(fp)
    0      10 INIT_COMPLEX (y, 7.14, 5.23);
        +++++   y .x =  7.14 ;   y .i =  5.23 ; ;
                # lda7.139999999999999968026e0,r4
                lda0x28f5c28f,r4
```

```
                  lda0x401c8f5c,r5
                  movlr4,r6
                  stlr6,80(fp)
                  # lda5.23000000000000042633e0,r4
                  lda0x1eb851ec,r4
                  lda0x4014eb85,r5
                  movlr4,r6
                  stlr6,88(fp)

0          11 ADD_COMPLEX  (z, x, y);
      +++++   z .x =  x .x+  y .x;  z .i =  x .i+  y .i; ;
0          12
                  # lda1.74499999999999992895e1,r4
                  lda0x33333333,r4
                  lda0x40317333,r5
                  movlr4,r6
                  stlr6,96(fp)
                  # lda9.48000000000000042633e0,r4
                  lda0x8f5c28f6,r4
                  lda0x4022f5c2,r5
                  movlr4,r6
                  stlr6,104(fp)
0          13 write_complex (z);
                  ldq96(fp),g0
                  callj_write_complex
0          14   }
                  #EPILOGUE:
                  ret
                  .def  _main;  .val  .;  .scl  -1;  .endef
```

The listing file includes information about the compilation. The heading line at the beginning of the listing contains the name and version of the compiler, the printing date of the listing, and the name of the primary source file. The next two lines of text describe the format of the listing. The remainder of the file contains the listing. The compiler does not paginate the listing and does not wrap long lines.

The format of the source text listing is as follows:

```
include-nesting-level  line-number  source-line
```

`include-nesting-level`	determines the depth of the file in the include file nesting hierarchy. Since lines from the primary source file are always at level 0, if you do not list included source text, all source lines in the listing are at level 0. An asterisk (`*`) following the include nesting level indicates the first line of a file.
`line-number`	is the location of a line relative to the beginning of the file containing that line.
`source-line`	is a line of source text.

A line with an expanded macro appears after the corresponding source line in the following format:

```
     source-line
+++++  macro-expanded-line
```

`macro-expanded-line`	is the source line containing the expansion of the macro.

The assembly language in the listing is similar to but not necessarily identical to the intermediate assembly language form of the program resulting from an `S` (Save-assembly) option. The compiler can add symbolic names that improve readability of the listing but are not accepted by the assembler.

Related Topics

Stop-after options `w` (Diagnostic-level) `Z` (Listname)

Program-Wide Analysis and Optimization 4

Introduction

This chapter teaches you how to use some of CTOOLS most powerful optimization features. This chapter discusses these topics:

- “Creating Program-wide and Module-local Optimizations”
- “Profiling Your Program”
- “Using make To Perform Program-wide Optimizations”
- “Runtime Support for Profile Collection”

To use the first two features you are going to:

1. Create a program database.
2. Specify which modules you want optimized.
3. Recompile your program using the `-fdb` option.

After these basic optimizations, you use profiling to gather information about the runtime characteristic of your program and then optimize performance based on that information.

The sections that follow describe the types of optimizations used in program optimization.

Individual and Program-wide Optimizations

The compiler can perform sophisticated inter-module optimizations, such as replacing function calls with expanded function bodies when the function call sites and function bodies are in different object modules. These are called program-wide optimizations because the compiler collects information from multiple source modules before it makes final optimization decisions. Throughout this chapter, standard (i.e., non-program-wide) optimizations are referred to as module-local optimizations.

About Profiling

The compiler can also collect information about the runtime behavior of a program by instrumenting the program. The instrumented program can be executed with typical input data, and the resultant program execution profile can be used by the global decision making and optimization phase to improve the performance of the final optimized program. The profile can also provide input to the global coverage analyzer tool (gcov960), which gives users information about the runtime behavior of the program at the source-code level.

Creating Program-wide and Module-local Optimizations

Program-wide optimizations are enabled by options that tell the compiler to:

1. Build a program database during the compilation phase.
2. Invoke a global decision making and optimization step during the linking phase.
3. Automatically substitute the resulting optimized modules into the final program during the linking phase.

Specifying the Program Database Directory

The program database directory (PDB) is the repository for all program-wide optimization information about a particular program. When using program-wide optimizations, you must specify the correct PDB to all compilation tools involved in building the program. You initially create the PDB, but the files within this directory are automatically managed by the various pieces of the program-wide optimization system. Once this is done, you do not change the files in the PDB.

The PDB can be specified by setting the environment variable G960PDB (gcc960 driver) or I960PDB (ic960 driver) to the correct location. You can also specify the PDB at compiler invocation time with the Z*dir* (gcc960) or Yd,*dir* (ic960) option, as shown in the examples below.

```
gcc960  -Zmypdb  foo.o
ic960  -Yd,mypdb  foo.o
```

Compiling for Program-wide Optimization with the fdb Option

All modules subject to program-wide optimization must be initially compiled with the fdb option (described in Chapter 2, "gcc960 Compiler Driver" and Chapter 3, "ic960 Compiler Driver"). Using this option causes the insertion of program database information in the object modules, and it implies a minimum module-local optimization level of O1 (although higher module-local optimization levels are allowed).

Compiling with the fdb option does not change the code or data generated for the object modules in any way; this option simply makes information collected during the initial compilation of the modules available to the global decision making and optimization step.

Global Decision Making and Optimization Using the gcdm Option

The tool that performs the global decision making and optimization step is called gcdm960. gcdm960 is invoked from within the linker when the gcdm option is used. You can also use the gcdm option in the compiler driver (gcc960 or ic960) to pass this option to the linker. Using the gcdm option causes gcdm960 to:

- automatically build and manage optimized object modules in the PDB
- arrange with the linker for optimized object modules from the PDB to be automatically substituted for some or all of the original object modules in the final program.

You can use multiple gcdm options in a linker or compiler invocation command, and each gcdm option can have multiple comma-separated arguments. (The gcdm option and its arguments are fully described in Chapter 4, "Program-Wide Analysis and Optimization".)

Selecting Modules for Optimization with Substitution Specifications

You tell gcdm960 which object modules to optimize and how to optimize them with substitution specifications. Substitutions are specified by arguments to the `gcdm` option in the linker or compiler invocation.

The term "substitution" reflects the fact that the linker replaces your `.o` files with optimized versions maintained in the PDB. Such a `.o` file from the PDB is called a "substitution module."

The example below illustrates the basic idea of substitution: It depicts an ic960 invocation command that uses the `gcdm` option and the `Yd` and `fdb` options to accomplish program-wide optimization (without profiling) for a simple program.

```
ic960 -o prog -Ttarg -Yd,./pdb -gcdm,subst=+O5 -fdb fee.c
     foo.c
```

(`-Ttarg` specifies the linker directive file for the target execution environment.)

The command accomplishes the following steps:

1. `fee.c` and `foo.c` are compiled with `fdb`, which inserts program database information into `fee.o` and `foo.o`.
2. The program is then linked to form `prog`, at which time gcdm960 is invoked with `-Yd,./pdb -gcdm,subst=+O5`.
3. `fee.o` and `foo.o` are replaced in `prog` with versions from `./pdb` built at level `O5` optimization (that is, built with program-wide optimizations).

NOTE. *The optimized replacements for* `fee.o` *and* `foo.o` *are present in the linked program but never appear in the current working directory.*

Profiling Your Program

Compiling for Profile Instrumentation with -fprof

As mentioned above, information on the runtime behavior of the program can be used by the compilation system during the global decision making and optimization step. To instrument a program, use the `fprof` option in addition to `fdb` when compiling:

```
ic960 -Yd,mypdb -fdb -fprof -c foo.c
```

See Chapter 2, "gcc960 Compiler Driver" and Chapter 3, "ic960 Compiler Driver" for more on the `fprof` option. This command causes profile instrumentation to be inserted into `foo.o` so that when the linked program is executed, a profile can be collected. Using runtime profiles to influence the final optimization of your program requires you to build the program twice: once to insert the instrumentation, as described here, and then again so that the compilation system can substitute modules that are recompiled with optimizations appropriate to their runtime behavior.

Collecting a Profile

If a program that contains one or more modules compiled with `fprof` is linked with the standard libraries and then executed, a file named `default.pf` containing the profile for those modules is automatically produced when the program exits. This is a "raw" profile containing program counters that log how many times various statements in the source program have been executed.

If you are not using the standard libraries, you must insert a call to a routine that creates the profile in an appropriate point in the program source code. For instructions on this step, see the section titled Runtime Support for Profile Collection (page 4-15). If you are using IxWorks*, functions are provided for collecting profiles (see page 4-14).

Building Self-contained Profiles with gmpf960

A “raw” profile contains program counters, which count how many times various statements in the source program have been executed. Information in the PDB is needed to correlate these program counters with the source program.

A raw profile (that is, a profile simply collected as described previously) has a very short useful life. When changes are made in your source code, any raw profiles previously obtained for that program are no longer accepted by the global decision making and optimization step.

A “self-contained” profile captures the program structure from the PDB and associates it with the program counters from the raw profile. When changes are subsequently made to the source program, the global decision making step interpolates or “stretches” the counters in the self-contained profile to fit the changed program.

A self-contained profile can be continually used to optimize the program it was collected for, even after days, weeks, or perhaps months worth of changes to the program. This frees you from having to collect a new profile every time the program changes, while still allowing profile-directed optimizations. Depending upon the nature and quantity of changes to the program, the accuracy of the profile gradually degrades over time as more interpolation is done.

A self-contained profile must be generated from a raw profile before the program that generated the raw profile is relinked. You should always create a self-contained profile immediately after the raw profile is collected.

To create a self-contained profile, use the gmpf960 profile merger tool. gmpf960 is invoked with the raw profile as an input file, as shown in this example:

```
gmpf960 -Z  mypdb -spf pfile2.spf pfile1.pf
```

This command creates a self-contained profile `pfile2.spf` from the raw profile `pfile1.pf`. The raw profile `pfile1.pf` was created by executing the instrumented program that was linked using `mypdb` as the program database directory. The `.pf` and `.spf` filename extensions for the profile files in this example are arbitrary; the different types of profiles are recognized by their contents, not by their filename extensions.

After a self-contained profile is created, you can specify it for the global decision making and optimization step using the `gcdm,iprof=file` syntax as described in the next section.

Using Profiles During Global Decision Making and Optimization with -gcdm,iprof

To supply a profile file `pfile` to the global decision making and optimization step, simply add the following option and argument to the compiler or linker invocation command:

```
gcdm,iprof=pfile
```

This is in addition to the `gcdm,subst` option. The `iprof` argument can specify either raw profiles or self-contained profiles.

Obtaining Program Coverage Analysis with gcov960

You can use both profile types as input to the gcov960 coverage analyzer tool, as follows:

```
gcov960 -cm -Z ./pdb -iprof pfile.pf fee.c foo.c
```

This command produces a coverage report in the files `fee.cov` and `foo.cov`, using the profile `pfile.pf`.

Using make To Perform Program-wide Optimizations

Since the program-building tool “make” is so widely used, the program-wide optimization features are designed to work well with it. However, you need not use the make tool to effectively use program-wide optimizations. If you do not use the make tool, you can skip this section.

Below is an example of a makefile (where *`targ`* is set appropriately):

```
SUBST=
PROF=
MODULES=*:*
OPT=-fdb
"-gcdm,subst=$(MODULES)+$(SUBST),iprof=$(PROF)"
FLAGS=-Ttarg $(OPT)
OBJECTS=fee.o foo.o main.o
```

```
prog: $(OBJECTS) force
     ic960 -o prog $(FLAGS) $(OBJECTS)
.c.o:
     ic960 -c $(FLAGS) $<
$(OBJECTS): makefile
force:
```

While primitive, this makefile can be used to exercise several significant capabilities of the program-wide optimization system. Refer to this example as you read the following sections; the example and discussion can help you determine the changes that must be made to your own makefiles (if any) to perform program-wide optimizations.

Adapting Makefiles for Program-wide Optimization

This section discusses the example makefile and how the program-wide optimization interface is expected to mesh with your current usage of optimization and debug options.

Specifying the PDB in the Makefile

In an ic960 or gcc960 development environment, you typically specify the PDB by setting the `I960PDB` or `G960PDB` environment variable outside of any makefile, rather than changing makefiles to specify the PDB to every tool invocation. The example makefile assumes that the PDB is specified outside of the makefile in this manner.

The appropriate location for the PDB directory is probably in the directory where the makefile compiles and links the object modules. For example, the UNIX and Windows statements below are suitable for many users.

```
setenv I960PDB ./pdb         (UNIX)
set I960PDB ./pdb            (Windows)
```

Replacing Optimization Options with fdb and gcdm

Except for the definition of the `OPT` macro, the example is typical of simple makefiles that use ordinary optimizations. From the point of view of the makefile and/or the build system, the `fdb` option combined with one or more gcdm options is often a direct replacement for ordinary optimization options such as O, because the compilation tools that accept ordinary optimization options also accept program-wide optimization options.

Programs linked by direct invocation of the linker are exceptions to this general rule. In such a case, the gcdm option must be added to the linker invocation.

Using Linker Invocations with gcdm for Automatic Management of Object Files at Link Time

The example makefile always produces a program load module with the same name. Since the options provided when the make tool is invoked affect the linked program when there have been no apparent changes to the source or object files, the makefile uses an artificial force dependence to guarantee that the program is linked at every invocation of make. This is a common practice, and keeps the makefile simple.

You could instead write the makefile so that different options to the link step produce program load modules with different names. The artificial force dependency could then be removed, perhaps saving an occasional unnecessary linker invocation. However, in the program-wide optimization system there is no more reason to try to eliminate extra linker invocations than there would be in an ordinary system. In fact, the development environment can often be simplified by forcing linker invocations (as in the example makefile) for the following reasons:

- The global decision-making and optimization step manages the results of previous work in the PDB so that all previously generated modules are reused whenever possible. The system keeps multiple sets (currently, two) of the most recently used substitution modules in the PDB, indexed by the substitutions that generated them. The makefile is not aware of this management task, and is simpler as a result.
- Even though program-wide optimizations can potentially trigger large quantities of compilation and optimization work at link time, the majority of this work usually occurs only the first time the program is linked with a particular set of substitutions, or on the first link after major changes are made to the program.

- The automatic management of substitution modules (defined in the Selecting Modules for Optimization with Substitution Specifications section) greatly simplifies some development tasks that are difficult for users in an ordinary environment, such as maintaining both debug and optimized versions of the object modules for a program. Given modules already compiled with the `fdb` option, users can have alternate program load module versions built efficiently by simply invoking the linker with appropriate `gcdm,subst` options.

See the next section for examples of using the sample makefile to automate program-wide optimizations.

Using Makefiles with Program-wide Optimizations for Common Development Tasks

Building an Optimized Program without Profiling

Using the example makefile, if you want to obtain a program built with program-wide optimizations, pass the options you want through the `SUBST` macro when invoking the make tool. For example, if you want level `O5` optimization, use:

```
make SUBST=O5
```

This causes the object modules in the program to be compiled and then linked with the options in the `FLAGS` macro. The make tool then issues the following commands:

```
ic960 -c -Ttarg -fdb -gcdm,subst=*:*+O5,iprof= fee.c
ic960 -c -Ttarg -fdb -gcdm,subst=*:*+O5,iprof= foo.c
ic960 -o prog -Ttarg -fdb -gcdm,subst=*:*+O5,iprof=
     fee.o foo.o
```

The link command causes substitution modules at optimization level `O5` to be built in the PDB to replace the original modules `fee.o` and `foo.o` in the program load module `prog`. The `iprof=` option without a filename indicates that you are not using a profile, which is the default behavior.

Building for Debugging without Program-wide Optimizations

If logic problems exist in the program, you can build a debug version of `prog` by invoking the make tool with:

```
make SUBST=g+O0
```

This causes the make tool to issue only the following link command (assuming the sources haven't changed):

```
ic960 -o prog -Ttarg -fdb -gcdm,subst=*:*+g+O0,iprof=
     fee.o foo.o
```

The link command causes substitution modules with no optimization and full debug information to be built in the PDB to replace the original modules `fee.o` and `foo.o` in the program load module `prog`.

After debugging the problem and then fixing it by changing one of the source files, you can reissue the `make SUBST=O5` command to get another program-wide optimized version of `prog`. Invoking the make tool recompiles the changed source file and then links the program with the `O5` substitution specification, as before. This causes the global decision making and optimization step to recompile the previous `O5` substitution modules as needed in the PDB, and those modules are then used in the program load module `prog`.

Building an Instrumented Program

You can create a profile-instrumented program either of two ways: compile source modules with the `-fprof` option, or link object modules using a `-gcdm,subst=+fprof` substitution.

- When compiling with `-fprof`, the object files generated in your working directory contain profile-instrumented code.
- When compiling with `-gcdm,subst=+fprof`, the profile-instrumented object files reside in the PDB, not in your work space.

These approaches both yield the same instrumented version of `prog`. However, compiling with the `fprof` option creates object modules useful only for collecting a profile. If you compile with the `fprof` option and do not want a profile, you must then use substitutions to replace every instrumented module in `prog`, or you must recompile the modules without the `fprof` option.

Linking Using an +fprof Substitution

The example makefile requires no changes to accommodate this method; just use:

```
make SUBST=fprof
```

No files are recompiled unless source files have changed; only the following link command is issued:

```
ic960 -o prog -Ttarg -fdb -gcdm,subst=*:*+fprof,iprof=
     fee.o foo.o
```

This command causes substitution modules with profile instrumentation to be compiled in the PDB to replace the original modules `fee.o` and `foo.o` in the linked program `prog`.

NOTE. *Profiles collected with* `+fprof` *substitutions must be made into self-contained profiles before linking.*

Compiling Using the fprof Option

To use the `fprof` compiler option to create an instrumented load module:

1. Edit the makefile to add `-fprof` to `FLAGS`.
2. Invoke the make tool without any substitutions, as follows:
   `make SUBST=`

   Since the object files depend on the makefile, and the makefile is edited, the make tool recompiles the modules before linking them:

```
ic960 -c -Ttarg -fdb -fprof -gcdm,subst=*:*+,iprof=
     fee.c
ic960 -c -Ttarg -fdb -fprof -gcdm,subst=*:*+,iprof=
     foo.c
ic960 -o prog -Ttarg -fdb -fprof -gcdm,subst=*:*+,iprof=
     fee.o foo.o
```

Since the substitution option list is empty, there are no substitutions, and the instrumented modules from the current working directory are linked.

Note that when you use the `fprof` option in this manner, the generated object module contains code that is unsuitable for linking into programs that are not intended to collect profile information. To solve this problem, you can use `+fprof` with `gcdm,subst` instead of using `fprof` when compiling.

Building an Optimized Program with Profiling

Assuming you have collected a profile named `prog.pf` by executing the instrumented version of `prog`, you can then use it for program-wide optimizations by invoking the make tool as follows:

```
make SUBST=O5 PROF=prog.pf
```

`prog.pf` can be either a raw profile or a self-contained profile. If `prog.pf` is a self-contained profile, you can continue to use it as shown above, even if changes are made to the program.

Profiling a Program in Pieces

Suppose that the target execution environment is memory limited so that `fee.o` and `foo.o` cannot both be instrumented for profiling at the same time. You can use substitutions to make partially instrumented versions of `prog`, and then create self-contained profiles for each piece, as follows:

```
make SUBST=fprof MODULES=":fe*"
```

Execute `prog` to obtain raw profile `default.pf`.

```
gmpf960 -spf fe1.spf default.pf
make SUBST=fprof MODULES=":fo*"
```

Execute `prog` to obtain a new raw profile `default.pf`.

```
gmpf960 -spf fo1.spf default.pf
```

Note that neither of the invocations of the make tool causes compilations; the make tool simply issues a link command in each case. Each link command constructs a version of `prog` that has a limited set of instrumented modules:

```
ic960 -o prog -Ttarg -fdb -fprof
     -gcdm,subst=:fe*+, iprof=  fee.o foo.o
ic960 -o prog -Ttarg -fdb -fprof -gcdm,subst=:fo*+,
     iprof=  fee.o foo.o
```

Note also that although the example contains only two modules, the strings that select the modules for partial program instrumentation use a general regular expression mechanism. Such strings can select any possible subset of the modules in a program for any substitution. This mechanism is discussed in detail with the `gcdm,subst` option in Chapter 6 "gcdm Decision Maker Option".

After creating the self-contained profiles `fe1.spf` and `fo1.spf`, use gmpf960 to create a single merged self-contained profile:

```
gmpf960 -spf prog.spf fe1.spf fo1.spf
```

The final `prog.spf` is identical to a profile obtained by instrumenting the entire program at once. Now issue the `make` command to get program-wide optimizations guided by `prog.spf`:

```
make SUBST=O5 PROF=prog.spf
```

Again, the make tool performs no compilations. The following link command is issued:

```
ic960 -o prog -Ttarg -fdb -gcdm,subst=*:*,+O5,
     iprof=prog.spf fee.o foo.o
```

This causes substitution modules at optimization level `O5` to be built (guided by the profile in `prog.spf`) to replace the original modules `fee.o` and `foo.o` in the program load module `prog`.

Runtime Support for Profile Collection for the IxWorks* Environment

Starting with CTOOLS release 6.5, the CTOOLS distribution includes a new profiling library that can be used in the Windriver Systems IxWorks* runtime environment with an i960® Rx processor. The library is named libixqrp.a and includes the following two routines that can be used to initialize and collect profile data. The above routines can be invoked from the Tornado* shell.

```
__ddmProfileClear():
```

This routine zeros all the profile counters and should be called at the beginning of the profile collection run.

```
__ddmProfileOutput():
```

This routine outputs all the profile information on to stdout and should be called at the end of the profile collection run. The file default.pf is not created when using IxWorks.

To link in this library, use the `lixq` linker switch.

NOTE. *If you are generating a relocatable module using the* `r` *linker switch, make sure that you use the* `P` *linker switch to include the profiling information used by the compiler in the generated relocatable module.*

Runtime Support for Profile Collection

When you link your instrumented program with the standard libraries supplied with CTOOLS and startup code, when your program exits, a raw profile named `default.pf` is automatically produced in the current directory. The format of this file is described in Chapter 5, "Profile Data Merging and Data Format (gmpf960)".

When you are not using the standard libraries or not using IxWorks, you must provide code to initialize the profile counters and to dump the counters in the required format, as described below.

Profile Initialization

Your startup code must call a profile initialization routine before calling `main`. The address of the default initialization routine is held in the predefined variable `__profile_init_ptr`. Here is an example of a call to the default initialization routine:

```
.comm  __profile_init_ptr
ld             __profile_init_ptr, r6
cmpobe 0, r6, 0f
lda            0(ip), g0
lda            ., g1
subo   g1, g0, g0
addo   g0, r6, r6# adjust for PIC
callx  (r6)
0:
```

Profile Data Merging and Data Format (gmpf960) 5

This chapter explains how to use gmpf960 to merge the execution profile data you learned how to collect in Chapter 4, "Profile Data Merging and Data Format (gmpf960)". You also learn how to use gmpf960 to create a report that shows how many times each basic block was "hit" or run during program execution.

Merging Profile Data

The gmpf960 utility combines the execution profiles created while executing an instrumented program. Once the profiles are merged, the gcdm960 utility uses the merged profile information to analyze the program's run-time characteristics and make decisions about possible program-level optimizations. For more information about gcdm960, see Chapter 6, "gcdm Decision Maker Option".

You can merge any mixture of the raw or self-contained profiles. See Chapter 4, "Profile Data Merging and Data Format (gmpf960)". The merged profile is normally a self-contained profile, although you can merge raw profiles into a single raw profile.

If the execution environment supports a file system, and the application uses the supplied libraries, then the process of gathering and formatting the data is automatic. When your instrumented program terminates, the profile data file `default.pf` is automatically written.

gmpf960 Invocation

The profile-merge utility recognizes a letter preceded by a hyphen `-` (or on Windows hosts only, a slash `/`) as an option. For example, `-o` specifies the Outfile option on all hosts; `/o` is also accepted on Windows hosts. gmpf960 uses the syntax:

```
gmpf960 [-option]... {-spf | o outfile} infile
[infile]...
```

An *option* is one of:

`h` displays a list of invocation options.

`rprofile` indicates how many times the counters for each basic block were incremented, for those blocks that were hit. This information is written to `stdout`.

`t` specifies that all input files are in text format.

`v960` displays version information and exits.

`Z` *pdb_dir* specifies the program database directory. If the merged output or any of the inputs is a self-contained profile, you must specify the PDB with the `Z` option or via the `G960PDB` or `I960PDB` environment variable.

`spf` *outfile* causes a self-contained profile to be produced as output. This is the preferred usage of gmpf960.RWLRobert W. Lee

`o` *outfile* specifies the output file. If a file with that name already exists, it is overwritten. You can even use the name of one of the input files. White space is optional between the option and argument. Note that this option is supported only for merging raw profiles into another raw profile.

infile specifies an input file. You can specify multiple input filenames; gmpf960 processes them sequentially. Input files can be the results of a program execution or a previous merging of profiles.

Discussion

The gmpf960 utility merges the execution profiles in all *`infile`* files and stores the resulting profile in *`outfile`*. Input files can be either the output from a previous invocation of gmpf960, or the `default.pf` profiles created automatically when you run your instrumented program.

NOTE. *The tools that accept profiles generally accept multiple profiles and merge them in the same manner as gmpf960. However, gmpf960 is the only tool that actually produces profiles, and in particular, is the only tool that can produce a self-contained profile by conversion from a raw profile. The other tools always perform the merge internally and discard the merged profile after processing.*

The `t` option is useful if your execution environment does not support automatic creation of the `default.pf` profile file. Use `t` if your input files are in the text format described below.

If the `t` option is not specified, the input files are assumed to be in their default binary format. Input files can be either the output from a previous invocation of gmpf960, or the `default.pf` profiles created automatically when you run your instrumented application.

Example

The following command reads and processes `run1.pf`, `run2.pf`, `run3.pf` and merges the results into the self-contained profile `summ.spf`.

```
gmpf960 -spf summ.spf run1.pf run2.pf run3.pf
```

Profile Format Specification

Normally, the raw profile file `default.pf` is created automatically when your application calls `exit`. Alternatively, the gdb960 debugger supports a `profile put` command that you can use to extract the profile data from target memory and write it to `default.pf` in the appropriate format.

If your execution environment does not support automatic generation of default.pf, you must manually extract the profile data from your system's memory and write it to a file in a format recognized by gmpf960.

The remainder of this section describes where the profile data resides in target memory, and the file formats recognized by gmpf960.

Profile Data Structures

When you build an instrumented application, a supporting C data structure is automatically linked with your application. This data structure is used to record your application's runtime behavior, or "profile."

The profile data is maintained in an array of unsigned long integers, called __profile_data_start. The size of the array, in bytes, is given by the symbol __profile_data_length. __profile_data_length is always a multiple of 4, and the number of elements in __profile_data_start is given by (__profile_data_length / 4).

default.pf File Format

The file default.pf is a binary file containing the value of __profile_data_length, followed by elements of __profile_data_start. Each value is stored in the file as a 4-byte two's complement unsigned integer. Furthermore, each value is stored in little-endian byte order, regardless of the endianness of your target memory and of your host system.

For example, assume that __profile_data_length has the value 12 (12 bytes is three 4-byte words), and that __profile_data_start contains the values 0x000090A4, 0x000000C7, and 0x00008FDD. Then the binary format of file default.pf (printed as hexadecimal words) would be:

```
0000000C
000090A4
000000C7
00008FDD
```

Depending on the tools available, you may find it difficult to create the binary format required in default.pf. To circumvent this step, you can write the profile data to a file in text format, and then use gmpf960 to translate the file into binary format.

The text file format consists of the value of `__profile_data_length`, followed by the values in `__profile_data_start`. The numbers must appear in the file as decimal, and must be separated by white space.

For example, assume that `__profile_data_length` has the value 20 (20 bytes is five 4-byte words), and that `__profile_data_start` contains the values 20, 15, 100, 2, and 63. If you use a text editor to create the text format of `default.pf`, it would be:

```
20
20 15 100
2 63
```

Note that there is no requirement as to the number of entries per line. The format definition of the text file simply requires that the numbers are separated by white space.

Example

Assume that you have a text-format profile in file `default.txt` and a binary-format profile in file `default.pf`. The following invocations of gmpf960 merge these two profiles, writing the results in the binary-format file `default.sum`.

```
gmpf960 default.txt -o default.tmp
gmpf960 default.pf default.tmp -spf default.sum
```

Any mixing of text, raw profile or self-contained profiles is allowed.

Creating a Runtime Report with gmpf960

You can also use gmpf960 to create a report that shows how many times the counters for each basic block were incremented. The examples given below assume that you compile and execute the following source file with the -fprof option to gather a runtime profile.

Example 5-1 C Code

```
/* Source File - t.c */

int i, j;

main()
{
        for ( i = 0; i < 10; i++ )
                j += i;

        return j;
}
```

To compile the above source file you can use the following command:

```
gcc960 -Fcoff -fprof -Tmcycx t.c -Z pdb
```

The generated executable file `a.out` can be downloaded to a Cyclone i960 Cx processor-based evaluation board and executed using the following command

```
mondb -ser a.out
```

This execution generates the `default.pf` file which contains the runtime profile for the above execution. You can use either `rprofile` option in gmpf960 or the gcov960 coverage analyzer to get the coverage results after running the program.

Using gmpf960

The command:

```
gmpf960 -spf foo.spf -rprofile -Z pdb default.pf
```

generates the following output:

Example 5-2 gmpf -rprofile Sample Output

```
Profile counts for module t.c=main$
Function name                      Line#  Block#   Times hit   From
========================|========|======|=========|======
main                      |     4 |     0 |        1 | 1 raw inputs
main                      |     5 |     0 |        1 | 1 raw inputs
main                      |     5 |     3 |       11 | 1 raw inputs
main                      |     5 |     2 |       10 | 1 raw inputs
main                      |     6 |     1 |       10 | 1 raw inputs
main                      |      8 |     4 |        1 | 1 raw inputs
```

Notice that the in the example above, the expressions in the `for` loop and the expression `j += i` are the only ones with multiple hits. The gcov960 sample output below provides you with the same information, however, the number of hits for each statement is recorded to the left of the line.

Using gcov960

The command:

```
gcov960 -rl -Z pdb
```

generates the following output:

Example 5-3 gcov960 Sample Output

```
int i, j;

              main()
        1 -> {
  1 11 10 ->    for ( i = 0; i < 10; i++ )
       10 ->            j += i;

        1 ->    return j;
              }


Number of Blocks:                                                 5
Number of Blocks Executed:                                        5
Number of Blocks Never Executed:                                  0
Percentage of Blocks in Source File that were executed:      100.00%

Program database:
Program profile:          default.pf
```

See the *i960 Processor Software Utilities User’s Guide* for more information on gcov960.

gcdm Decision Maker Option

6

This chapter describes the `gcdm` option, which invokes the gcdm960 global optimization decision maker during the link process. The decision maker then invokes the compiler and linker as necessary to perform program-wide optimizations. For an overview of how to use this option, see Chapter 4, “Program-Wide Analysis and Optimization”.

gcdm Option Syntax

The `gcdm` option has the following syntax:

```
{ - | / } gcdm,argument[,argument]...
```

As with other options, you can use the `/` delimiter only in Windows. The `gcdm` option *`arguments`* and the sections that describe them are listed in Table 6-1.

Table 6-1 gcdm Option Arguments (Sheet 1 of 2)

gcdm Option Arguments	Description	Section Titles
• subst={*module-set*}{*option-list*} • nosubst=*module-set*	Controls which modules are substituted.	Substitution Controls
• [no]ref=*module-set*	Specifies whether functions or data defined in objects reside outside the current module set presented to the linker.	External Reference Controls

Table 6-1 gcdm Option Arguments (Sheet 2 of 2)

gcdm Option Arguments	Description	Section Titles
• inline=*n*	Sets the level of inlining used by the compiler.	Inlining Level Control
• iprof=*file*	Causes profile information to be used in program-wide optimizations.	Input Profile Control
• sram=*start*, *end*[,*start*, *end*]... m=*start*, *len*[,*start*, *len*]...	Specifies fast memory regions (e.g., SRAM) to use for heavily referenced variables.	Fast Memory Controls
• dryrun	Writes a list of the current subst commands to a text file.	Dryrun Control
• dec=*file* • rsummary • rdecisions • rcall-graph • rreverse • rprofile • rvariables	Options for creating gcdm reports.	Report Controls

gcdm Option Arguments

Substitution Controls

The substitution controls allow you to substitute optimized modules into your application (using `gcdm,subst`), and to suppress unintended substitutions (using `gcdm,nosubst`). When a given object module is named in multiple `subst` or `nosubst` options, the last `subst` or `nosubst` that names the module applies. The substitution controls also allow fine control of how affected modules are optimized. The following subsections describe substitution and substitution suppression. Detailed information on controlling optimizations is presented in the discussion of *`option-list`* in the next subsection.

Substitution Specifications

`subst={`*`module-set`*`}{`*`option-list`*`}`
In the linked program, `gcdm,subst={`*`module-set`*`}{`*`option-list`*`}` causes substitution of modules optimized according to the *`option-list`* for all of the modules in *`module-set`*. Note that no space is allowed between *`module-set`* and *`option-list`*.

A *`module-set`* specification is a string supplied by the user that names the modules to be affected by the `gcdm` option. For a description of how to specify a *`module-set`*, see "Module-set Specification" at the end of this chapter.

An *`option-list`* can consist of one or more of the substitution options discussed in three categories below. Note that the first two categories are mutually exclusive; you can use substitution options from the third category with those from either of the first two categories. (For example, the `+O5` control is incompatible in a substitution with the `+fprof` control.) An option list can also consists of a single `+`, specifying no substitution.

Whole-program Optimization Option (Category 1)

`+O5`

This option selects program-wide optimizations, including global function inlining, superblock formation, and global alias analysis. This option is not allowed in an *`option-list`* with module-local (Category 2) options.

Module-local Optimization Options (Category 2)

`+fprof  +O0  +O1  +O2  +O3  +O4`

These module-local substitution options correspond to the gcc960 and ic960 drivers' `-fprof` (Instrument) and `-O`*`n`* (Optimize) options described in Chapter 2, "gcc960 Compiler Driver" and Chapter 3, "ic960 Compiler Driver". (The compilation system interprets the `-O`*`n`* arguments correctly, based on which compiler driver you are using.) The module-local substitution options are not allowed in an *`option-list`* with whole-program optimization (Category 1) options.

`+fprof`
: causes generation of profile instrumentation, as described for the `-fprof` compiler option (in Chapter 2, "gcc960 Compiler Driver" and Chapter 3, "ic960 Compiler Driver"). When the `+fprof` substitution option is used (instead of the `-fprof` compiler driver option), only the substitution modules in the PDB contain the actual instrumented code. This is useful in some cases. For example, a library compiled with `-fdb` but without `-fprof` is suitable both for users who do not want to use program-wide optimizations, and for those who do, as follows:

- All program database information required to support program-wide optimizations is present in the library, since it is compiled with `-fdb`.
- To collect a full program profile (including the library) for use with program-wide optimizations, a substitution such as `-gcdm,subst=*:*+fprof` generates a program that is appropriately instrumented.
- If you do not use program-wide optimizations (that is, you do not use gcdm,subst options), there is no extra runtime overhead, and the program can be optimized to any module-local optimization level higher than -O0.

`+O0 +O1 +O2 +O3 +O4`
: allow substitutions of modules with various levels of module-local optimization. (The compilation system interprets the `-On` arguments correctly, based on which compiler driver you are using.) These are typically used for the following purposes:

- to substitute a few non-optimized modules into a program built with program-wide optimizations in order to help debug it.

- to specify a module-local optimization level other than O1 with a +fprof substitution.

Miscellaneous Substitution Options (Category 3)

`+g  +asm_pp+`*`prog`*`  +clist+`*`arg`*`  +f`*`string`*

These can be used with either the whole-program or module-local substitution options in Categories 2 and 3, above.

`+g` enables debug information generation for substitution modules.

`+asm_pp+`*`prog`* causes *`prog`* to be invoked after the assembly code for a substitution module is generated, with the name of the file containing the substitution assembly code as its third argument. (The first two arguments are ignored.) This allows the post-processing of substitution assembly code by user-supplied tools.

`+clist+`*`arg`* generates a listing containing assembly code and/or preprocessed source code of each module selected by the substitution into a file named *`name`*`.L` in the current working directory, where *`name`* is the base filename of the object module being substituted. *`arg`* is a string consisting of *`s`*, `o` or both. The `s` character causes inclusion of the substitution module's pre-processed source code in the listing. The `o` character causes inclusion of the substitution module's assembly code in the listing. In order for preprocessed source code to be displayed in listings generated by `clist` substitutions, the modules must either have been originally compiled with the ic960 driver or compiled with the gcc960 driver using the `ffancy-errors` (ic960) or `fmix-asm` (gcc960) option.

`+f`*`string`* The `+f`*`string`* substitution options listed below apply the corresponding individual `-f`*`string`* optimization options discussed in Chapter 2, "gcc960 Compiler

Driver" and Chapter 3, "ic960 Compiler Driver". The `no` form of these options (e.g., `+fno-unroll-loops`) is also accepted.

```
+fbbr,  +fcoalesce
+fcondxform,  +fconstprop
+fcopyprop,  +fcse-follow-jumps
+fcse-skip-blocks,  +fdead-elim
+fexpensive-optimizations, +ffunction-cse
+fmarry_mem,  +fpeephole
+frerun-cse-after-loop,  +fsblock
+fsched-sblock,  +fschedule-insns
+fschedule-insns2,  +fshadow-globals
+fshadow-mem,  +fspace-opt
+fsplit_mem,  +fstrength-reduce
+fthread-jumps,  +funroll-all-loops
+funroll-loops
```

These options automatically default appropriately based on the selected optimization level.

Substitution Suppression

nosubst=*module-set*

The nosubst=*module-set* argument suppresses substitution for the named modules. This is equivalent to subst=*module-set*+ (the *option-list* consists only of a + character). `nosubst` is typically used to exclude a subset of modules from a previous `subst`.

For example, the `gcdm` option and argument:

```
-gcdm,subst=*:*+O5,nosubst=:intr_handler
```

would substitute all modules except `intr_handler`.

External Reference Controls

ref=*module-set*
noref=*module-set*

These reference controls cause gcdm960 to assume/not assume that functions or data defined in the objects named by `module-set` are referenced outside the set of object files presented to the linker. You would normally use `ref` to keep the global decision making and optimization step from discarding modules that appear to be unused. The last `ref` or `noref` to name a given module applies. `noref` is typically used to exclude a subset of modules from a previous `ref`. The default is `noref`.

Inline Level Control

`inline=`*`n`*

This `gcdm` option argument controls how aggressively global inlining decisions are made. *n* defaults to `3`, and *n* must be less than or equal to `4`. The higher the argument, the more aggressive the inlining, and the larger your program may become. Note that inlining must be enabled (i.e., `+O5` control is used) for this control to have any effect.

Input Profile Control

`iprof=`*`file`*

This control causes the profile information in *`file`* to be incorporated into program-wide optimization decisions. *`file`* is a raw profile or a self-contained profile.

See Chapter 4, "Program-Wide Analysis and Optimization" for a discussion of profiles.

Fast Memory Controls

`sram=`*`hexstart,hexend[,hexstart,hexend]...`*

`m=`*`hexstart,hexlen[,hexstart,hexlen]...`*

The compilation system optimizes software to exploit on-chip cache and data RAM areas when you specify the architecture with the `-A` option. This optimization attempts to place the most heavily accessed data and variables in fast RAM. fast memory controls (gcdm option). The `gcdm` option lets you identify other SRAM areas that are available in a system.

Memory regions have an implicit order ranking with respect to the optimization tools; the left-most region specified is assumed to be the most desirable. Thus, the tools attempt to place the most heavily referenced variables into the first memory region specified. When that region is full, the tools begin placing variables into the second region specified.

For example, the control `m=Ox210,Ox3F0` places the most heavily referenced variables in an SRAM address beginning at `Ox210`. `Ox3F0` specifies the length of the memory range to be used for this purpose.

Using the `sram=Ox100,Ox3ff` control indicates to the system that the memory range `Ox100-Ox3ff` is available for data placement.

See your processor manual for information on memory region allocations.

Dryrun Control

```
dryrun
```

The `dryrun` argument echoes the commands that would be executed to implement all specified `subst` options into the report file, without actually doing the optimization work.

Report Controls

The `gcdm` option arguments listed here allow for creation of various optimization reports and creating and naming a report file.

dec=file

Causes the optimization decisions report to be sent to *`file`*, instead of to `stdout` (which is where reports appear by default).

dryrun

Echoes the commands that would be executed to implement all specified `subst` options into the report file, without actually doing the optimization work.

rsummary

Prints a summary of program-wide optimization decisions to the report file.

This is a typical `rsummary` report:

```
Initial linked text size was 20720 bytes.
About 21760 bytes are assumed available for the final
   text section.
0 variables were allocated to fast memory.
2 function call sites were inlined.
```

The first line shows the size of the application's text section before program-wide optimization.

The second line shows the decision maker's goal for the final size of the application's text section after program-wide optimization.

The third line shows that no variables were allocated into high-speed memory.

The fourth line shows that two call sites were inlined.

When the `-fvirtual-opt` option is supplied to the compiler, the summary also includes the total number of virtual function calls and the number of virtual function calls that have been resolved.

rdecisions

Creates a report that includes the initial and goal text sizes as described above, as well as a list of variables allocated to fast memory, a list of the estimated sizes of all functions before and after program-wide optimization, and a list of inlined call sites.

The `Inlined arcs` section of the report lists call sites selected for inlining:

- The `Caller` field is the function containing the call site that is inlined.
- The `Callee` field is the function being called at the inline site.
- The `Site` field is a numbering of the call site in the calling function. The first call in the calling function is site 1, the next call is site 2, and so on. This field is useful for distinguishing between call sites when the a function makes multiple calls to the same function.
- The `Percent` field is the percent of all dynamic calls for which this call site is responsible.
- The `Height` field is the height in the call tree of the called function.

rcall-graph

Creates a call graph report showing the dynamic behavior of the program.

- The `Function Callee` field lists the arcs in the call-graph. The format is:

```
Func
  Callee1
  Callee2
  Callee3
  ...
```

In this listing *`Func`* is the calling function. *`Callee1`*, *`Callee2`*, and *`Callee3`* are the functions that are called from function *`Func`*. A *`?`* in the callee field indicates that this call site is a call through a pointer. In this case the compiler does not know what function is called from this call site.

- The `Site` field is the call site number of the call to this function. Each call site in a function is assigned a number starting with 1.
- The `Count` field has two meanings. When applied to a calling function the count is the number of times this function was called during all profiled executions. When applied to a called function the count is the number of times this particular function was called from this specific call site during all profiled executions.
- The `Percent` field is the percentage of the total number of profiled dynamic calls that this `Count` accounts for.
- The `Size` field is relevant only for called functions; the value shown is the number of intermediate language statements in the function before program-wide optimization.
- For callees, the `Reg` field indicates how many registers were needed to generate code for the function. For callers, the `Reg` field indicates how many registers were used across the particular call site.
- The `Inline` field is relevant only for called functions; a value of 0 indicates that a called function was never inlined, and a value of 1 indicates it was inlined one or more times.

NOTE. *Functions that were not instrumented appear in the call graph only if they are referenced by some function that was instrumented.*

rreverse

Prints a reversed call graph to the report file. This control changes the format of reports generated by the `rcall-graph` control. When you use `rreverse`, the call graph report lists all the sites where a function is called from, rather than listing the call sites of each function. In other words, rather than listing each caller followed by its callees, the report lists each callee followed by its callers.

rclosure

This control reports the transitive closure of all possible callee functions.

rprofile

Prints the profile counts for the basic blocks that were hit to the report file.

- The `Line#` field is the line number within the file.
- The `Block#` field is the basic block that corresponds to this line number.
- The `Times hit` field is the number of times that this line of code was executed.
- The `From` field indicates how the value in the `Times hit` field was obtained.

For values that were completely estimated by the decision maker, the field contains “guess.”

For values obtained from profiles that were not subject to interpolation, this field contains *`n`* `Raw inputs`, where `n` is the number of profile files used to obtain the value.

For values obtained from interpolated profiles, this field contains *`n`* `Stretched inputs`, where *`n`* is the number of profile files used to obtain the value.

rvariables

Lists the variables allocated to fast memory with `-m` or `-sram` to the report file.

- The `Variable` field is the name of the variable to be allocated to fast memory.

- The `Size` field is the size of the variable in bytes.
- The `Usage Count` field is the number of times this variable was accessed during execution of the program.
- The `Address` field is the variable's address in fast memory.

Module-set Specification

A module-set specification (used in substitution controls and external reference controls, described earlier in this chapter) selects a subset of zero or more modules from the set consisting of all eligible modules in the program. A module-set specification has the format:

`[ archive ] : module`

The following rules govern module-set selection.

1. The set of eligible modules are those linked into the program that were compiled with the `-fdb` compiler driver option (described in Chapter 2, "gcc960 Compiler Driver" and Chapter 3, "ic960 Compiler Driver").
2. When either of the characters `:` or `+` appears twice in succession, that character is quoted and the meaning is a single `:` or `+` character.

   When a module-set contains an unquoted `:` character, it is interpreted as a pair of regular expression strings in the style of the UNIX Bourne shell, with the string to the left of the `:` matching object file archives and the string to the right of the `:` matching individual object files. For example:

   — matches all eligible modules

   — matches only eligible modules not linked in from libraries

   — `a:b.o` matches `b.o` from library `a`, provided the module is eligible
3. When a module-set contains no unquoted `:` characters, it is assumed to be the name of a function or variable in the program. In this case, the module-set refers to the object file that contains the definition of the variable or the body of the function, provided the module containing the variable definition or function body is eligible.
4. When a module-set is empty (that is, no characters occur between the option and the `=` character) the module-set defaults to `:*`, which refers to all eligible modules in the program not linked in from libraries.

C Language Implementation 7

This chapter discusses the following topics:

- "Data Representation"
- "Calling Conventions"
- "Object Module Section Use"
- "Pragmas"
- "Language Extensions"
- "Inline Assembly Language"

Data Representation

This section describes the scalar and aggregate data types recognized by the compiler, the format and alignment of each type in memory, and the range of values each type can take. For information on ANSI C data types, see *C: A Reference Manual*.

The i960 processors use a little-endian byte ordering, such that the address of a 4-byte (32-bit) variable is the address of the low-order byte of the variable. The i960 Cx, Hx, and Jx processors also support big-endian addressable memory, such that the address of a 4-byte (32-bit) variable is the address of the high-order byte of the variable.

Scalars

A scalar data type holds a single value, such as the integer value 42 or the bit string `10011`. Table 7-1 lists scalar data types for the i960 processor.

Table 7-1 Scalar Data Type (Sheet 1 of 3)

Data Type	Size (bytes)	Format	Range
unsigned char	1	ordinal	0 to 255
[signed] char	1	2's-complement integer	-128 to 127
unsigned short	2	ordinal	0 to 65535
[signed] short	2	2's-complement integer	-32768 to 32767
unsigned int	4	ordinal	0 to 4,294,967,295
[signed] int	4	2's-complement integer	-2,147,483,648 to 2,147,483,647
unsigned long	4	ordinal	0 to 4,294,967,295
[signed] long	4	2's-complement integer	-2,147,483,648 to 2,147,483,647
unsigned long long	8	ordinal	0 to 18,446,744,073,709,551,615
[signed] long long	8	2's-complement integer	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	single-precision floating-point	1.17549435*10^{-38} to 3.40282347*10^{38} (approximate absolute value)
double	8	double-precision floating-point	2.2250738585072* 10^{-308} to 1.7976931348623* 10^{308} (approximate absolute value)
long double	16	extended-precision floating-point	3.362103143112094*10^{-4932} to 1.189731495357231*10^{4932} (approximate absolute value)

1. Bit fields occupy as many bits as you assign them, up to a word (4 bytes), and their length need not be a multiple of 8 bits (1 byte).
2. The enum data type is identical in size and range to char, short, or int data type, depending on the range of constants in the enum declaration.

Table 7-1 Scalar Data Type (Sheet 2 of 3)

Data Type	Size (bytes)	Format	Range
bit field (unsigned value)[1]	1 to 32 bits	ordinal	0 to 2^{size}-1 (Size is the number of bits in the bit field.)
bit field[1] (signed value)	1 to 32 bits	2's complement integer	-2^{size-1} to $2^{(size-1)}$-1 (Size is the number of bits in the bit field.)

[1] Bit fields occupy as many bits as you assign them, up to a word (4 bytes), and their length need not be a multiple of 8 bits (1 byte).

[2] The enum data type is identical in size and range to char, short, or int data type, depending on the range of constants in the enum declaration.

Table 7-1 Scalar Data Type (Sheet 3 of 3)

Data Type	Size (bytes)	Format	Range
pointer	4	address	-
enum[2]	1, 2, or 4	2's complement integer or ordinal	varies

1. Bit fields occupy as many bits as you assign them, up to a word (4 bytes), and their length need not be a multiple of 8 bits (1 byte).
2. The enum data type is identical in size and range to char, short, or int data type, depending on the range of constants in the enum declaration.

Compiler options (*e.g.*, gcc960's `f[no-]signed-char` or `f[no-]unsigned-char`; ic960's `Gcs` or `Gcu`) set the `char` declaration default to `signed char` or `unsigned char`. Wide characters (character constants prefixed with an `L`) are syntactically supported but semantically identical to other character constants. Note that with gcc960 `char` defaults to `unsigned`, whereas ic960 defaults to `unsigned`.

The approximate ranges of `float`, `double`, and `long double` data types appear in Table 7-1.

NOTE. *On architectures with an internal floating-point unit (80960KB/SB), the compiler uses 32-bit and 64-bit general registers for intermediate results when performing calculations with* `float` *and* `double` *data types. Therefore, the accuracy of these data types is limited to 32 bits and 64 bits, respectively. The compiler does use the internal floating-point registers (*`fp0-fp3`*) when performing calculations with* `long double` *data types, yielding IEEE-754 accuracies at the expense of execution speed and code size.*

The alignment of a scalar data type is equal to its size. Although the extended-precision floating-point representation of `long double` requires only 10 bytes (80 bits), the natural architectural alignment of `long double` is 16 bytes. Therefore, to accommodate the semantic requirements of the C `sizeof` operator, the size of a `long double` is 16 bytes.

The following scalar alignments apply to individual scalars and to scalars that are elements of an array or members of a structure or union:

`char`	is aligned on a 1-byte boundary.
`short`	is aligned on a 2-byte boundary.
`int`	is aligned on a 4-byte boundary.
`long long`	is aligned on a 8-byte boundary.
`pointer`	is aligned on a 4-byte boundary.
`float`	is aligned on a 4-byte boundary.
`double`	is aligned on an 8-byte boundary.
`long double`	is aligned on a 16-byte boundary.

Aggregates

An aggregate data type consists of one or more scalar data type objects. You can declare the following aggregate data types:

array	consists of one or more elements of a single data type placed in contiguous locations from first to last.
struct	is a structure containing one or more scalar or aggregate data types. The members are allocated in the order they appear in the definition but do not always occupy contiguous locations.
union	is a single location that can contain any of a specified set of scalar or aggregate data types.

Structure Alignment

The alignment of a structure affects how much space the structure occupies and how efficiently the processor can address the structure members. A compiler option (for gcc960, `mi960_align`; for ic960, `Gac`) allows selection of any of the following alignment options for structures:

Optimal natural alignment	is the default alignment. For structures smaller than 16 bytes, this alignment is the size of the structure rounded up to the nearest power of 2. The compiler aligns structures of 16 bytes or larger on 16-byte boundaries. Optimal natural alignment produces the most efficient code for assigning values to structures and for passing structures as arguments.
Backward-compatible natural alignment	aligns a structure according to the greatest alignment requirement of any member of the structure. This alignment provides higher data density than optimal natural alignment and produces code and data compatible with that generated by ic960 releases before Release 3.0.
ABI-compatible alignment	aligns a structure or union to the maximum of the following: the greatest alignment requirement of any members of the structure; or 2 if the structure's size is 2 and 4 if the structure's size is 3 or larger.
User-constrained alignment	aligns a structure according to any legal value you specify. A compiler option (for gcc960, `mi960_align`; for ic960, `Gac`) or `#pragma i960_align` allows specification of alignments of 1, 2, 4, 8, and 16. Alignments can also be specified using `#pragma align`, described in this chapter.

Structure alignment can result in unused space, called padding, between members of the structure and between the last member and the end of the space occupied by the structure. The padding at the end of the structure is called tail padding. Because of differences in padding under different alignments, changing the alignment can change both the size of the structure and the offsets of members relative to the beginning of the structure.

The offset of a structure member from the beginning of the structure is as follows:

- Under both forms of natural alignment, the offset of a structure member is a multiple of the member's natural alignment. For example, since a `short` aligns on a 2-byte boundary, the offset of a `short` member from the beginning of a structure is a multiple of 2 bytes.
- Under user-constrained alignment, the offset of a structure member is a multiple of the lesser of the member's alignment or the alignment constraint you specify.
- For example, in a 1-byte alignment (`noalign`), the offset of a `short` member is not necessarily even.

The rules for structure member natural alignment are:

Scalar types	align according to their natural architectural alignment. For example, a `short` data type aligns on a 2-byte boundary.
Array types	align according to the alignment of the array elements. For example, an array of `short` data type aligns on a 2-byte boundary.

Union types	align according to the greatest alignment requirement of any member of the union. In the example below, `un1` aligns on a 4-byte boundary since the alignment of `c`, the largest element, is 4:
	`union un1 {`
	`  short a;/* 2 bytes */`
	`  char b;/* 1 byte  */`
	`  int c;/* 4 bytes */`
	`};`

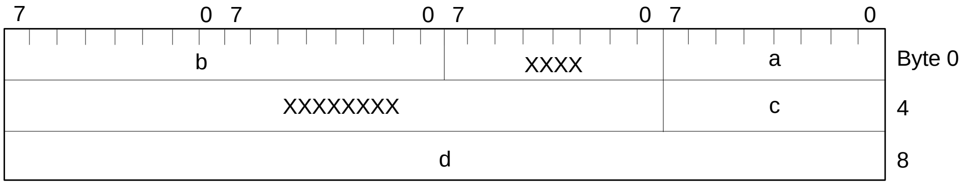
Structure types	align according to the alignment of the member types either natural or user-constrained.

Specifying optimal or backward-compatible natural alignment changes the size of a structure. Natural alignments differ only in tail padding. Member offsets, and therefore the padding between members, are the same under optimal natural alignment as under backward-compatible natural alignment. For example, the following structure occupies memory as shown in Figure 7-1 under either optimal or backward-compatible natural alignment:

```
struct strc1
{
   char  a;  /* occupies byte 0             */
   short b;  /* occupies bytes 2 and 3      */
   char  c;  /* occupies byte 4             */
   int   d;  /* occupies bytes 8 through 11 */
};
```

Under optimal natural alignment, the size and alignment of the `struct` type are both 16. Under backward-compatible natural alignment, the size is 12 and the alignment is 4.

Figure 7-1 Natural Alignment



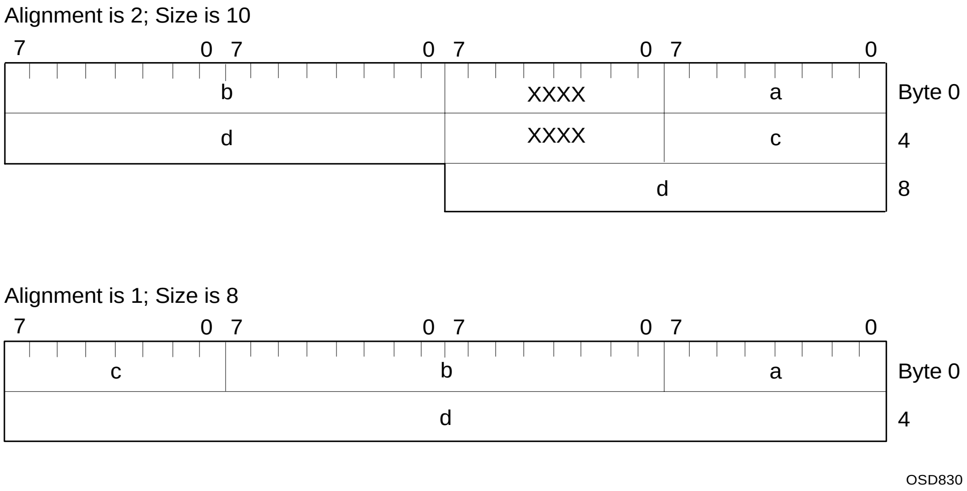
OSD829

Specifying a user-constrained alignment changes both the tail padding and the padding between structure members, which can also affect the structure size. A user-constrained alignment smaller than the natural alignment of a structure can result in a more tightly packed structure, saving space but slowing execution.

The example in Figure 7-2 compares the layouts in memory of the following structure under two different user-constrained alignments:

```
struct strc1  /*  Alignment is 2:   Alignment is 1:    */
  {          /*  ---------------   ---------------    */
    char  a;  /*  byte 0             byte 0            */
    short b;  /*  bytes 2 and 3      bytes 1 and 2     */
    char  c;  /*  byte 4             byte 5            */
    int   d;  /*  bytes 6 through 9  bytes 4 through 7 */
  };
```

Figure 7-2 User-constrained Alignment



A user-constrained alignment larger than the natural alignment aligns the structure on the natural-alignment boundaries. User-constrained alignment can increase the amount of tail padding relative to natural alignment but does not increase the padding between members of a structure. For example, specifying an alignment of `16` for `strc1` aligns the structure as in Figure 7-1.

When a `struct` has a member that is also a `struct`, the alignments of the member type and of the container need not be the same. For example:

```
struct NATURAL
{
     char  c1;
     short s;
     char  c2;
}
struct CONSTRAINED_1
{
     char  c;
     struct NATURAL  n;
}
```

If struct NATURAL has natural alignment, one byte of padding appears between the members c1 and s. Under optimal natural alignment, the size is 8 and the alignment is 8. Under backward compatible natural alignment, the size is 6 and the alignment is 2. If struct CONSTRAINED_1 has a user-constrained alignment of one, no padding appears between members c and n, nor does any padding follow the member n. However, all of the padding mentioned previously as part of struct NATURAL still appears in member n of struct CONSTRAINED_1.

Bit Field Alignment

Every bit field lies entirely within some bit-field container that has the same size and alignment as an int; that is, the container alignment is the smaller of 4 or a user specified alignment. A bit field can cross byte boundaries but cannot cross a container boundary.

Alignment of an individual bit field is necessary when the bit field, unaligned, overruns the end of the container in which it starts. A bit-field size of zero also forces bit-field alignment. The alignment of a bit field and the position of the bit field within a structure are determined as follows:

- The byte position of a bit field within a container is the current byte offset in the structure modulo the container alignment. This value is the byte offset relative to the most recent container alignment boundary. For example, if the container alignment is 1, the byte position is always zero. If the container alignment is 4, the byte position can be 0, 1, 2, or 3.
- The bit position of the bit field is the number of bits already allocated in the current byte, plus eight times the container byte position. This value is the bit offset, in the range 0 to 31, relative to the most recent container alignment boundary.

- If the value of the container bit position plus the size in bits of the new bit field is greater than 32 or if the size of the new bit field is zero, the compiler inserts padding to align the bit field on the next container alignment boundary. Bit-field alignment can result in padding of up to 31 bits. If the bit-field size is non-zero and the bit field fits entirely within the current container, the compiler does not insert padding to align the bit field.
- For big-endian, the bit position within the container is 31 minus the above-calculated bit position.

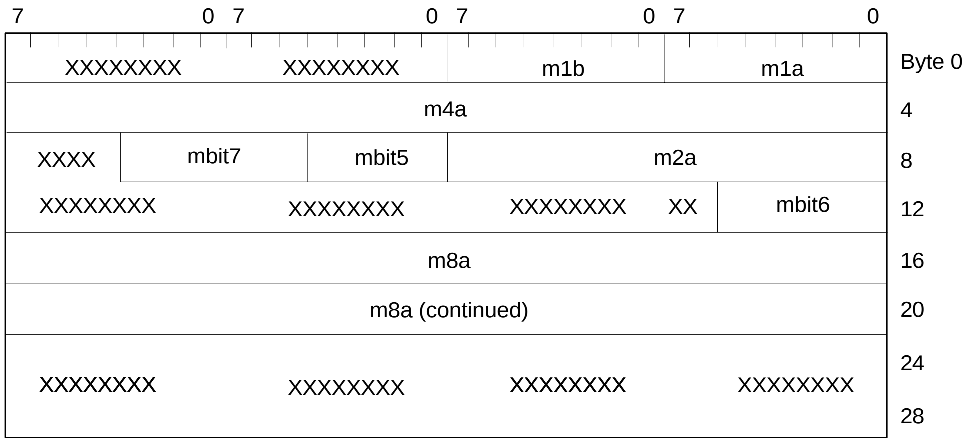
Examples

These examples show how different alignment pragmas alter the alignment of the components of a structure. The structure is declared as follows:

```
struct std_struct
   {
   unsigned char  m1a;
   unsigned char  m1b;
   int   m4a;
   unsigned short m2a;
   unsigned mbit5:5;
   unsigned mbit7:7;
   unsigned mbit6:6;
   double m8a;
   };
```

Figure 7-3 shows the optimal natural alignment of the structure, without any alignment pragma.

Figure 7-3 Optimal Natural Alignment of std_struct



OSD401

Figure 7-4 shows the backward-compatible natural alignment of the structure, without any alignment pragma but with the appropriate compiler option for backward compatibility specified (for gcc960, `mic-compat`; for ic960, `Gbc`).

Figure 7-4 Backward-compatible Natural Alignment of std_struct

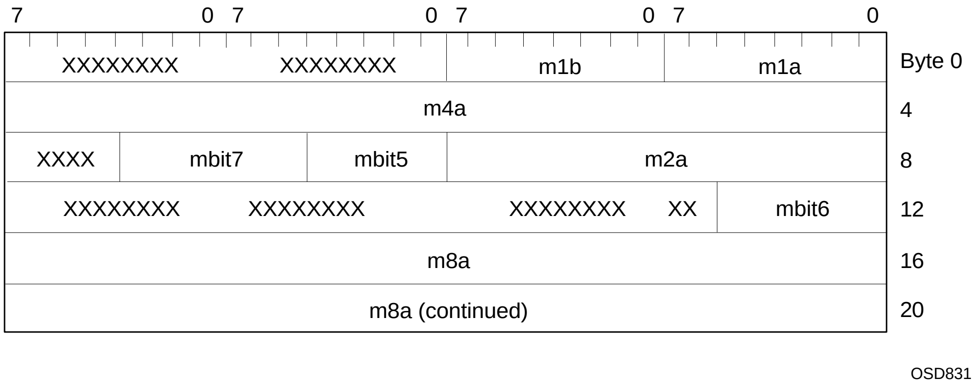


Figure 7-5 shows `std_struct` aligned on 1-byte boundaries, with the following alignment pragma:

```
#pragma noalign (std_struct)
```

Figure 7-5 #pragma noalign Alignment of std_struct

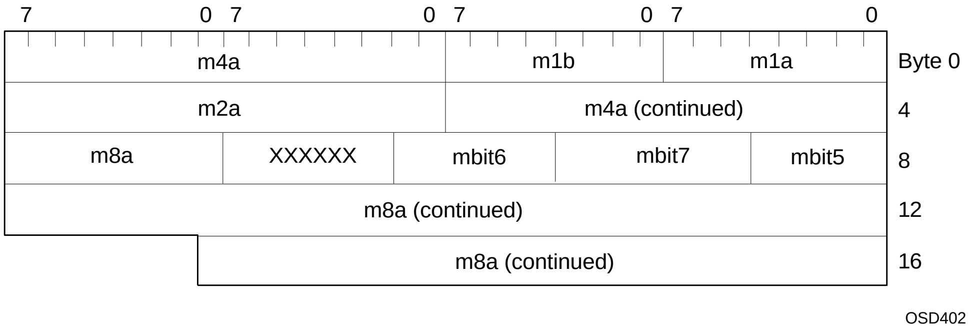
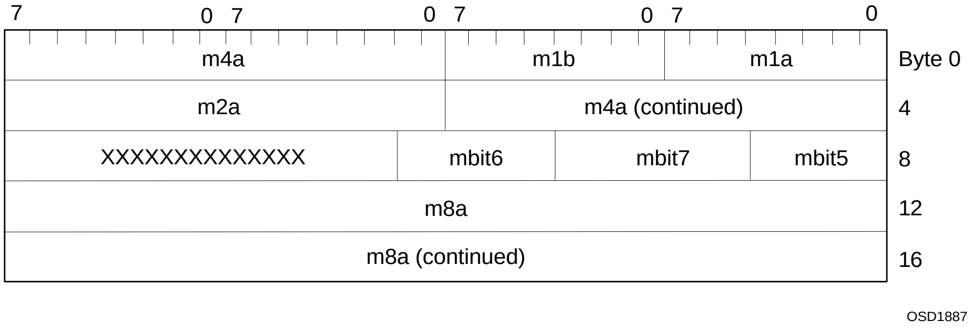


Figure 7-6 shows `std_struct`, aligned on 2-byte boundaries, as follows:

```
#pragma i960_align (std_struct = 2)
```

Figure 7-6 #pragma align Alignment of std_struct



Other Type Keywords

The `void` data type is neither a scalar nor an aggregate. Use `void` as the return type of a function, to indicate that the function does not return a value. Use `void *` as a pointer to an unspecified data type.

The `const` and `volatile` type qualifiers do not define data types. Rather, they associate attributes with other types. Use `const` to specify that an object is a constant and is not to be changed. Use `volatile` to specify that an object may change in ways unknown to the compiler. Optimization is inhibited on `volatile` objects. Inhibition of optimization is necessary for objects such as memory mapped I/O registers or data accessed by interrupt functions.

Calling Conventions

This section describes the standard i960 processor function calling convention and describes how the compiler generates code to conform to this calling convention.

The standard i960 processor calling convention places an absolute minimum overhead on simple, commonly called functions with few parameters. This is done by passing information between the calling function and the called function in the i960 architecture’s global registers as much as possible.

Definitions

call-preserved register
The register must have the same value upon exit from a function as it did upon entry to the function.

call-scratch register
The register may have a different value upon exit from a function than it did upon entry to the function.

The following list summarizes usage of the global registers `g0` through `g15` and the floating-point registers `fp0-fp3`.

`g0...g7`
These eight registers pass parameters into the called function from the calling function. If the return value of the function is four words or less in size, then the return value is passed back to the calling function in registers `g0` through `g3`. If the function returns a long double and generates code for the KB or SB processor and compatibility with ic960 R2.0 is requested, then registers `g0` through `g7` are call-scratch registers.

`g8...g11`
These four registers may be used for parameter passing in addition to `g0` through `g7`. If a parameter or a piece of a parameter is passed in one of these registers, that register is considered a call-scratch register. That register is considered a call-preserved register otherwise. If the called function can not be sure that a register has had a parameter passed in it, then the register must be considered a call-preserved register.

`g12`	`g12` is used as the PID bias register if generating code for position independent data (PID). `g12` is a call-preserved register.
`g13`	
`g13`	If the called function returns a struct or union larger than four words, then the calling function passes a pointer to the space allocated for the return value in `g13`. `g13` is a call-scratch register.
`g14`	If the function requires an argument block, this register contains a pointer to the argument block; otherwise it contains zero. If `g14` contains zero upon entry, then it must contain zero upon exit. If `g14` contains a pointer to an argument block upon function entry, then `g14` is considered a call-scratch register. `g14` may also be used to hold the return address when a function is called using a BAL instruction. In this case, `g14` must contain zero upon return from the function. This dual usage of `g14` means that a function that requires an argument block cannot be entered using a BAL instruction.
`g15`	`g15` is defined by the i960 architecture as the frame pointer (FP).
`fp0, fp1,`	If the function returns a long double and
`fp2, fp3`	generates code for the KB or SB processor and compatibility with ic960 R2.0 is requested, then `fp0` contains the return value of the function. `fp0-fp3` are considered call-scratch registers.
`AC`	The arithmetic control (`AC`) register is a call-scratch register. The condition codes are not preserved across a function call.

The 16 local registers (`r0` through `r15`) are 32-bit registers that provide a separate set of registers for each active function. Each time a function is called, the processor automatically sets up a new set of local registers for that function and saves the local registers for the calling function.

The particular use of each local register is:

`r0`	contains the previous frame pointer (`pfp`)
`r1`	contains the stack pointer (`sp`)
`r2`	contains the return instruction pointer (`rip`)
`r3...r15`	are general-purpose registers

Parameter Assignment to Registers

Parameters are passed in ascending-numbered registers, starting with `g0`, in the order the parameters appear (left-to-right) in the actual call. Both scalar and small aggregate (4 words or less) parameters are passed in registers.

The size of a parameter's data type determines the number of registers the parameter occupies. A parameter with a type size of one word or less occupies one register. A parameter with a type size of two words or less occupies two registers, and so on up to four words and four registers.

A parameter's type also determines in which register it must start. If the type's alignment is 4 bytes or less then the parameter may be passed starting in any register. If the type's alignment is 8 bytes then the parameter must be passed starting in an even numbered register. If the type's alignment is 16 bytes then the parameter must be passed starting in `g0`, `g4`, or `g8`. Any gaps left in the parameter registers due to alignment are not filled by following parameters.

Argument Blocks

An argument block is used to pass parameters when the parameters cannot be passed in registers. This can occur either because there are not enough registers left to pass the parameter, or when the parameter is too large (greater than 4 words) to pass in registers. As soon as a parameter is passed in an argument block, all further parameters get passed in the argument block. The calling function is responsible for the creation of an argument block if one is needed. When an argument block is created it must contain enough space at the beginning to store all the possible parameter registers `g0-g11`. Thus the first 48 bytes of an argument block are reserved for storing these registers. The first parameter passed in the argument block starts at an address 48 bytes above the base of the argument block.

Return Values

All return values four or fewer words in length are returned in registers `g0-g3`. For return values larger than four words the calling function must pass a pointer to a memory area to store the return value. This value is passed in register `g13`. The called function returns such a value by copying the value into the memory area pointed to by `g13`.

ic960 R4.5 implements a special return mechanism for functions returning long double, when generating code for ic960 R2.0 compatibility, and for a processor with on-chip floating-point support. In such a case the return value is returned in the `fp0` register.

Compiler Implementation

For compatibility with past implementations, the compiler allows some leniency in the implementation of the standard calling convention.

The compiler is more relaxed about the call-preserved status of `g8-g11` across a function call. At a function call, the compiler assumes that the called function may change `g8-g11` if any parameters are passed in an argument block, or if any parameters were passed in any of the registers `g8-g11`. However, the compiler properly implements the calling convention on the called function side, preserving `g8-g11` as necessary to satisfy the calling convention.

Object Module Section Use

The compiler generates assembly language that uses the following object file format sections to allocate storage for code and data:

`.text` The compiler places all assembly language instructions and constant data in the `.text` section. Constant data includes initialized variables with the `const` type qualifier, as well as string and floating-point literals.

`.data` The compiler places any initialized data in the `.data` section. Initialized data includes any statically allocated variables that you declare with an initialization list.

`.bss` The compiler locates uninitialized data in the `.bss` section as follows:

Uninitialized static variables go directly into `.bss`.

Uninitialized external variables are defined with the `.comm` directive. If the command line specifies the relaxed ref-def linkage (gcc960's `mno-strict-ref-def` option or ic960's `Gdc` option), the linker places these variables in `.data` if an initializing definition exists in another module. Otherwise, the linker places these variables in `.bss`. If the command line specifies strict ref-def linkage (gcc960's `mstrict-ref-def` option or ic960's `Gds` option), all uninitialized static variables are placed directly in the `.bss` section.

For more discussion of object module formats, refer to the *i960 Processor Software Utilities User's Guide*.

NOTE. *The compiler does not allocate storage in any section for variables declared as* `extern`*. Storage is allocated in the module defining the variable.*

Pragmas

Pragmas can supply implementation-defined information to the compiler. This section describes the supported pragmas in alphabetical order. For information about pragma syntax and pragmas in general, see *C: A Reference Manual*.

#pragma align [for gcc960 driver]

```
#pragma align n
```

n specifies the alignment value in bytes. Any of the following values are valid: `0`, `1`, `2`, `4`, `8`, `16`.

NOTE. *This pragma functions differently with the gcc960 and ic960 drivers.*

The `#pragma align` *n* feature sets the maximum formal alignment requirement for structs/unions to *n* bytes. *n* must be 0, 1, 2, 4, 8, or 16; other values are ignored. 0 instructs the compiler to revert to the maximum alignment in use before the last `#pragma align`. `n=16` is the default when `mic-compat` is not enabled; `n=1` is the default under `mic-compat`.

To get the alignment *a* for a struct or union *u*, given `#pragma align` *n*:

- let *m* be the largest alignment of all members of *u*.
- let *s* be *u*'s unpadded size rounded up to the next power of 2.
- then `align(u) = max (m, min (n, s))`.

Thus, a structure can never be given an alignment requirement that is less than the largest alignment required for any of its members; `#pragma align` can be used only to limit the amount of extra padding added to improve the alignment of the entire structure. Note that restricting structure alignment padding can affect the size and performance of the generated code.

The following examples show how `#pragma align` can affect the allocation of structs.

```
struct s0{          struct s1{          struct s2{
char x[9];          char x[8];          char y;
};                  struct s0 z;        short z;
                    };                  short zz;
                                        };
```

#pragma:	size of s0:	size of s1:	size of s2:
align 1	9	17	6
align 2	10	18	6
align 4	12	20	8
align 8	16	24	8
align 16	16	32	8

`#pragma align` does not restrict the alignment of individual `static`, `extern`, or `auto` variable allocations that happen to be structures. The compiler aligns each separate memory variable allocation based strictly on the size of the allocation, without regard to the formal alignment requirement of the variable's type.

#pragma align [for ic960, or for gcc960 with ic960 option]

```
#pragma align [ [(]size[)]]
#pragma align [(]identifier[=size][,...][)]
#pragma noalign [ [(]identifier[,...][)] ]
```

`size` specifies the alignment value in bytes. Any of the following values are valid: `1`, `2`, `4`, `8`, or `16`.

`identifier` specifies the structure tag used in `struct` type specifiers, as described in *C: A Reference Manual*.

NOTE. *This pragma functions differently with the gcc960 and ic960 drivers.*

Specifies alignment values for structures and unions.

Default

The default is optimal natural alignment.

Discussion

Use `#pragma align` to align structure members using the natural alignment value or a specified alignment size. Use `#pragma noalign` to specify byte alignment only. `#pragma noalign` is equivalent to `#pragma align` with a size of `1`. The `align` and `noalign` pragmas specify alignment values for `struct` types.

The alignment pragma applies to the whole structure; you cannot specify differing alignments for individual structure members. If you do not specify *`size`*, the compiler uses natural alignment.

Since the scope of an alignment pragma is all subsequent source text, nesting declarative scopes does not affect an existing alignment. However, you can place an alignment pragma within a structure declaration, so that the pragma affects any subsequent nested or top-level structure declaration.

The compiler aligns a `struct` type at the opening brace that brackets the `struct` declaration list, according to the following rules:

Rule 1 If the `struct` type has a tag and the tag identifier has appeared in an alignment pragma, the alignment is specified by the most recent alignment pragma for the tag identifier.

Rule 2 If the `struct` type has no tag and the `struct` declaration list is nested within another `struct` declaration list, the alignment is the same as that of the immediately enclosing `struct` type.

Rule 3 For any other situation, the alignment is specified by the most recent alignment pragma with no tag identifier.

The compiler generates warnings for the following condition:

- When an alignment pragma redefines the alignment for a specific structure tag name:

  ```
  #pragma align xyz=4
  #pragma noalign xyz
  ```

Examples

The following examples show different ways nested structures can be aligned:

```
#pragma noalign (outer1) /* Both outer1 and inner1 are
*/
#pragma noalign (inner1) /* packed (aligned on         */
struct outer1 {          /* 1-byte boundaries).        */
   struct inner1 {
      short s1;
      char c1;
   } si1;
   int i2;
};

#pragma noalign (outer2)         /* outer2 is packed. */
struct outer2 {
 struct inner2 {/* Since the inner structure has a tag
*/
 short s2;     /* (inner2) but no alignment specified,*/
```

```
 char c2;       /* alignment of inner2 uses the default*/
   } si1;       /* alignment.  The short s2 aligns on  */

               /* 2-byte boundaries and is the largest*/
               /* member of inner2; thus the default  */
               /* alignment of inner2 is 2.          */
   int i2;
};

#pragma noalign (outer3)         /* outer3 is packed. */
struct outer3 {
 struct {   /* Since the inner structure has no tag, it*/
 short s;   /* is aligned the same as the immediately  */
 char c ;   /* enclosing structure, outer3.  Thus both */
 } si1;     /* structures are packed.                  */
   int i2;
};
```

The following example shows nested unnamed structure definitions and alignment pragmas:

```
#pragma align my_structure = 16
struct my_structure              /* 16-byte alignment */
   {
      char f1;
      struct                     /* 16-byte alignment */
      {
      int ff2;
      } f2;
   };
#pragma align my_structure2 = 16
struct my_structure2             /* 16-byte alignment */
   {
      char f1;
#pragma align 4
      struct                     /* 16-byte alignment */
         {
         int ff2;
         } f2;
   };
/* If no more alignment pragmas appear, any subsequent
 * structs have 4-byte alignment.
 */
```

The following example shows alignment of a structure using the structure tag *identifier*:

```
#pragma align my_structure
struct my_structure              /* natural alignment */
   {
      char f1;
   };
#pragma noalign my_structure2
struct my_structure2            /* no alignment; i.e. */
   {                            /* 1-byte alignment   */
      char f1;
    };
#pragma align my_structure3 = 16
struct my_structure3             /* 16-byte alignment */
   {
      char f1;
    };
```

The following example shows alignment of structures without *identifier* specification:

```
#pragma align
struct my_structure              /* natural alignment */
   {
      char f1;
   };
#pragma noalign
struct my_structure2              /* no alignment */
   {
      char f1;
   };
#pragma align 16
struct my_structure3             /* 16-byte alignment */
   {
      char f1;
   };
```

#pragma cave

`#pragma cave [ [(] *function* [...] [)] ]`	
function	specifies function(s) for the compiler to prepare for compression. If no function is specified, the pragma applies to all functions defined following the pragma.

Prepares code for link-time compression and runtime decompression.

Default

The compiler does not prepare code for compression.

Overview

Compression assisted virtual execution (CAVE) reduces the physical memory requirements of ROM-based applications through link-time compression and on-demand runtime decompression of user-specified functions. The compiler, linker, runtime dispatcher, and compression and decompression routines cooperate to provide this feature. Code is typically compressed by a ratio between 1.5 and 1.7. Runtime decompression speed is about 30 clock cycles per byte of compressed code.

When the CAVE mechanism is used, either through `pragma cave` or the corresponding compiler driver options, selected functions in the application are designated to be secondary functions. All other functions are termed primary functions. The primary set should contain performance-critical functions, which are not to be affected by the CAVE mechanisms; the secondary set is subject to compression. Secondary functions are compressed by the linker and reside in memory in compressed form. At runtime, calls to secondary functions are intercepted by the CAVE dispatcher and the functions are decompressed if necessary.

Selecting Functions for Compression

The gcc960 `mcave` option, the ic960 `Gcave` option, or `#pragma cave` are used to designate the specified functions as secondary. You can use runtime profile information generated by gcov960 to aid in selecting the set of secondary functions.

Linking

The compiler places secondary function bodies within special CAVE sections (named `cave`) in each generated object file. The linker combines all input CAVE sections into one output CAVE section. Due to interdependencies between data or function addresses within compressed secondary functions and their compressed representations, address assignment must be done prior to compressing the secondary functions. As a result, a gap is formed between the compressed CAVE section and the section that follows, as shown below.

Before Linking		After Linking
.text section		.text section
uncompressed		compressed cave section
cave section		gap in memory
.data section		.data section
heap		heap
stack		stack

To utilize the compression savings the developer must use linker options or directives to position the CAVE section last in read-only memory.

Runtime Decompression

During program execution secondary functions reside in memory in compressed form. Every call to a secondary function is intercepted at runtime by a special dispatcher routine. The dispatch routine is contained in the `libc` library supplied with the tools. To ensure interception of all secondary functions, including invocations through indirect calls or interrupts, the compiler generates interceptor entries in the`.text` section, preceding the function bodies in the `cave` section as follows:

```
.section  .text
_foo:
  lda     L1,reg
  call    __dispatcher
  ret
.section  cave
  .word  L2-L1,0
L1:
  function body
L2:
```

Here the location `L1` of the secondary function body is passed to the dispatcher. The word preceding the function body is set by the assembler to indicate the uncompressed size.

The dispatcher performs the following steps:

1. Allocates a decompression buffer on the current runtime stack.
2. Saves the caller’s context.
3. Performs decompression.
4. Restores the caller’s context.
5. Invalidates the instruction cache.
6. Calls the decompressed secondary function.

The dispatcher prevents the runtime stack from being overrun by a long chain of recursive invocations by reusing the functions that are already active on the stack. The interceptor’s invocation of the dispatcher pushes a unique return address on the runtime stack. The return address is then used by the dispatcher to search the stack for the existing recursive activation. If found, the function is called immediately.

The dispatcher decompresses and executes secondary functions on the current runtime stack. Allocation and freeing of decompression memory is performed automatically through the call and return mechanism.

You must allocate more stack when using CAVE. The maximum additional runtime stack requirement is the total size of all secondary functions that may be active simultaneously.

Special Code Generation for Secondary Functions

When a decompressed secondary function is loaded on the runtime stack, its runtime location is different from the link-time one. Absolute intra-function and IP-relative inter-function references are invalid. These types of reference are not used during code generation for CAVE functions.

Since taking the address of a label is illegal in C, intra-function absolute references can be generated only in a jump-table implementation of the `switch` statement. Restricting the `switch` statement implementation in secondary functions to compare-and-branch instructions eliminates absolute intra-function references.

The IP-relative inter-function references are avoided in secondary functions by generating the 80960 `callx` instruction instead of the `call` instruction. The `callx` instruction transfers control to absolute rather than IP-relative locations.

Debugging CAVE Functions

CAVE functions are decompressed and executed on the runtime stack. The source-level debug information cannot be properly maintained in the current implementation. Consequently, secondary functions can be debugged only at the machine level. To debug:

1. Set a breakpoint on a CAVE function. Execution breaks on the first interceptor instruction (`lda L1`, `reg`).
2. Step into the dispatcher.
3. Display the disassembled instructions of the dispatcher.
4. The last two instructions in the dispatcher are:

   ```
   callx 80(r10)
   ret
   ```

5. `callx` is a call to a decompressed secondary function. Set a breakpoint on `callx` and step into the function.
6. Continue debugging the function on the machine level.

#pragma compress

```
#pragma compress   [ [(] function [,...] [)] ]
#pragma nocompress [ [(] function [,...] [)] ]
```

`function` specifies the function for the compiler to compress or not compress.

Controls the replacement of RISC instructions with CISC instructions.

Default

The compiler does not usually generate compressed (microcoded CISC) instructions, but the code produced may still use complex addressing modes for memory accesses. The compiler may generate single-line instructions (e.g., `cmpoble`) for two-line compare-and-branch instructions (e.g., `cmpo` and `ble`) but does not always do so.

Discussion

The `compress` and `nocompress` pragmas control the replacement of RISC instructions with CISC instructions.

If code size is of primary importance, use `compress` to replace RISC instructions with CISC instructions, thereby compressing the code size. Generated instructions use complex addressing modes. When `compress` is in effect, the compiler also generates single-line instructions for compare-and-branch instructions when possible.

Use `nocompress` to use RISC instructions, increasing the number of instructions but producing code that may run faster when instructions are found in the instruction cache. Generated instructions do not use complex addressing modes. Single-line instructions for compare-and-branch instructions are not generated.

In addition, `#pragma compress` disables some optimizations that increase code size greatly: automatic function inlining and loop unrolling.

If you do not specify *function*, the code compression pragma applies to all functions following the pragma. The compiler takes no action and issues no warning when the function name is specified but not found.

#pragma i960_align [for gcc960 and ic960]

```
#pragma i960_align [ [(]size[)]]
#pragma i960_align [(]identifier[=size][,...][)]
#pragma noi960_align [ [(]identifier[,...][)] ]
```

size specifies the alignment value in bytes. Any of the following values are valid: `1`, `2`, `4`, `8`, or `16`.

identifier specifies the structure tag used in `struct` type specifiers, as described in *C: A Reference Manual*.

Discussion

See the discussion of `pragma align` (for ic960, or for gcc960 with the `ic960` option).

#pragma inline

```
#pragma inline   [ [(] function [...] [)] ]
#pragma noinline [ [(] function [...] [)] ]
```

function specifies the function for the compiler to expand or not to expand inline. If no function is specified, the pragma applies to all functions defined following the pragma.

Controls replacement of a function call with the function body.

Default

The compiler does not replace the function call with the function's body. The `#pragma inline` has effect at optimization level 1 and higher. Chapter 11, "C Language Implementation" describes optimization levels in more detail.

Discussion

Use `#pragma inline` to replace a function call with the function body expanded at the place of the function call. Expanding a function inline increases the code size but decreases the execution time.

Note that a function that accepts a variable number of arguments cannot be expanded inline.

#pragma interrupt

```
#pragma interrupt [ [(] function [,...] [)] ]
#pragma nointerrupt [ [(] function [,...] [)] ]
```

`function` specifies the interrupt handler.

Specifies an interrupt handler.

Default

A function is not an interrupt handler.

Discussion

Use `#pragma interrupt` to declare a function as an interrupt handler. The `interrupt` pragma must precede the function definition. If no function is specified, the pragma applies to all functions defined following the pragma.

For interrupt handlers, the compiler tries to use global and floating-point registers only for a call. If the function uses any global or floating-point registers, the compiler preserves the registers. For any call, the compiler saves all registers except `g8` through `g11`. A register in the range `g8` through `g11` is saved only if it may be changed in the called function.

The compiler stores saved registers in contiguous locations, starting at offset `0x40` from the frame pointer, as follows:

- `g0` at `0x40(fp)`
- `g4` at `0x50(fp)`
- `g8` at `0x60(fp)`
- `fp` at `0x7c(fp)`

In processors with on-chip floating-point support, the compiler saves floating-point registers `fp0` through `fp3` starting at `0x80(fp)`.

An interrupt handler must not have parameters or return a value.

```
volatile int ready=0
   int poll()
   {
      while (!ready)
      ;
   } return ready;

#pragma interrupt(foo)
void foo(void)
{
   ready=1;
}
```

NOTE. *If an interrupt function accesses variables that are also accessed by the program, those variables should be declared* `volatile`*. If* `ready` *is not declared volatile, the optimizer may think that* `ready` *is always zero in function* `poll` *and may create an infinite loop by removing the test for (*`!ready`*).*

Note that `pragma interrupt` and `pragma isr` (described below) differ only in where the registers are saved. For `pragma interrupt`, the registers are saved at known offsets. For `pragma isr`, the compiler makes a context-specific choice of where to save the registers.

#pragma isr

Specifies routines to be compiled as interrupt service routines (`isr`'s). The syntax is:

```
#pragma isr [(] function_name [ [,] function_name
]...[)]
```

When a routine specified as an interrupt service routine is compiled, the compiler generates code so that registers `g0-g15` have the same values on exit that they had when entering the function. In addition, the code generated for the routine makes no assumptions about register `g14`'s value on entry. By guaranteeing these registers' values and not assuming `g14` to be zero, `#pragma isr` ensures that the routine's address can be placed directly in the interrupt vector table, and the state of the processor is the same at routine exit as it was at routine entry.

#pragma longcall

Specifies that a function should be called using the callx instruction

```
#pragma  [no]longcall  [( function  [,.])]
```

function identifies the function(s) to which the pragma applies. If the function is missing, then the pragama applies to all functions called in the compilation unit following the pragma.

Default

The compiler will use callx to invoke functions if the `mlong-calls` (or `Gxc` for ic960) compilation switch is used, otherwise the call instruction is used.

Discussion

The call instruction executes faster than the callx instruction. However, the target of a call instruction is limited to the range -2^{21} to $2^{21} - 1$ bytes in a call instruction. In other words you cannot use a call instruction to invoke a function that is located beyond this range. Using the longcall pragma for that function at the call site forces the compiler to use a callx instruction instead of the call.

The pragma longcall should be used at the call site; using the pragma longcall at the definition of a function will not cause a callx to be used at all sites where the function is invoked.

Pragma longcall overrides the -mlong-calls compiler switch.

With this pragma we can restrict the use of a callx instructions only to those call sites that need them.

#pragma optimize

```
#pragma optimize [(] [identifier =]"string"[,]
   [identifier = "string"]... [)]
```

Enables or disables optimizations. If specified, the identifier denotes a function with which the `#pragma optimize` string is to be associated. The string is a comma-separated list of optimizations to enable or disable. Currently recognized optimizations are:

`tce`	enable tail-call-elimination optimization
`notce`	disable tail-call-elimination optimization
`lp`	enable leaf-procedures optimization
`nolp`	disable leaf-procedures optimization

If no function is specified then this pragma applies to the rest of the file.

Any optimizations other than those recognized above are ignored.

#pragma pack

`#pragma pack` *n*

When used without an alignment pragma or option, this pragma has the same effect for both the gcc960 driver and the ic960 driver: it restricts the maximum alignment value that is honored for structure members to *n* bytes. A value of 0 tells the compiler to revert to the maximum field alignment in use before the last `#pragma pack`. Before the first `#pragma pack` is encountered, *n*`=16`.

NOTE. *The ic960 driver's* `pragma align` *and the gcc960 and ic960 drivers'* `pragma i960_align` *override* `pragma pack`. *The interaction of* `pragma pack` *and the gcc960 driver's* `pragma align` *is described below.*

Using #pragma pack with gcc960's #pragma align

When a member alignment requirement would exceed *n*, *n* is used instead — both for assigning the member's offset within its structure, and for determining the member's contribution to the structure's formal alignment requirement. It does not, however, restrict the overall formal alignment calculation for structures described for gcc960's `#pragma align`. To limit a structure's formal alignment requirement (presumably to limit extra padding at the end) you must use gcc960's `#pragma align` in addition to `#pragma pack`.

For example:

```
#pragma pack 2
struct s{
char a;
int  b;
};
```

`s.b` would be placed at offset 2 from the base of `s`; `sizeof(struct s)` would be 6 under gcc960's `mic-compat` (`#pragma align 1`) and 8 under default alignment (`#pragma align 16`). The formal alignment requirement of `struct s` would be 2 under `mic-compat` and 8 under default alignment.

The examples in the tables below all use the following sample structure:

```
typedef struct {
  char   m1;
  short  m2;
  double m3;
  char   m4;
  int    m5;
} s0;
```

Table 7-2 Example Offset Values

	Normal i960 Rules	**gcc960 Driver's #pragma pack 4**	**gcc960 Driver's #pragma pack 2**
offset_of(s0, m1)	0x0	0x0	0x0
offset_of(s0, m2)	0x2	0x2	0x2
offset_of(s0, m3)	0x8	0x4	0x4
offset_of(s0, m4)	0x10	0xc	0xc
offset_of(s0, m5)	0x14	0x10	0xe
sizeof(s0)	0x20	0x20	0x20
	#pragma pack 1	**#pragma pack 4 #pragma align 4**	**#pragma pack 2 #pragma align 2**
offset_of(s0, m1)	0x0	0x0	0x0
offset_of(s0, m2)	0x1	0x2	0x2
offset_of(s0, m3)	0x3	0x4	0x4
offset_of(s0, m4)	0xb	0xc	0xc
offset_of(s0, m5)	0xc	0x10	0xe
sizeof(s0)	0x10	0x14	0x12

#pragma pure

Specifies that a function has no effects other than returning a computed value and that it does so based solely on its input parameters.

`#pragma [no]pure [ ( `*`function`*` [,... ] ) ]`

`function` identifies the specific function to which the pragma applies. If *`function`* is missing, the effect of the pragma is applied to all functions called in the compilation module following the pragma. If a function name is specified, the pragma must be placed before the function definition.

Default

The compiler assumes functions are not pure and does not perform optimizations possible with pure functions.

Discussion

`pragma pure` informs the compiler that a named function has no effects other than returning a computed value and that it does so based solely on its input parameters. Specifically, the compiler assumes the following about the function:

- No I/O is performed.
- No global variables or memory locations are read or modified.
- No modifications of registers occur, except those explicitly defined by the calling sequence.

This knowledge enables the compiler to perform optimizations around function calls, optimizations it could not perform without this knowledge. If a function is “pure”, then the compiler can perform (around that function call) constant propagation, common subexpression elimination, global-variable migration, and dead-code elimination.

#pragma section

Allows COFF or ELF section naming.

```
#pragma section [ string ]
```

string	is alphanumeric characters a-z, A-Z, 0-9.

Discussion

This pragma causes all text, data and bss sections the compiler emits to be suffixed with `string`. For COFF the string must be three characters or less in length. For ELF, the string can be any length.

Using `#pragma section` without *string* sets the suffix back to null (the default).

This pragma is not supported for the b.out object format.

#pragma system

Specifies a system function.

```
#pragma system [ [(] function [=index] [,...] [)] ]
#pragma nosystem [ [(] function [=index] [,...] [)] ]
```

function	specifies the system function.
index	specifies the index into the system procedure table.

Discussion

If no function is specified, the pragma applies to all functions defined or called following the pragma. Use `pragma system` to specify a function to be called from the system procedure table. The compiler generates a `calljx` instruction for the system function call, which the linker replaces with the following:

```
lda  index, g13
calls g13
```

`index`	is the index of the system function in the system procedure table and is available to the linker through the symbol table entry for the function. This value must be in the range `0` to `259`.

For information on the `calljx` and `calls` instructions and the system function table, refer to the *i960 Processor Assembler User's Guide*.

You must associate a single system procedure table index with each system function before the final link of your program. The linker generates an error message for any system function that has no index or multiple conflicting indexes.

You can make this association in either or both of the following ways, if the defined index is consistent across all definitions:

- Specify `pragma system` at both the definition and the calling of the function. The compiler then generates the appropriate symbol table information, including the index.
- Use the`.sysproc` assembler directive to associate a system function name with an index.

Since register `g13` is used for the system function index, a system function cannot return a value larger than four words. Refer to the *i960 Processor Software Utilities User's Guide* for more information.

Language Extensions

GNU C provides several language features not found in ANSI standard C. (The `pedantic` option directs gcc960 to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro `__GNUC__`, which is automatically defined under gcc960 (but not under ic960).

Statements and Declarations Inside of Expressions

A compound statement in parentheses can appear inside an expression. This allows you to declare variables within an expression. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo()`.

This feature is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either `a` or `b` twice, with bad results if the operand has side effects. If you know the type of the operands (you can assume `int`), you can define the macro safely as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit field, or the initial value of a static variable.

Naming an Expression's Type

You can give a name to the type of an expression using a `typedef` declaration with an initializer. Here is how to define *`name`* as a type name for the type of *`exp`*:

```
typedef name = exp;
```

This is useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe “maximum” macro that operates on any arithmetic type:

```
#define max(a,b) \
  ({typedef _ta = (a), _tb = (b);  \
    _ta _a = (a); _tb _b = (b);    \
    _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`.

Referring to a Type with typeof

Another way to refer to the type of an expression is with `typeof`. The syntax of using of this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.

There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to `int`.

If you are writing a header file that must work when included in ANSI C programs, write `__typeof__` instead of `typeof`.

A `typeof` construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

- This declares `y` with the type of what `x` points to.:

  ```
  typeof (*x) y;
  ```
- This declares `y` as an array of such values:

  ```
  typeof (*x) y[4];
  ```
- This declares `y` as an array of pointers to characters:

  ```
  typeof (typeof (char *)[4]) y;
  ```

It is equivalent to the following traditional C declaration:

```
char *y[4];
```

To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, try rewriting it with these macros:

```
#define pointer(T)  typeof(T *)
#define array(T, N) typeof(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, `array (pointer (char), 4)` is the type of arrays of 4 pointers to `char`.

Generalized Lvalues

Compound expressions, conditional expressions and casts are allowed as lvalues provided their operands are lvalues. This means that you can take their addresses or store values into them.

For example, a compound expression can be assigned, provided the last expression in the sequence is an lvalue. These two expressions are equivalent:

```
(a, b) += 5
a, (b += 5)
```

Similarly, the address of the compound expression can be taken. These two expressions are equivalent:

```
&(a, b)
a, &b
```

A conditional expression is a valid lvalue if its type is not void and the true and false branches are both valid lvalues. For example, these two expressions are equivalent:

```
(a ? b : c) = 5
(a ? b = 5 : (c = 5))
```

A cast is a valid lvalue if its operand is valid. Taking the address of the cast is the same as taking the address without a cast, except for the type of the result. For example, these two expressions are equivalent (but the second may be valid when the type of `a` does not permit a cast to `int *`):

```
&(int *)a
(int **)&a
```

A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converted back to the specified type to become the value of the assignment. Thus, if `a` has type `char *`, the following two expressions are equivalent:

```
(int)a = 5
(int)(a = (char *)5)
```

An assignment-with-arithmetic operation such as `+=` applied to a cast performs the arithmetic using the type resulting from the cast, and then continues as in the previous case. Therefore, these two expressions are equivalent:

```
(int)a += 5
(int)(a = (char *) ((int)a + 5))
```

Conditional Expressions with Omitted Middle Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression:

```
x ? : y
```

has the value of `x` if that is nonzero; otherwise, the value of `y`.

This example is perfectly equivalent to:

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

Arrays of Length Zero

Zero-length arrays are allowed. They are very useful as the last element of a structure that is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};
{
  struct line *thisline
    = (struct line *) malloc \
    (sizeof (struct line) + this_length);
  thisline->length = this_length;
}
```

In standard C, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

Non-lvalue Arrays Can Have Subscripts

Subscripting is allowed on arrays that are not lvalues, even though the unary & operator is not. For example, this is valid though not valid in some other C dialects:

```
struct foo {int a[4];};
struct foo f();
bar (int index)
{
  return f().a[index];
}
```

Arithmetic on Pointers to void and Pointers to Functions

Addition and subtraction operations are supported on pointers to `void` and on pointers to functions. This is done by treating the size of a `void` or of a function as 1.

A consequence of this is that `sizeof` is also allowed on `void` and on function types, and returns 1.

The `Wpointer-arith` option requests a warning if these extensions are used.

Non-constant Initializers

The elements of an aggregate initializer for an automatic variable are not required to be constant expressions. Here is an example of an initializer with run-time varying elements:

```
foo (float f, float g)
{
  float beat_freqs[2] = { f-g, f+g };
  ...
}
```

Constructor Expressions

Constructor expressions are supported. A constructor looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer. The type must be a structure, union or array type.

Assume that `struct foo` and `structure` are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a `struct foo` with a constructor:

```
structure = ((struct foo) {x + y, ’a’, 0});
```

This is equivalent to writing the following:

```
{
  struct foo temp = {x + y, ’a’, 0};
  structure = temp;
}
```

You can also construct an array. If all the elements of the constructor are (made up of) simple constant expressions, suitable for use in initializers, then the constructor is a lvalue and can be coerced to a pointer to its first element, as shown here:

```
char **foo = (char *[]) { "x", "y", "z" };
```

Array constructors whose elements are not simple constants are not very useful because the constructor is not an lvalue. There are only two valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a `switch` statement, while the latter does the same thing an ordinary C initializer would do.

```
output = ((int[]) { 2, x, 28 }) [input];
```

Declaring Attributes of Functions

You can declare certain things about functions called in your program that help the compiler optimize function calls.

A few functions, such as `abort` and `exit`, cannot return. These functions should be declared `volatile`. For example:

```
extern volatile void abort ();
```

tells the compiler that it can assume that `abort` does not return. This makes slightly better code, but more importantly it helps avoid spurious warnings of uninitialized variables.

Many functions do not examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared `const`. For example:

```
extern const void square ();
```

says that the hypothetical function `square` is safe to call fewer times than the program says. A function should not be declared `const` unless:

- no I/O is performed.
- no non-local variables are read or modified either directly or via pointers passed into the function.

Inquiring about Alignment

The keyword __alignof__ allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like sizeof.

For example, the target machine requires a double value to be aligned on an 8-byte boundary, then __alignof__ (double) is 8. This is true on the i960 processor.

When the operand of __alignof__ is a lvalue rather than a type, the value is the largest alignment that the lvalue is known to have. It may have this alignment as a result of its data type, or because it is part of a structure and inherits alignment from that structure. For example, after this declaration:

```
struct foo { int x; char y; } foo1;
```

the value of __alignof__ (foo1.y) is 4, the same as __alignof__ (int), even though the data type of foo1.y does not itself demand any alignment.

Inline Functions Are as Fast as Macros

By declaring a function inline, you can direct the compiler to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included.

To declare a function inline, use the inline keyword in its declaration. For gcc960, use either inline or __inline. For ic960, use __inline. For example:

```
inline int
inc (int *a)
{
  (*a)++;
}
```

(If you are writing a header file to be included in ANSI C programs, write __inline__ instead of inline. See the Alternate Keywords section.)

You can also make all “simple enough” functions inline with the option `finline-functions`. Note that certain usages in a function definition can make it unsuitable for inline substitution.

When a function is `inline`, if all calls to the function are integrated into the callers, and the function’s address is never used, then the function’s own assembler code is never referenced. In this case, the compiler does not actually output assembler code for the function, unless you specify the option `fkeep-inline-functions`. If there is a nonintegrated call, then the function is compiled to assembler code as usual. The function must also be compiled as usual if the program refers to its address, because that reference can not be inlined.

Except when doing two-pass compilation, if an inline function is not `static`, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

If you specify both `inline` and `extern` in the function definition, then the definition is used only for inlining. In no case is the function compiled on its own, not even if you refer to its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it.

This combination of `inline` and `extern` has almost the effect of a macro. The way to use it is to put a function definition in a header file with these keywords, and put another copy of the definition (lacking `inline` and `extern`) in a library file. The definition in the header file causes most calls to the function to be inlined. If any uses of the function remain, they refer to the single copy in the library.

NOTE. *Function inlining occurs only at optimization level* `O1` *or higher. Inline functions are not inlined at* `O0`*. Inlining can be enabled with* `finline-functions` *at* `O1`*, and it occurs automatically at* `O2`*.*

Controlling Names Used in Assembly Code

You can specify the name to be used in the assembler code for a C function or variable by writing the `asm` (or `__asm__`) keyword after the declarator as follows:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable `foo` in the assembler code should be `myfoo` rather than the usual `_foo`.

On systems where an underscore is normally prepended to the name of a C function or variable, this feature allows you to define names for the linker that do not start with an underscore.

You cannot use `asm` in this way in a function definition; but you can get the same effect by writing a declaration for the function before its definition and putting `asm` there, like this:

```
extern func () asm ("FUNC");
func (x, y)
     int x, y;
...
```

It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols. Also, you must not use a register name; that would produce completely invalid assembler code.

Specifying Registers for Local Variables

You can define a local register variable with a specified register like this:

```
register int *foo asm ("r5");
```

`r5` is the name of the register that should be used.

Defining such a register variable does not reserve the register; it remains available for other uses in places where flow control determines the variable’s value is not live. However, excessive use of this feature may leave the compiler too few available registers to compile certain functions.

Alternate Keywords

The option `traditional` disables certain keywords; `ansi` disables certain others. This causes trouble when you want to use GNU C extensions, or ANSI C features, in a general-purpose header file that should be usable by all programs, including ANSI C programs and traditional ones. The keywords `asm`, `typeof` and `inline` cannot be used since they won’t work in a program compiled with `ansi`, while the keywords `const`, `volatile`, `signed`, `typeof` and `inline` won’t work in a program compiled with `traditional`.

The way to solve these problems is to put __ at the beginning and end of each problematical keyword. For example, use `__asm__` instead of `asm`, `__const__` instead of `const`, and `__inline__` instead of `inline`.

Other C compilers won’t accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:

```
#ifndef __GNUC__
#define __asm__ asm
#endif
```

Inline Assembly Language

Introduction

Two distinct styles of inline assembly language are supported by the compilation system: asm statements and asm functions. The recommended way to use inline assembly language is asm statements; asm functions are supported for compatibility with previous CTOOLS960 releases.

Resource Usage

The compiler makes assumptions about the machine resources: registers and memory. It manages access to these resources based on the C program, and its knowledge of the code it is generating, and inline assembly language can violate these assumptions.

Both styles of inline assembly language provide the programmer with ways to communicate the usage/modification of machine resources. Inline assembly code that uses/modifies such machine resources without informing the compiler may cause incorrect code to be generated by the compiler.

Before and after each call to a C function, the compiler generates instructions to preserve resources for the calling function while the called function executes. For example, any general purpose registers that might be updated by the called function must be saved on the stack before and after each function call. The term for this resource management is “the calling convention.”

The calling convention for a call to an `asm` function differs from that of a call to a C function. In particular, the compiler assumes by default that the only resources used by an `asm` function are its parameters, local temporaries, and the return value.   The compiler must be explicitly informed about other resources that can be used by the `asm` function. The compiler does not manipulate assembly language within `asm` functions. It relies on the assembler to check the assembly language. The result is that the compiler treats the body of an `asm` function as text. The compiler parses the text for symbolic names (parameters, local temporaries, and labels). However, the compiler does not recognize function calls, memory references, or explicit register usage within the `asm` function text.

asm Statements

You can use an `asm` statement to pass an assembler instruction through the compiler, and you can specify the instruction's operands using C expressions. Typically, `asm` is used to gain access to machine instructions that have no corresponding C paradigm.

`asm` statements are somewhat similar to function calls; both use parameter mechanisms to help describe the statements' inputs. In `asm` statements, however, an extensive mechanism is also provided for describing the asm's effects; the compiler can then assume that an asm has no effects or inputs that are not explicitly stated. In contrast, a function call is assumed to read or write all program variables unless proven otherwise. No such assumption is made for asm statements.

NOTE. *The compiler assumes that the inserted assembly instructions can only be executed immediately after the statement that precedes them, and that after the inserted assembly instructions have been executed, program execution resumes at the statement immediately following them.*

Syntax Examples

The following brief syntax examples are provided here for reference when studying the detailed grammar below. The effects and components of each specific example are discussed in detail in the Examples section below.

Example 1: sf1 (Simple)

```
asm volatile ("mov    0,sf1");
```

Example 2: sf1 (Complex)

```
asm volatile ("mov    sf1,%0; mov     %1,sf1":
       "=&d" (old_mask) : "dI" (new_mask));
```

Example 3: emul

```
asm("emul %1,%2,%0" : "=t" (temp) : "dI" (in1), "dI"
(in2));
```

Example 4: synmovq

```
__asm__ volatile ("synmovq %2,%3" : "=m"(*IAC_dst)
       : "m"(*IAC_p),"d"(IAC_dst),"d"(IAC_p)); }
```

Example 5: attadd

```
__asm__  __volatile__("atadd   %4,%2,%1" :
"=m"(*p),"=d"(wtmp)
   : "dI"(val),"m" (*p),"d"(p));
```

Example 6: modpc

```
__asm__  __volatile__("modpc   %1,%1,%0" : "=d"(new_pc)
   : "dI"(mask),"0"(new_pc)));
```

asm Statement Syntax

`asm` statements have the following syntax:

`asm [ volatile ] (` *`asm-template`* `[`*`asm-interface`*`] ) ;`

`asm-template` A C language ASCII string containing zero or more `substitution-directives`.

`substitution-directive%d` where no white space follows the `%`, and `d` is a decimal digit.

`asm-interface`

`:[out-list][:[in-list][:clobber-list ]]`

`out-list` `output-spec [,out-list]...`

`in-list` `input-spec [,in-list]...`

`clobber-list`	`clobber-spec [,clobber-list]...`
`output-spec`	`"=constraint" (C language object)`
`input-spec`	`"constraint" (C language expression)`
`clobber-spec`	`"regname"`

NOTE. *The keywords* `__asm` *and* `__volatile` *can be used in place of* `asm` *and* `volatile`.

asm Syntax Explanations

asm Keyword

asm statements begin with the keyword `asm`. Alternatively, the keyword `__asm` can be used to ensure ANSI C compliance.

volatile

If the optional keyword `volatile` is given, the asm is volatile. Two volatile asm statements are never moved past each other by optimizations, and a reference to a volatile variable is not moved relative to a volatile asm. The alternate keyword `__volatile` can be used to ensure ANSI C compliance.

asm-template

`asm-template`   A C language ASCII string containing zero or more *`substitution-directives`*.

The *`asm-template`* is a C language ASCII string that specifies how to output the assembly code for an instruction. Most of the template is a fixed string; everything but the *`substitution-directives`* (if there are any) is passed through to the assembler. Substitution directive syntax is explained below.

Generally, this fixed string is the body of the desired assembler instruction. This can be any instruction valid for the current i960 architecture.

NOTE. *The validity of the assembly code is not checked by the compiler.*

substitution-directive

`substitution-directive``%d` where no white space follows the `%`, and `d` is a decimal digit.

The character `%` occurring in the *`asm-template`* specifies where to substitute operands into the assembly instruction. The `%` followed by a digit *`n`* says to insert operand *`n`* at that point in the string. Operands are specified in the asm's *`output-specs`* and *`input-specs`*. Operands are numbered 0 through 9. No more than 10 operands can be specified.

asm-interface

`asm-interface` :[*`out-list`*][:[*`in-list`*][:*`clobber-list`*]]

The asm interface consists of three parts: an optional *`out-list`*, an optional *`in-list`*, and an optional *`clobber-list`*. These are separated by colon characters (`:`). See the preceding discussion of Resource Usage for background information on the *`asm-interface`* specification.

: (colon)

The colon (:) character is used to separate the *`out-list`* and *`in-list`*. Another colon is used to separate the *`clobber-list`* if one is used. If the *`out-list`* is missing, but an *`in-list`* is given, the input list must be preceded by two colons (`::`) to take the place of the missing *`out-list`*.

out-list

`out-list`   *`output-spec`* `[`*`,out-list`*`]...`

An *`out-list`* consists of one or more *`output-specs`* separated by commas. For the purposes of substitution in the *`asm-template`*, each *`output-spec`* is numbered. The first operand in the *`out-list`* is numbered 0, the second is 1, and so on. Numbering is continuous through the *`out-list`*, and into the *`in-list`*. The total number of operands is limited to 10 (i.e., 0-9). See *`substitution-directives`* above.

in-list

`in-list` *`input-spec`* [*`,in-list`*]*`...`*

Similar to an *`out-list`*, an *`in-list`* consists of one or more `input-specs` separated by commas. For the purposes of substitution in the *`asm-template`*, each *`input-spec`* is numbered, with the numbers continuing from those in the *`out-list`*.

clobber-list

`clobber-list` *`clobber-spec`* [,*`clobber-list`*]*`...`*

A *`clobber-list`* tells the compiler that the asm uses or changes a real machine register that is either coded directly into the asm or is changed implicitly by the assembly instruction. The `clobber-list` is a comma-separated list of `clobber-specs`.

output-spec

output-spec "=*constraint*" (*C language object*)

The *`output-specs`* tell the compiler about objects whose values can be written by the inserted assembly instruction. In order to more fully describe the output effects of the asm, you can list *`output-specs`* that are not actually referenced in the *`asm-template`*. See the `synmovq` and `attadd` examples below for specific examples of this.

input-spec

input-spec "*constraint*" (*C language expression*)

The *`input-specs`* tell the compiler about expressions whose values may be needed by the inserted assembly instruction. In order to more fully describe the input requirements of the asm, you can list *`input-specs`* that are not actually referenced in the *`asm-template`*. See the `synmovq` and `attadd` examples below for examples of this.

clobber-spec

`clobber-spec` "*`regname`*"

Each *`clobber-spec`* specifies the name of a single machine register that is "clobbered."

Resources that cannot be clobbered are:

`fp`(the frame pointer)
`sp`(the stack pointer)
`r0`, `r1`, `r2`(reserved)
`g14`

C language object

This can be any assignable C language lvalue. Typically this is just a variable name. A *`C language object`* must be of a type that matches its corresponding *`constraint`*. A *`C language object`* used in an *`output-spec`* must be of a type such that it can be assigned into. Object types must be the same size that their *`constraints`* would match. For example, the C type `int` is 32 bits; so is a global register. This would cause no mismatch. An integer type would not match a quad-word, however. If the object type and *`constraint`* do not match, the compiler attempts to add code to fix the mismatch, but in general it is better practice to avoid mismatches in the first place.

C language expression

This can be any legal C language expression. As in a *`C language object`* above, a *`C language expression`* must match its corresponding *`constraint`*. Unlike a *`C language object`* used in *`output-specs`*, a *`C language expression`* used in *`input-specs`* does not need to be assignable.

constraint

Each *`C language object`* or *`C language expression`* can have an associated `constraint`. The *`constraint`* is a string that tells the compiler what its associated operand must look like in order for the *`asm-template`* to generate a legal assembly instruction.

A *constraint* consists of one or more of the characters listed below. The compiler generates code if necessary to make the *C language object* or *expression* match one of the *constraint* characters. The associated operand is an integer literal or a machine register or an assembly label that is put in place of a substitution directive.

In general, it is better to write the asm such that the compiler does not need to generate extra code to make a *constraint* match. An operand can contain an empty *constraint* string if it is not used in the *asm-template*.

The valid *constraint* characters are as follows:

`=`	Specifies that the operand is assigned into. All *output-spec constraints* must start with this character.
`&`	Unless an output operand uses the *& constraint*, the compiler may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. If the assembler code consists of more than one instruction, this assumption may be false. In this case, you should use the `&` *constraint* for each output operand that may not overlap an input.
`d`	Allows any local or global word register.
`r`	Allows any local or global word register.
`l`	Allows any local register (r3-r15).
`b`	Allows any global register (g0-g15).
`t`	Allows any two-word register.
`q`	Allows any quad-word register.
`f`	Allows any floating-point register fp0 through fp3. This *constraint* is only valid for the i960 KB and i960 SB processors and only then if the gcc960 *msoft-float* option is not used.
`m`	Allows any memory operand.

`I`	Allows a constant in the range 0 through 31. This is the allowable range for a literal value in most instructions for the i960 processor.
`n`	Allows a known 32-bit constant.
`i`	Allows a 32-bit constant including a constant address.
`G`	Allows a floating-point constant of 0.0.
`H`	Allows a floating-point constant of 1.0.
`F`	Allows a floating point constant with any value.
`0-9`	This is a matching *`constraint`*. An operand that matches operand `n` (`0-9`) is allowed. If used, this must be the only character in the *`constraint`*. The specified operand must be an *`output-spec`*, and the *`constraint`* in which the matching `constraint` appears must be an *`input-spec`*. The *`asm-template`* should not refer to this operand, only to the operand `n` specified. This constraint is often used to ensure that an input operand and an output operand are in the same register. Generally, this is unnecessary on the i960 architecture.

Detailed Examples

Example 1: sf1.c (Simple)

The following example refers to the short C program shown in Example 7-1 below. The asm instruction is shown in bold.

Example 7-1 sf1.c (Simple)

```
/* Clears interrupt mask in sf1 for i960 CA processor */
void clear_interrupt_mask()
{
   asm volatile ("mov 0,sf1");
}
```

Consider the line containing the asm:

```
asm volatile ("mov 0,sf1");
```

- `"mov 0,sf1"` is the *`asm-template`*. It contains no *`substitution-directives`*, and the asm has no *`out-list`* or *`in-list`*. It simply writes a zero into register `sf1`. If `sf1` contains all zeros, all interrupts except `nmi` are disabled.

Note that this asm can be coded without the input or output operands because it neither uses nor affects any object or resources that the compiler knows about.

Example 2: sf1.c (Complex)

The following example refers to the short C program shown in Example 7-2. The asm containing the `sf1` instruction is shown in bold.

Example 7-2 sf1.c (Complex)

```
/*
 * Changes interrupt mask, and returns old interrupt mask
 * for i960 CA microprocessor.  Illustrates & constraint.
 */
int change_interrupt_mask(int new_mask)
{
    int old_mask;
    asm volatile("mov sf1,%0; mov %1,sf1":
               "=&d" (old_mask) : "dI" (new_mask));
    return old_mask;
}
```

Consider the line containing the asm:

```
asm volatile("mov sf1,%0; mov %1,sf1":
               "=&d" (old_mask) : "dI" (new_mask));
```

- `"mov sf1,%0; mov %1,sf1"` is the *`asm-template`*. The *`asm-template`* actually contains two `mov` instructions. The first writes the contents of register `sf1` onto operand 0 (`old_mask`) and the second writes operand 1 (`new_mask`) into register `sf1`.
- `"=&d" (old_mask)` is the only *`output-spec`*. It is the first operand (operand 0). The `"=&d"` is the *`constraint`*. The `=` says that this operand must be assignable. The `&` tells gcc960 not to allocate this output in the same register as an input operand. This is necessary because the first `mov` creates output before the second `mov` has used its input. The `d` indicates that this operand must go in a word register. If `old_mask` is not a word register, the compiler will generates code following the asm to copy the word register it chose for this output operand into `old_mask`.

- "dI" (new_mask) is the only *input-spec*. It is operand 1. The "dI" *constraint* indicates that operand 1 must be in a word register, or be a constant from 0 to 31. The compiler generates extra code as necessary to make sure new_mask matches one of the *constraints* before the asm is generated.

Example 3: emul.c

The example refers to Example 7-3 below. The asm containing the emul instruction is shown in bold.

Example 7-3 emul.c

```
typedef struct
{
   unsigned int lo32;
   int hi32;
}  int64;

typedef int int32;

static inline
int64 asm_emul(int32 in1, int32 in2)
{
   int64 temp;
   asm("emul %1,%2,%0": "=t" (temp)
                    : "dI" (in1), "dI" (in2));
   return temp;
}
int32 mul32_check_overflow(int32 a, int32 b)
{
   int64 t;
   t = asm_emul(a, b);
   if ((t.lo32 & 0x80000000) != 0)
   {
    if (t.hi32 == -1) /*upper32 matches lower32 sign bit*/
         return t.lo32;
   }
   else
   {
         return t.lo32;
   }
   else
```

```
    {
       if (t.hi32 == 0) /*upper32 matches lower32 sign bit */
          return t.lo32;
    }
    overflow_error("32 bit multiply overflowed");
    return t.lo32;
}
```

Consider the line containing the asm:

```
asm("emul %1,%2,%0" : "=t" (temp) : "dI" (in1), "dI"
(in2));
```

- `"emul %1,%2,%0"` is the *`asm-template`*. The `emul` instruction takes three arguments: `src1`, `src2`, and `dst`. These values are provided by the *`out-list`* and *`in-list`*.
- `"=t" (temp)` is the only *`output-spec`*. It is the first operand, i.e., operand 0. The `"=t"` *`constraint`* indicates that this operand must go in a double word register in order for the *`asm-template`* to generate a legal instruction.
- `"dI" (in1)` is the first *`input-spec`*. It is operand 1. The `"dI"` *`constraint`* indicates that operand 1 must be in a word register, or be a constant from 0 to 31 for the *`asm-template`* to generate a legal instruction. The compiler generates the extra code as necessary to make sure the value of `in1` will matches one of the *`constraints`* before the asm is generated.
- `"dI" (in2)` is the second *`input-spec`*. It is operand 2. Again the `"dI"` *`constraint`* indicates that operand 1 must be in a word register, or be a constant from 0 to 31. As before, the compiler makes sure that the operand matches one of the *`constraints`* before generating the asm. In this example, `temp` is declared as a local variable, and its type (int64) has the necessary size (8 bytes) and alignment (8 bytes) to go into a two-word register. Similarly, `in1` and `in2` must match at least one of their *`constraints`* because their size and alignment is the same as that required for a value in a word register.

Compile this example using:

```
gcc960 -S -O2 emul.c
```

NOTE. *That no extra code is generated to set up operands for the* `emul asm`.

Example 4: synmovq.c

The following example refers to the short C program shown in Example 7-4 below. The asm containing the `synmovq` instruction is shown in bold.

Example 7-4 synmovq.c

```
struct IAC_record {
    unsigned short field2;
    unsigned char  field1;
    unsigned char  message_type;
    unsigned long  field3;
    unsigned long  field4;
    unsigned long  field5;
};
struct IAC_record Cent_IAC_Space = { 0,0x60,0x40,0,0,0 };
static __inline__ void
post_interrupt(struct IAC_record *IAC_p)
{
   struct IAC_record *IAC_dst = (struct IAC_record *)0xFF000010;
   __asm__ volatile ("synmovq %2,%3" : "=m"(*IAC_dst)
                 : "m"(*IAC_p),"d"(IAC_dst),"d"(IAC_p)); }
```

Consider the lines containing the asm:

```
__asm__ volatile ("synmovq %2,%3" : "=m"(*IAC_dst)
               : "m"(*IAC_p),"d"(IAC_dst),"d"(IAC_p));}
```

- `"synmovq %2,%3"` is the *`asm-template`*. `synmovq` writes four words into reserved memory on the i960 KB processor, and then sends a message to the i960 processor telling it to do a software interrupt. `synmovq` takes two arguments, *`src`* and *`dst`*, where *`src`* is the location to copy from, and *`dst`* is the location to copy to. These values are provided by the *`out-list`* and *`in-list`*.
- `"=m" (*IAC_dst)` is the only *`output-spec`*. It is the first operand, i.e., operand 0. The `"=m"` *`constraint`* indicates that any memory operand can be used.
- `"m" (*IAC_p)` is the first *`input-spec`*. It is the second operand, i.e., operand 1. Again, any memory operand can be used.
- `"d" (IAC_dst)` is the second *`input-spec`*. It is the third operand, i.e., operand 2. The `"d"` *`constraint`* indicates that any global or local word register or a constant from 0 to 31 may be used. This register is only read, not written, so it acts as its own input.
- `"d" (IAC_p)` is the third *`input-spec`*. It is the fourth operand, i.e., operand 3. Again, any global or word register may be used.

NOTE. *In this example, four operands were specified, although the `asm-template` required only two. The additional operands (in this instance, operands 0 and 1) tell the compiler about objects whose values may be changed by the asm statement or whose value the asm statement may need. In this case, the asm modifies memory, which may affect optimizations the compiler performs at runtime. The only actual output from the asm is the memory written.*

Example 5: atadd.c

The following example refers to the short C program shown in Example 7-5 below. The asm containing the `atadd` instruction is shown in bold.

Example 7-5   atadd.c

```
static inline
int atadd(p, val)
volatile int *p;
int val;
{
      int wtmp;
      __asm__ __volatile__("atadd   %4,%2,%1" : "=m"(*p),"=d"(wtmp)
                : "dI"(val),"m" (*p),"d"(p));
      return wtmp;
}
volatile int critical_var;
int other_var;
int add_crit()
{
      atadd(&critical_var, 1);
      if (atadd(&critical_var, 2) != 1)
            atadd(&other_var, 1);
}
```

Consider the lines containing the asm:

```
__asm__ __volatile__("atadd   %4,%2,%1" :
"=m"(*p),"=d"(wtmp)
                   : "dI"(val),"m" (*p),"d"(p));
```

- `"atadd %4,%2,%1"` is the *`asm-template`*. `atadd` adds to memory and locks the bus until it is finished. This feature is used by multi-processor systems. `atadd` takes three arguments. These values are provided by the *`out-list`* and *`in-list`*.
- `"=m" (*p)` is the first *`output-spec`*. It is the first operand, i.e., operand 0. The `"=m"` *`constraint`* indicates that any memory operand can be used.

- `"=d" (wtmp)` is the second *`output-spec`*. It is the second operand, i.e., operand 1. The `"d"` *`constraint`* indicates that any global or word register can be used.
- `"dI" (val)` is the first *`input-spec`*. It is the third operand, i.e., operand 2. The `"dI"` *`constraint`* indicates that any global or word register containing a constant in the range 0 through 31 can be used.
- `"m" (*p)` is the second *`input-spec`*. It is the fourth operand, i.e., operand 3. Again, any memory operand may be used.
- `"d" (p)` is the third *`input-spec`*. It is the fifth operand, i.e., operand 4. Again, any global or word register may be used.

NOTE. *Again, this example specifies five operands, though the `asm-template` requires only three. The additional operands tell the compiler about objects whose values may be changed by the asm statement or whose value the asm statement may need.*

Example 6: modpc.c

The following example refers to the short C program shown in Example 7-6 below. The asm containing the `modpc` instruction is shown in bold.

Example 7-6 modpc.c

```
extern inline unsigned
modpc (unsigned new_pc, unsigned mask)
{
      int wtmp;
      __asm__ __volatile__("modpc   %1,%1,%0" : "=d"(new_pc), :
              "dI"(mask),"0" (new_pc));
      return new_pc;
}
int
raise_priority int(n)
{
      unsigned cur_pc;
      cur_pc = modpc(0, 0);  /* just read the pc */
      if ((cur_pc & 0x2) != 0)
      {
        /* we’re in supervisor mode, so we can change it */
        unsigned priority = ((cur_pc >> 16) & 0x1f) + n;
        unsigned priority_mask = 0x1f << 16;
        if (priority > 31)
             priority = 31;
        cur_pc &= ~priority_mask;
        cur_pc |= priority << 16;
        modpc(cur_pc, priority_mask);
        return 1;
      }
      return 0;
}
```

Consider the lines containing the asm:

```
__asm__ __volatile__("modpc   %1,%1,%0" : "=d"(new_pc),
: "dI"(mask),"0" (new_pc));
```

- `"modpc  %1,%1,%0"` is the *`asm-template`*. The `modpc` instruction reads and modifies the i960 architecture's process control register. The instruction takes three arguments.
- `"=d"(new_pc)` is the only *`output-spec`*. It is the first operand, i.e., operand 0. The `"=d"` *`constraint`* indicates that this is an output operand, and that any global or local word register can be used.
- `"dI"(mask)` is the first *`input-spec`*. It is operand 1. The `"dI"` *`constraint`* indicates that the operand must be a word register, or be a constant in the range 0 through 31. Note that operand 1 is referenced twice in the *`asm-template`* because the `modpc` instruction requires the same input operand in two places.
- `"0" (new_pc)` is the second *`input-spec`*. It is operand 2. The `"0"` constraint indicates that this operand and operand 0 must be allocated to the same register. This is required because in the *`asm-template`* this register is both a source and a destination. Note that operand 2 is not referenced in the *`asm-template`*, but that the reference to operand 0 is also the use of operand 2 as specified by the `"0"` constraint.

Note that this example shows how the 0-9 constraint is used to match an input to an output operand when a src/dst operand is needed in an *`asm-template`*. This example also shows that input-only operands (such as `mask`) can be freely referenced multiple times in an *`asm-template`* without needing to be specified multiple times in the `in-list`.

If you are writing a header file that should be includable in ANSI C programs, use `__asm__` instead of `asm` and `__volatile` instead of `volatile`. See the Alternate Keywords section for more information.

asm Functions

An `asm` function definition is a special form of a prototyped function definition. The keyword `asm` preceding the return-type specifier identifies an `asm` function definition. An `asm` function definition can occur anywhere a C function definition can occur. However, the definition of an `asm` function must precede any call to it.

NOTE. *An* `asm` *statement or* `asm` *function should not issue an assembler directive that changes the object module section to something other than* `.text`*. The compiler assumes the* `asm` *statement leaves the assembler in the* `.asm` *section.*

When processing an `asm` function call, the compiler generates additional instructions for loading registers, for other operations needed to pass parameters, and for acceptance of a return value. A call to an `asm` function is not a true function call, however, because the compiler expands the assembly-language body of the function inline.

An `asm` function definition can contain one or more templates. The compiler selects a template for expansion based on the values and data types of arguments you specify and based on use of any return value in the function call. Use of any C expression as an argument to an `asm` function is legal.

Also, any of the following are legal within an `asm` function:

- trigraphs
- spliced lines (backslash-newline pairs)
- C-style comments (`/*. . .*/`)
- macros and preprocessor directives

asm Function Definition Syntax

The declaration syntax for `asm` functions and parameters is the same as standard C function syntax. The following is an informal definition of `asm` function syntax:

```
asm return-type name (parameter-declarations)
{
% control-line
  template
[. . .]
}
```

`return-type`	is the data type returned by the `asm` function.
`name`	is the identifier used to invoke the `asm` function.
`parameter-declarations`	defines the data types and names of the `asm` parameters.
`control-line`	introduces each *`template`*, defines the parameter and return value classes, and specifies any calling-convention or non-`asm` processing.
`template`	is zero or more lines of text for processing by the assembler.

The following restrictions apply to `asm` parameter lists:

- An `asm` function cannot be a `stdarg` function; that is, an `asm` parameter list cannot contain an ellipsis (`. . .`).
- Each declaration in an `asm` parameter list must include an identifier.
- The data type of any `asm` parameter cannot be larger than 16 bytes.
- The data type of an `asm` function return value cannot be larger than 16 bytes.

An `asm` function can contain zero or more expansion cases, each of which starts on a new line and consists of a control line (starting with `%`) followed by a template. A control line can contain zero or more controls and can be continued to the next line with a backslash immediately before the newline character. A control can be any of the following:

- a parameter-declaration list to specify return values or `asm` parameter classes.
- the `call` or `error` keyword to cause an action other than `asm` in-lining.
- a `label` declaration to declare a label local to the `asm` function.
- the `use` or `spillall` keyword to preserve registers and variables.
- the `pure` keyword to indicate that the `asm` function has no side effects.

An `asm` parameter declaration in a control line specifies the classes for any parameters or return value. The keyword `return` is a special parameter identifier, denoting the return value and specifying its class. A control line can also contain declarations for local temporary variables.

The *`template`* can be any text. The compiler performs some preprocessing on the template text, but the assembly-language syntax checking is done by the assembler.

Template Selection

When the compiler encounters a call to an `asm` function, the compiler selects a template for expansion by comparing the call context with each control line in the function definition. The call context includes:

- the category (value, data type, and location) of each argument in the call.
- a boolean that shows whether the function uses the returned value.

Selection Criteria and Coercion

If a control line contains an `error` or `call` control and no parameter declarations, the control line unconditionally matches any call.

If a control line contains any parameter declarations or does not contain an `error` or `call` control, the control line matches a call only when the argument categories match the parameter declarations in the control line.

If an `error` or `call` control line contains parameter declarations, the compiler generates the message or function call only if the parameter classes match the call context.

The compiler processes `asm` functions by doing the following:

- Checking the `asm` function for correct syntax and semantics. If any of the following control lines are present, the compiler reports an error:
  - — `error` or `call` with any other control (`spillall`, `use`, or `label`)
  - — an `error` control line with more than one line of template text
  - — a `call` control line with any template text
- Reporting an error, if an `error` or `call` control line without parameter declarations is not the last control line in the `asm` function definition.

- Ensuring that all control lines contain either parameter declarations or an unconditionally matching control by adding default parameter declarations for all parameters declared in the function prototype to any control line that does not already contain `error`, `call`, or parameter declarations. This action includes adding parameter declarations to a control line containing `spillall`, `use`, or `label` controls but no parameter declarations. Default parameter declarations use `tmpreg` class for `return` and `reglit` class for parameters.
- Ensuring that the last control line unconditionally matches any call context. Unless the final control line in the `asm` function definition contains nothing but an `error` or `call` control, the compiler adds a final control line containing a `call` control, as follows:

  ```
  %call function;
  ```
- The default *`function`* for a `call` control is an external function of the same name as the `asm` function. The last control line is the only one that unconditionally matches any call context.
- Comparing the call context to each control line, in sequence from beginning to end of the `asm` function definition. The compiler expands the template of the first control line that exactly matches the call context. Tables 7-3 and 7-4 show how the call context and parameter classes can match.
- If no control line exactly matches the call context, attempting to coerce the call context into one of the control lines, starting at the end of the `asm` function and working back to the beginning.
  - A `ldconst` instruction coerces a constant argument into a register.
  - A `movr` instruction coerces a floating-point literal argument into a register.
  - A `ld` instruction coerces a memory argument into a register.
  - A `mov` instruction coerces a general-register argument into a temporary variable.
  - A `movr`, `movrl`, or `movre` instruction coerces an argument that is not a floating-point register or literal into a floating-point register and coerces an argument that is a floating-point register or literal into a general register.
- Expanding the last control line if no control line exactly matches the call context and the call context cannot be coerced into the last conditional control line.

Table 7-3 Return Value Class Matching[1]

Return Value Use	Return Class void	ftmpreg	tmpreg
not used	•	•	•
used	-	•	•

[1] A bullet (•) indicates a match. A hyphen (-) indicates no match.

Table 7-4 Argument Category to Parameter Class Matching and Coercion[1]

Argument	Parameter	Class			
Category	const	ftmpreg	freglit	tmpreg	reglit
Integer Constant (0-31)	•	ldconst, movr	ldconst, movr	ldconst	•
Other Integer Constant	•	ldconst, movr	ldconst, movr	ldconst	ldconst
Floating Constant (0.0 or 1.0)	•	movr	•	movr	movr
Other Floating Constant	-	ldconst, movr	ldconst, movr	ldconst	ldconst
General-register Variable	-	movr	movr	mov	•
Memory	-	ld, movr	ld, movr	ld	ld
General-register Temporary	-	movr	movr	•	•
Floating Register Temporary	-	•	•	movr	movr

[1] A bullet (•) indicates a match with no coercion needed. A hyphen (-) indicates no match and no coercion possible. A movr instruction for coercion indicates that movr, movrl, or movre can be used.

Parameter Classes

An `asm` parameter or return class can be any of the following:

`tmpreg`

places the parameter in a general-purpose register, of the compiler's choice, that the `asm` function can modify. For a `tmpreg` parameter longer than one word, specify the number of registers needed in parentheses after `tmpreg`.

For example, `tmpreg(3)` allocates three consecutive registers. If `tmpreg` is specified without a number of registers, the default is `tmpreg(1)`.

A `tmpreg` return value also occupies the specified number of registers. If no class is specified for `return`, the default is `tmpreg(`*n*`)`, where *n* is the size from `1` to `4` needed to contain the return value.

The maximum number of parameters that can be placed in registers is 10.

`ftmpreg`

places the parameter in a floating-point register, of the compiler's choice, that the `asm` function can modify. You can use `ftmpreg` only on processors with on-chip floating-point support. When used to declare `return`, `ftmpreg` places the return value in a floating-point register.

`reglit` places the parameter in a general-purpose register, of the compiler's choice. The `asm` function must not modify the register.

If the parameter is a literal, it can be used as is. Thus, the asm body should use the parameter only in an assembly language context that allows a literal.

For a `reglit` parameter longer than one word, specify the number of registers needed in parentheses after `reglit`. For example, `reglit(3)` allocates three consecutive registers. If `reglit` is specified without a number of registers, the default is `reglit(1)`.

A `reglit` return value also occupies the specified number of registers. The declaration `reglit return` is equivalent to `tmpreg return`.

`freglit` places the parameter in a floating-point register of the compiler's choice. The `asm` function must not modify the register. You can use `freglit` only on processors with on-chip floating-point support. When used to declare `return`, `freglit` places the return value in a floating-point register. The declaration `freglit return` is equivalent to `ftmpreg return`.

`const` indicates a constant expression. The `const` keyword can be followed by:

(*signed-integer*), specifying the indicated integer value.

(*signed-integer-low: signed-integer-high*), specifying an integer value in the indicated range.

`(0.0 : 1.0)`, specifying a floating-point value of `0.0` or `1.0`. Only use `const` to declare parameters, not `return`.

`void` indicates that the return value is not used. Use `void` to declare only `return`, not a parameter.

Declarations must be consistent between the `asm` function prototype and the control line. If the `asm` class of a parameter or return register does not match the declared C parameter or return type, the compiler issues a warning message. Table 7-5 lists the matching data types and classes.

Table 7-5 C Data Types and asm Classes

Class Designations	Data Types
reglit, tmpreg, reglit(1), tmpreg(1)	any integer type; any pointer type; float; struct, or union types of 1 to 4 bytes
reglit(2), tmpreg(2)	double; struct, or union types of 5 to 8 bytes
reglit(3), tmpreg(3)	long double; struct, or union types of 7 to 12 bytes
reglit(4), tmpreg(4)	struct or union types of 13 to 16 bytes
freglit, ftmpreg	float, double, or long double

NOTE. *Avoid writing a parameter declaration that can never match any call context. Such a declaration creates a pocket of unreachable code. For example, unreachable code results from declaring a parameter in an* `asm` *function prototype as an integer C type and declaring the corresponding parameter in the control line as* `ftmpreg` *or* `freglit` *class. The control line parameter declaration then matches only a floating-point data type argument, but the parameter can accept only an integer argument. Similarly, when specifying an integer return type in a function prototype, any* `return` *declaration specified in the control line must also be integer. The compiler recognizes when the parameters in the function prototype and the control line are mismatched and issues a message.*

Argument Categories

An argument category can be any of the following:

General-register variable	is a register-resident value (e.g., a `register` variable). Depending on the level of optimization, this category can include a more complex expression. During compilation, the expression must evaluate to a register-resident variable that is one of the operands in the expression. For example, the expressions `x+0` and `x*y/y` both evaluate to `x`.
Memory	is a memory-resident value.
General-register-temporary	indicates an expression that the compiler cannot evaluate to a single variable or constant. This category includes most expressions containing an operator. A common exception is an expression in which the top-level operator implies indirection (that is, `*`, `[ ]`, or `->`). Such an expression falls into the memory category. Depending on the optimization level, the general-register-temporary category can include an expression in which the top-level operator is an assignment to a register-resident variable. Floating-point values can also fall into this category.
Floating-point-register-temporary	indicates a floating-point expression that can be classified more efficiently into a floating-point register than into a general register. This category is available only on processors with on-chip floating-point support.

Integer constant	is a constant integer value. Depending on the level of optimization, this category can include an expression containing variable operands, if the compiler can evaluate the expression to a constant. For example, the expressions `x+5-x`, `x-7`, and `x` can evaluate to constants during compilation if the value of `x` is a known value at compile time.
Floating-point constant	is a constant floating-point value. The rules for classifying arguments as floating-point constants exactly parallel the rules for classifying arguments as integer constants.

Template Expansion

Once the compiler selects an expansion case, one of the following sequences occurs:

- If the control line contains the `error` control, the compiler reports an error, using the first line following the `error` control line as the text of the error message. For example, invoking the following `asm` function as `traps(1)` prints the message `Reached trap1`:

```
asm int traps (int i)
{
% const(1) i; error;
Reached trap1
% const(2) i; error;

Reached trap2
% error;
Reached traps without 1 or 2
}
```

- More than one line of template text following an `error` control line results in a compiler syntax error.
- If the control line contains the `call` control, the compiler generates a call to an external function using the `call` assembly-language instruction. You can specify the name of the external function, as in the following example:

```
%call my_alt_afn;
```

- If you do not specify a name in the `call` control, the compiler uses the name of the `asm` function. For example, calling the following as `select(3)` results in a call to an external function named `select`:

  ```
  asm int select (int i)
  {
  % const(-2:2) i;
  . . .
  % call select;
  }
  ```
- Any lines of template text following a `call` control line result in a compiler error.
- If the control line does not contain `call` or `error`, the compiler inserts the selected template in the assembly-language output in place of the `asm` function call.

Declarations

The control line can declare the following:

- parameters, including return
- local temporary variables
- labels

Parameter declarations and local temporary declarations are syntactically identical. If the declared name is the same as a parameter declared in the function prototype, a parameter is declared. Otherwise, the declaration is of a local temporary variable.

In the template text, the compiler replaces the name of any declared parameter with the corresponding register or literal argument. The `return` keyword becomes the name of the register in which the return value of the `asm` function is expected. The compiler replaces the name of any local variable with the name of an available register.

You can use a `reglit` or `tmpreg` class parameter or local variable as an integer aggregate containing up to four general-purpose registers, as declared on the control line. To select a register, specify an integer in parentheses after the identifier. For example, `itmp(0)` selects the first register of `itmp`. If `itmp` is declared on the control line as `itmp(4)`, specify `itmp(3)` to select the fourth register allocated for `itmp`.

NOTE. *If a template uses a label, multiple expansions of that template can result in more than one label with the same name, causing ambiguous branch or jump destinations. To avoid this ambiguity, use the* `label` *control to declare the label in the control line. The compiler then generates a unique name for each declared label every time the expansion case is selected.*

Preserving Register and Memory Values

The following asm controls enable the compiler to preserve function resource requirements:

`use` declares that certain registers can be read and/or modified by the template. You can specify any of registers `g0` through `g13`, `r3` through `r15`, and `fp0` through `fp3`, when present, as arguments to the `use` control. For example, the following control line preserves registers `g5` through `g8`, `r3`, and `r11`:

```
% use g5, g6, g7, g8, r3, r11;
```

If any of the registers `pfp`, `sp`, `rip`, `g14`, or `fp` are specified in a `use` control, the compiler issues an error message.

`spillall` declares that some memory locations used outside of the `asm` function can be modified or used by the template. The compiler forces synchronization of load and store operations at the function call; that is, no load or store operation moves past the call of an `asm` function containing the `spillall` control.

`pure` declares that the named `asm` function has no effect other than returning a computed value. Specifically, no I/O is performed, no global variables or memory locations are read or modified, and no modifications of registers occur, except those explicitly defined by the calling sequence. When `pure` is used, the compiler can perform optimizations before and after each function call, because `pure` guarantees the `asm` function has no effect other than returning the computed value. If a function is `pure`, the compiler can perform additional optimizations across the function call.

NOTE. *If none of the above controls appear in text to direct preservation of resources, the compiler makes the following assumptions:*
- *The only registers used by an* `asm` *function are those implicitly assigned by the compiler for parameters, local temporaries, and the return value.*
- *The* `asm` *function does not reference any non-volatile memory locations.*
- *The* `asm` *function can have other side effects, such as performing I/O.*

Examples and Hints

You can define control lines in a sequence that selects the expansion case based on the strictest comparison first, relaxing the matching criteria as earlier expansion cases are rejected, as follows:

1. `const` and `void return` parameters.
2. `ftmpreg` and `freglit` parameters; for example, to match `long double` arguments.
3. `tmpreg` and `reglit` parameters; for example, to match `integer`, `float`, and `double` arguments.

Sequential Template Expansion. The following is a C language program that uses an `asm` function with two expansion templates:

```
#define status_reg 0xFE00FF00
asm int poll(void)
{
% void return; tmpreg t; spillall;
  ld   status_reg, t;   #first template
% reglit return; spillall;/* return the current status
*/
  ld   status_reg, return;   #second template
%   error;
}
#define DEVICE_READY 0x00000001
main()
{
extern void service_device();
  poll();      /*clear status bits*/
  while (1) {
      if (poll() & DEVICE_READY)
           service_device();
       }
}
```

The first call of `poll` does not use the return value and therefore matches the `void return` control line, expanding the first template. The second call uses the return value and therefore matches the `tmpreg return` control, expanding the second template.

In this example, loading the status register also clears the status, so the `poll` function can be used just to clear the status if the function return value is ignored. However, when the return value is ignored, the program must still allocate a register into which it can load the temporary value.

Compiling this program produces assembly language similar to the following:

```
_main:
      ld  0xFE00FF00 , g4;  #first template
L5:
      ld  0xFE00FF00 , g4;  #second template
      bbc    0,g4,L5
      callj  _service_device
      b      L5
```

IAC Breakpoint. The following example shows an `asm` block that sends an inter-agent communication (IAC) breakpoint to the processor. For information on the IAC structure, see the i960 KB processor manual.

```
struct message {
    unsigned short  field2;
    unsigned char   field1;
    unsigned char   message_type;
    unsigned int    field3;
    unsigned int    field4;
    unsigned int    field5;
} iac_struct;
/*
 * This routine issues an IAC message to the local
 * processor where the program resides.  It accepts
 * a pointer to a preformed IAC message as input and
 * uses the synmovq instruction to send the IAC to the
 * processor.
 */
asm void send_iac(struct message *base_msg)
{

%void return; reglit base_msg; tmpreg myreg; spillall;
    lda 0xff000010, myreg    /* load local IAC address */
    synmovq myreg, base_msg       /* issue IAC message */
%error;
Incorrect C call to send_iac
}
/*
```

```
 * Send a breakpoint IAC to the processor.  The
 * address is supplied by the routine that calls
 * set_bp.  Do not forget to enable breakpoints in the
 * trace control.  Fields 1, 2, and 5 are not used.
 */
void set_bp(unsigned int addr1, unsigned int addr2)
{
iac_struct.message_type = 0x8f;
iac_struct.field3       = addr1;
iac_struct.field4       = addr2;
send_iac(&iac_struct);
}
```

In this example, the first line (`asm void send_iac(struct message *base_msg)`) declares that the function does not return a value and the `base_msg` argument is a pointer to a structure of type `message`.

The second and eighth lines contain braces. These lines begin and end the function definition, which contains two expansion definitions.

The third line is a control line containing three parameter declarations, as follows:

```
%void return; reglit base_msg; tmpreg myreg; spillall;
```

The `void return;` declares that no value is returned by this `asm` function. The `reglit base_msg;` declares that the `base_msg` parameter matches either a literal or a register argument. The `tmpreg myreg;` declares that the `myreg` local variable is a temporary register. The `spillall` control informs the optimizer that this template references memory.

The fourth and fifth lines load the IAC address into a temporary register and issue an IAC message.

The sixth line is a control line containing the `error` control and the seventh line is the text of the `error` message.

If the compiler cannot coerce the call arguments into the previous expansion definition (the declarations in the third line), the compiler displays the following error message and aborts the compilation:

```
Incorrect C call to send_iac
```

C++ Language Implementation 8

The C++ implementation is consistent with the C language implementation described in Chapter 7. This chapter highlights the differences from the C language implementation. It also provides a description of the unimplemented C++ features and description of the template implementation limitations.

Data Representation

The C++ compiler follows the same rules as described in Chapter 7, "C Language Implementation" for the format and alignment of various scalar and aggregate data types. The C++ compiler, however, recognizes the following scalar data types as well.

- `bool:` The `bool` type has the same size and alignment as an `int` and can be assigned a value of either true or false.
- `reference:` References are implemented internally as pointers. However, these implementation details are transparent to the end user and reference types in general should be treated the same as the type to which they refer.

Calling Conventions

The C++ compiler follows the same calling conventions as described in Chapter 7, "C Language Implementation". However, be aware that the compiler uses hidden parameters. Consider the following example:

```
class Base {
public:
 int set_a(int i) { a = i; }
private:
 int a;
};
```

The implementation of member function `set_a` uses a hidden parameter, the address of the Base instance for which this member function was invoked (the `this` argument). As a result, the user should expect argument `i` to be passed in register `g1`. Return values and register usage are handled the same way as described in Chapter 7, "C Language Implementation".

Pragmas

Pragmas can supply implementation-specific information to the compiler. The CTOOLS C++ compiler supports the same set of pragmas as the CTOOLS C compiler. However, certain pragmas behave differently in C++. The following sections highlight these differences.

Specifying a Tag-Name with align, noalign, or i960_align

When you specify a tag-name with align, noalign, i960_align, the pragma applies to all occurrences of that tag. For example:

```
# pragma align str=2

struct str {
 char c;
 struct str {
     char c;
 } s1;
};
```

```
struct str1 {
 char c;
 struct str {
     char c;
 } s1;
};
```

In the above example, the align pragma affects the alignment of types str, str::str, and str1::str.

The compiler currently does not implement referring to a specific type-tag (e.g., through the use of scope resolution operator) in a pragma:

```
# pragma align str1::str=2     // will not work
```

Specifying a Function Name with a Pragma

When you specify a function name with a pragma (e.g., pragma compress, cave, inline, interrupt, isr), the pragma applies to all occurrences of that name.

```
# pragma inline max

int max(int a, int b);
float max(float a, float b);

struct S {
 int a, b;
 int max(int a, int b);
};
```

In the above example, the inline pragma affects `max(int, int)`, `max(float, float)`, and `S::max(int, int)`

The compiler does not allow specifying a single instance of a function name in a pragma. For example, the statement:

```
# pragma inline S::max(int, int)
```

is not supported.

Link Time Considerations

The compiler creates two new `.text` sections named `ctors` and `dtors`.

- The `ctors` section is used to initialize(construct) static objects.
- The `dtors` section is used to destroy static objects

Starting with CTOOLS 6.0 a new set of linker directive files are included (`.ld` files) for use with the ic960 driver. These new ld files place the `ctors` and `dtors` sections immediately after the `.text` section. The C++ Iostream library is linked immediately before the C high-level libraries, as specified with the linker directive `PRE_HLL`.

With the gcc960 driver, use the `-stdlibcpp` option to link in the C++ Iostream library ahead of the C libraries and place the `ctors` and `dtors` sections immediately after the `.text` section.

Calling C Functions from C++

Use the `extern "C"` directive provided by the C++ language

```
// Example assumes that file1.cc and file2.c are linked together
// Begin file1.cc
extern "C" {
    int baz(int a, int b);   // Compiler does not do name
    void foo(void);          // mangling
};

int baz(float);              // Compiler treats this as a
                           // C++ routine and does name mangling

int baz(float f1)
{
 return int(f1);
}
```

```
int main()
{
 foo();                      // invokes the definition in file2.c
 return baz(10, 20)  +       // invokes the definition in file2.c
        baz(float(10.6));
}

// End file1.cc

/* Begin file2.c */

int baz(int a, int b)
{
 return a + b;
}

void foo(void) {
 baz(10, 20);
 return;
}

/* End file2.c */
```

Calling C++ Functions from C

Use the `extern "C"` directive provided by the C++ language.

```
// Example assumes that file3.cc and file4.c are linked together
// begin file3.cc
extern "C" int baz(void);

extern "C" {
int foo(int a, int b)
{
 return a + b;
}
}
```

```
int main()
{
 return baz();        // invokes the function defined in file4.c
}
// end file3.cc

/* begin file4.c */
int baz()
{
 return foo(10, 20); /* invokes function defined in file3.cc */
}
/* end file4.c */
```

asm Statements and asm Functions

The C++ compiler implements asm statements in a manner that is consistent with the C compiler. However, asm functions are not implemented in the C++ compiler.

Unimplemented C++ Language Features

The current release does not implement the following C++ language features:

Exception Handling

C++ provides constructs that allow exceptions to be raised and caught. The current release does not implement C++ exception handling. The following example illustrates the use of exception handling:

```
#include <iostream.h>

int main()
{
    int i;
    try {
     cout << "Enter an integer > 0 ";
     cin >> i;

     if (i <= 0)
         throw inv_data;
     ...
     ...
    }
    catch (Invalid_Data) {
     cout << "Invalid data input\n";
     exit(10);
    }
}
```

Run Time Type Information(RTTI)

C++ provides constructs that allow you to determine the type of an object during execution. This makes it possible to write specialized code based on the run-time type of the object. The current release does not implement RTTI. The following example illustrates the use of RTTI:

```
#include <typeinfo>
class B {
public:
    virtual int foo();
};

class D {
public:
    virtual int foo();
};

D d1;
B *bp = &d;

int baz(B *bp)
{
    if (typeid(*bp) == typeid(D))
```

```
                  do_something ...
                 else
                  do_other_stuff ...
             }
```

Namespaces

Namespaces allow a programmer to declare variable names without the fear of a collision with names declared by other users. Namespaces allow two independent library developers to use the same names for their library routines and allows the user to choose between the two. The following example illustrates the use of namespaces.

```
namespace A {
    int max(int a, int b)
    {
     int tmp;
     if (a > b)
         tmp = a;
     else
         tmp = b;
     return tmp;
    }
}

namespace B {
    int max(int a, int b)
    {
     return a > b ? a : b;
    }
}

int tmp;

using namespace A;
int main()
{
    tmp += max(10, 20);    // Calls namespace A’s max(int, int)
}
```

Debugging Information for Templates

Debugging information for templates is currently not supported.

GCC960/ic960 Compatibility 9

This chapter describes the incompatibilities between ic960 and gcc960, and between the current release of gcc960 and other releases of ic960.

char and short Parameters

The ic960 R3.0 compiler expects `char` and `short` parameters and return values to be clean upon entry to and exit from procedures. Since these types are passed and returned in registers, this means that, in the case of `signed` types, the sign bit must be extended, and in the case of `unsigned` types, the high-order bits of the register must be zero. By default, gcc960 (and ic960 R4.5 and later) does not expect these values to be clean, and generates appropriate operations to sign- or zero-extend these values on entry to or exit from a procedure. This applies only to ANSI-compliant programs that specify the type of parameters at declaration time in the function prototype.

gcc960 emulates ic960 R3.0's behavior if the `mic3.0-compat` or `mic2.0-compat` options (see below) are selected.

enum Variable Byte Count

The ic960 R3.0 compiler creates `enum` variables with only enough bytes of precision to hold the requested enumeration. gcc960 always generates 4-byte `enum` variables. gcc960 emulates ic960's behavior if the `mic3.0-compat` option is selected. An `enum` variable compatible with ic960 releases prior to R3.0 can be achieved using the `mic2.0-compat` option.

char Types

The ic960 compiler (all releases) treats default `char` types as `signed`, whereas gcc960 treats them as `unsigned`. gcc960 emulates ic960's behavior if the `mic3.0-compat` or `mic2.0-compat` options (see below) are selected, or if the `fsigned-char` option is selected. The preprocessor symbol `__CHAR_UNSIGNED__` is set appropriately to allow programs to determine which model is in use.

Identifying Architectures

The traditions for architecture-identifying preprocessor macro definitions are somewhat different between ic960 and gcc960. Both interfaces define the macros `__i960`, `__i960`*xx*, and `_i960`, where *xx* is the architecture (e.g., CA for the i960 CA processor, as selected by the `ACA` option). These are the recommended macros for testing for the i960 processor architecture.

For compatibility reasons, the compilation system also defines additional variations on these macros, as shown in Table 8-1.

Table 9-1 Architecture Macros and Compatibility

	gcc960	ic960
_ _i960_ _	X	-
_i960*xx*	-	X
_ _i960_*xx*_ _	X	-
_ _i960*xx*_ _	X	-

#pragma align

ic960 and gcc960 both implement a `#pragma align` directive. They interpret the pragma differently, and the results (changes in the alignment of members of structures) are not compatible. In the absence of this pragma, ic960 and gcc960 structures should be compatibly aligned. `pragma i960_align` is provided for compatibility with ic960's `pragma align`, and behaves the same for both compiler interfaces.

mic3.0-compat Option

The gcc960 `mic3.0-compat` option selects the appropriate behavior for `enum` variables, selects default `signed char` variables, and selects clean linkage (described above) for `char` and `short` parameters and return values.

mic2.0-compat Option

The gcc960 `mic2.0-compat` option selects the same behaviors as `mic3.0-compat`, except that the behavior for the `enum` variable is subtly different and the alignment rules for structure elements are changed to be compatible with this (now obsolete) release of ic960. The `mic-compat` option supported in gcc960 R1.2 and R1.2.1 is now synonymous with `mic2.0-compat`.

Position Independence and Reentrancy 10

This chapter describes reentrancy and position-independence. Use it for writing i960 processor applications that require position-independent or reentrant programs. Position independence enables relocation of both the `.text` and `.data` sections.

Position-independent Code and Data

Position independence refers to an application that can be relocated when loaded. The application can be loaded at various addresses, but the code and data do not move during execution. This feature enables creation of programs for specific EPROMs used in a system.

The ic960 driver's `G` option with its arguments `pc`, `pd` and `pr`, or the gcc960 driver's `mpic`, `mpid` and `mpid-safe` options, control generation of position-independent code and data. For more information about command-line options, see Chapter 2, "gcc960 Compiler Driver", and Chapter 3, "ic960 Compiler Driver".

Position-independent Data

When the position-independent data option is specified, references to variables in the program are made relative to `g12`. Initialization code for a program must supply a data address bias in the position-independent data bias register (`g12`). For all accesses to statically allocated variables, the value in `g12` is used to calculate the effective address. Register `g12` must be read-only for the entire program.

For example, suppose object _x is in the .data or the .bss section. Normally, the compiler generates an address of the object with an absolute addressing mode:

```
lda _x, g0
```

When you compile your program with position-independent data, the compiler generates this instruction to take the address of _x:

```
lda _x(g12), g0
```

NOTE. *If PID is specified, the value in* g12 *must be correctly computed and stored by user-provided startup code.*

Position-independent Code

When the position-independent code option is specified, the compiler computes effective addresses by biasing them based upon the instruction pointer (ip).

Suppose object _x is in the .text section. The compiler generates a code bias address into a register at the beginning of any function that needs a direct address in the .text section. It does this via a code sequence similar to this:

```
lda 0(ip), r3
lda .    , r4
subo       r4, r3, r3
```

which leaves the bias in r3. Then the compiler uses r3 to bias the reference to _x as:

```
lda _x (r3), r4
```

The first three instructions compute the difference between the link time address and load time address of the .text section.

For example, if the code section links to begin at address zero, the subtraction result is the address at which the code section was actually loaded. Even if the code section links to begin at some other address, the subtraction result is still the correct value for biasing pointers into the code section.

Example 10-1 Position-independent ROM Code

Imagine designing two circuit boards for use in a new laser printer. ROM chips on these boards contain type fonts and graphic elements. To provide alternative printing capabilities, either board inserts into an optional slot in the printer chassis. Memory allocated for each board is:

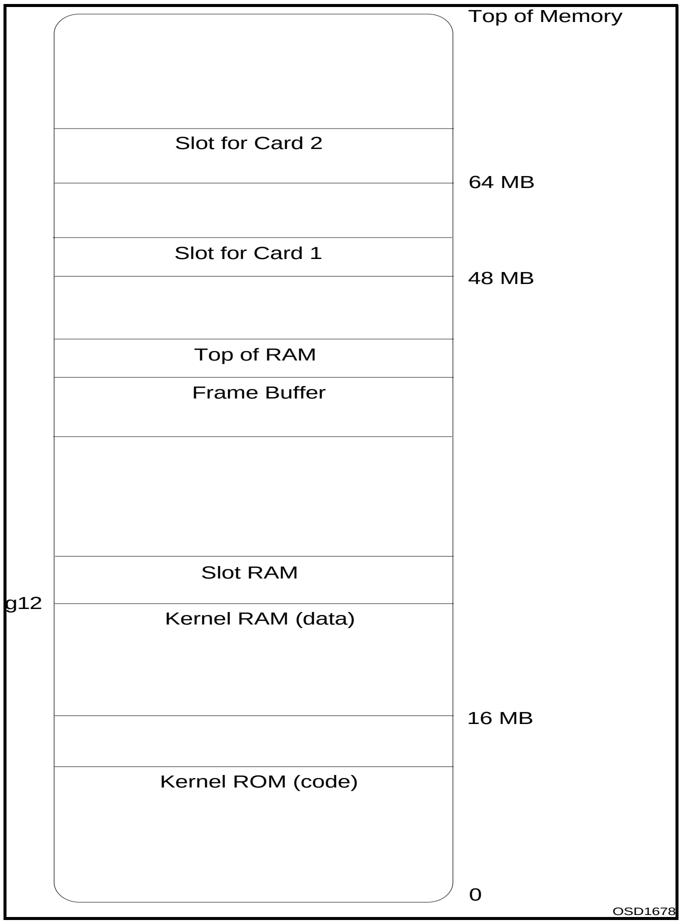
board 1	`20000 - 3ffff`
board 2	`40000 - 5ffff`

Although ROM and RAM for each board have different load addresses, the controlling software for the printer must work correctly with either board in use. In the printer, kernel ROM and RAM are at fixed addresses in low memory. A large memory space is set aside for the kernel's ROM and RAM.

Compiling the ROM code with the PID option and placing the correct bias values in `g12` makes the optional ROMs relocatable.

Figure 10-1 shows memory allocation for board 1. When the code executes, the ROM code for either board loads at the correct address.

Figure 10-1 Memory for Hypothetical Position-independent Application



Guidelines for Writing Relocatable Programs

A program can contain position-independent code (PIC), position-independent data (PID), or both. Be aware of the following restrictions:

- Use position-independence only where necessary, because a program containing position-independent code may execute more slowly than one without.
- Position-independent programs cannot be relocated during execution.

For all i960 processors, the address space is flat (unsegmented) and byte-addressable. Addresses run contiguously from 0 to 2^{32}-1. Programs can allocate space for data, instructions, and stack anywhere within the flat address space. However, the following restrictions apply:

- Instructions must be aligned on word boundaries.
- Addresses `FF000000H` through `FFFFFFFFH` in the upper 16 megabytes of the address space are reserved for specific functions. Check with your system hardware designer to determine the effects of use of the addresses in this range.
- On i960 Cx and Jx processors, the lower 1 kilobyte of address space (addresses `0000H` through `03FFH`) is reserved for accessing internal memory (RAM). On i960 Hx processors, the lower 2 KB is internal memory. Instruction fetch operations from this address range are not allowed.
- The `.data` and `.bss` sections must be relocated as a unit.

Because biasing occurs during code execution, the compiler does not support static initialization of pointers with the address of a position-independent object. The compiler generates a warning in these cases.

For example, the following program has two pointers, `p` and `g`, whose initial values might not be correct when position-independence is used.

```
static int i;
static int *p = &i;
static int *q = 0;
static int *r = (int *) 0x7fff0000;
int f();
int (*g) () = f;
```

In the compiler's output, `p` contains the unbiased address of `i`, and `g` contains the unbiased address of `f`. To use the initialized `p` or `g`, a program must perform the correct biasing of values before the point where the program uses the pointers.

Reentrant Functions

Reentrant functions can suspend execution, and later resume execution from the same state at which the suspension took place. Current state data must be preserved while a reentrant function is suspended.

A reentrant function can be active in several different places, in any of the following ways:

- a multi-tasking situation with two or more threads executing in the same memory space; for example, an interrupt handler
- a time-sliced environment in which two or more processes are executing, with one process active and all others suspended at any given time
- a recursive function, with any one instance of a function active while all duplicate instances of the function are suspended

For a function to be reentrant, it must not:

- modify memory or registers in use by a concurrent or suspended function
- reference shared variable data
- call a non-reentrant function

Designing Reentrant Functions

Since the compiler cannot determine data use across modules, the compiler does not issue any warnings for potentially non-reentrant code sequences. For more information about library reentrancy, refer to the *i960 Processor Library Supplement*.

Initializing the Execution Environment 11

This chapter describes the initialization process for the i960 processor execution environment, including startup assembly-language routine, configuration files, and associated options.

Startup Code

The startup routine is a module that initializes the processor and library, then invokes the user's program. In addition to processor initialization, the startup routine performs some initialization specific to random-access memory (RAM-based) or read-only memory (ROM-based) target environments. Since RAM-based applications typically operate under a system monitor and load to the correct addresses after powering up the board, the startup routine must initialize system monitor requirements but need not boot-load the program. For a ROM-based application, the startup routine must:

- Put the initialization boot record for the i960 processor in place.
- Configure system data structures correctly.
- Make initialized data available in the RAM address space.

For any program, the startup routine must initialize the i960 processor registers as follows:

- Provide a global entry point called `start`. This symbol is the entry point for debug monitors.
- Initialize the frame pointer and stack pointer to the correct value.
- Initialize `g14` to zero, as required by the i960 processor calling convention.

- Fill the uninitialized `.bss` data sections with zeros.
- Set the arithmetic controls (AC) register to `0x3B001000`. For library functions to execute correctly, the rounding mode bits of the AC must be set to round-to-nearest, the floating-point normalizing bit must be set, and the following faults must be masked:
    - integer overflow
    - floating-point overflow
    - floating-point underflow
    - floating-point inexact
- Since the i960 C-series and J-series processors' AC register does not allow setting of floating-point bits, use `_setac` in the setup. The `_setac` and `_getac` routines are independent of architecture and work correctly for all i960 architectures. Startup routines for KA, KB, SA, and SB processors can also use the `modac` instruction as an alternative.

When writing code to initialize the C runtime environment, you must address the following issues:

- The startup code provides the bias value for position-independent data sections. If the program contains position-independent data (PID), startup code must initialize register `g12` to the data-address bias. The `g12` register is the data address bias register. The compiler generates references to statically allocated variables relative to `g12`. The contents of `g12` must be divisible by 16 (i.e., the address must be on a quad-word boundary). After initialization, `g12` must be considered read-only; user code should not modify it.
- If the gcc960 command line specifies `mpid` or the ic960 command line contains the Generate option with the PID argument (`-G pd`), the compiler does not use `g12` as a general purpose register. However, it does use `g12` to offset static variables, as explained above.

If the target environment includes the MON960 monitor, startup must provide a global entry point called `start`, used by debug monitors as the entry point to the new program. Startup code must call `__LL_init` to perform all initialization specific to the processor and to the board.

Initialization differs for each processor and board. For example, some board-specific startup routines initialize `mem_end` in the linker configuration file instead of in `__LL_init`. Each board-specific low-level library included with the assembler contains an appropriate `__LL_init`.

See the startup file crt960.s under the src/lib/libll/common directory for an example.

- If a program uses the C runtime library, startup code must call __HL_init to ensure correct operation of all library functions, including any I/O routines such as printf.
- The __HL_init function calls the _exit_init, _stdio_init, and _thread_init routines to allocate memory for library data structures and to open standard devices. These routines require definition of sbrk and open in the board-specific low-level library. The __HL_init function is in the architecture-specific high-level libc.a library. For more information about high-level libraries, refer to the *i960 Processor Library Supplement*.
- If performing profile-driven optimizations, the startup routine must call a profile initialization routine before calling any instrumented functions.
- If you are linking in any C++ modules, startup code must call _do_global_ctors before you invoke main. See crt960.S for an example.
- The startup routine also calls an executing program's main function, passing parameters to main if necessary. The startup routine also performs cleanup after main returns, usually by calling exit. If the target environment supports program command-line arguments such as argc and argv, call __arg_init to initialize such variables immediately before calling the program main function. The __arg_init function is found in the MON960 low-level library. This function is described in the *Library Supplement*.
- The linker combines the startup routine with other object modules. Normally, a configuration file provides the name of the startup file. To override the startup file named in the configuration, use the linker C (Startup) option. For more information on passing options to the linker from the compiler invocation command line, see Chapter 2, "gcc960 Compiler Driver" or Chapter 3, "ic960 Compiler Driver".

RAM-based Initialization

The lib/cycx.ld configuration file links the crt960.o startup file to run a program under the MON960 monitor.

ROM-based Initialization

ROM-based startup routines must ensure that all the variable data is in RAM. The routines must do the following:

- Physically move any system data structures that the program modifies; move the structures to the RAM address space.
- Move the initialized variable data from ROM to the `.data` section.
- Restart the processor, using the IAC (inter-agent communication) for `KA`, `KB`, `SA`, and `SB` architectures, or using the `sysctl` instruction for the `Cx`, `Hx` and `Jx` architectures.

A startup routine performs the following operations to create a ROM-based application:

- Create an initialization boot record as a separately translated module.
- Create architecture-specific data structures.
- Initialize any necessary board-specific memory subsystems in either the `main` or the startup routine of your program

Use the linker to locate the initialization boot record, system data structures, and program code in the appropriate memory location for the architecture and board configuration, as follows:

- Put `.text` code sections in the ROM address range
- Put `.data` and `.bss` data in the RAM address range

Use the linker to define variables used symbolically in the startup routine. The linker automatically generates symbols named __*`Bsection`* for the beginning and for the end of each section of your program.

The linker can generate the following symbols for the startup routine:

`__Bdata`	is the starting address of RAM data
`__Edata`	is the end of the `.data` section
`__Btext`	is the starting address of the `.text` section
`__Etext`	is the end of the `.text` section
`__Ebss`	is the end of the `.bss` section
`__Bbss`	is the starting address of the `.bss` section
`__Bctors`	is the starting address of the C++ `.ctors` section
`__Ectors`	is the end of the C++ `.ctors` section

`__Bdtors` is the starting address of the C++ `.dtors` section

`__Edtors` is the end of the C++ `.dtors` section

It is also possible to explicitly define variables in the configuration file. Supplied configuration files contain definitions of the following:

`user_stack` is the starting address of the user stack

`supervisor_stack`is the starting address of the supervisor stack

`interrupt_stack`is the starting address of the interrupt stack

After linking, you can use the `move` command of the rom960 utility to modify object module section headers and to place named data sections at specified addresses or locations. This command should be used to temporarily move the data sections into the ROM address space, usually immediately after the `.text` section, and does not change the relocation information contained in the section to be moved. The startup routine then must copy the data to the RAM area specified by the linker.

Linker Configuration Files

A linker configuration file is a linker script that provides information to the linker about the intended execution environment. Several linker configuration files are provided, and each contains linker options to create a complete and unique execution environment. Use the `T` (Target) linker option to specify the configuration file. For more detail on the `T` (Target) option, see the *i960 Processor Software Utilities User's Guide*.

RAM-based Configuration File

The commands passed to the linker define the memory layout and location of the linked program. Configuration information used by the linker includes:

- memory layout
- linker controls
- startup routine
- high-level libraries
- low-level libraries
- floating-point support

ROM-based Configuration File

The optional ROM-builder section of a configuration file contains commands to be passed to the rom960 utility. rom960 commands must begin with the #* characters in columns 1 and 2. The *i960 Processor Software Utilities User's Guide* provides explanations and examples of rom960 commands in a configuration file.

Optimization 12

Readable and maintainable source text is not always organized for efficient execution. The compiler can optimize the arrangement of instructions and data use for faster execution and smaller memory requirements. This chapter describes the different ways in which the compiler can optimize your program and explains ways to control optimization.

Optimization Categories and Mechanisms

Compiler optimizations affect these aspects of your program:

- constants and expression evaluation
- calls, jumps, and branches
- loop optimizations
- memory optimizations
- register use
- instruction selection and sequencing

Some optimizations are independent of the i960 architecture and others take specific advantage of the i960 processor instruction set and registers. Program-level optimizations are also available when profile data exists for the program.

Table 12-1 Constants and Expression Evaluation

Optimization	**ic960**	**gcc960**
Register management	any level	any level
Branch prediction	0	0
Code compression	0	0
Constant-expression evaluation	0	0
Identity collapsing	0	0
Branch optimization	1	1
Char/short cleaning reduction	1	1
Dead-code elimination	1	1
Leaf-function identification	2	2
Local CSE elimination	1	1
Local-variable promotion	1	1
Loop-invariant code motion	1	1
Specialized-instruction selection	1	1
Tail-call elimination	2	2
Conditional transformation	2	2
Global alias analysis	2	5
Induction variable elimination	2	2
Instruction scheduling	2	2
Constant propagation	2	3
Loop unrolling	2	3
Memory access coalescing	2	3
Variable shadowing	2	3
Allocation of variables to fast memory	3	5
Inter-module, inline function expansion	3	5
Profile-based branch prediction bits setting	3	5
Basic block rearrangement	3	5
Superblock optimizations	3	5

The compiler can simplify some arithmetic and boolean calculations involving repeating expressions, constants, or operational identities. Optimizations involving such simplifications are:

- common sub-expression elimination
- constant expression evaluation
- constant propagation
- identity collapsing

Each is explained in one of the following sections.

NOTE. *The following source examples are for illustration only. The compiler performs its transformations on an internal representation, not at the source level.*

Common Sub-expression Elimination

Common sub-expression elimination detects and combines redundant computations within an expression. For example, this line of source text contains the sub-expression `x[a] * y[b][c]` three times:

```
i = (x[a] * y[b][c]) + (x[a] * y[b][c]) + (x[a] * y[b][c]);
```

Instead of calculating `x[a] * y[b][c]` three different times, the compiler rewrites the expression to perform the calculation once and store the result for reuse:

```
temp = x[a] * y[b][c];
i = (temp) + (temp) + (temp);
```

The compiler eliminates common sub-expressions on the results of floating-point operations and on integer operations. In some cases the compiler can perform this optimization for common sub-expressions separated by branch instructions.

This optimization is performed by the `O` (Optimize) compiler option at level `1` (`O1`) and higher.

Constant Expression Evaluation (Constant Folding)

A constant expression contains only constant operands and simple arithmetic operators. Instead of storing the numbers and operators for computation when the program executes, the compiler evaluates the constant expression and uses the result. Constant folding is another name for this optimization.

The examples in Table 12-2 show the effects of constant expression evaluation. The variables d and e are affected by bit-shift operations but are still subject to constant expression evaluation.

Table 12-2 Effects of Constant Expression Evaluation

Original Source Text	Replacement
a = 1 + 2;	a = 3;
b = 3 - 4;	b = -1;
c = 5 * 6;	c = 30;
d = (2 << 1) + 1;	d = 5;
e = (12 >> 2) + 2;	e = 5;
f = 1.2 + 3.8;	f = 5.0;
g = 10.0 * 0.5;	g = 5.0;
h = i + 2 + 5;	h = i + 7;

Any of the following data types can be operands subject to constant expression evaluation:

- integers
- floating-point numbers
- pointers

Dead-Code Elimination

The compiler eliminates two kinds of dead code:

unused when code generates a value that is not used subsequently in the program or in its output.

unreachable when the control flow of the program can never execute the instructions.

Unused code operations can arise from several sources, including:

- Naive code generation can produce operations that are useless in some contexts as part of a generic translation.
- Other optimizations, such as common sub-expression elimination, can make some operations useless.
- Conditional compilation or other code improvements can eliminate the uses of the results of an operation.

By analyzing a program, the compiler can detect and remove useless operations from generated code.

Commonly, instructions become unreachable when function inlining substitutes constants for variables or when the preprocessor substitutes constants for preprocessor symbols. By analyzing the control flow in a program, the compiler can detect many (though not all) instances of unreachable instructions and remove them from the generated code.

Identity Collapsing

The compiler recognizes instances of arithmetic operations in which an identity constant is one of the operands. For an identity constant, the result of the operation is the same as one of the operands. The examples in Table 12-3 demonstrate identity collapsing.

Table 12-3 Identity Collapsing Examples

Original	Replacement
a + 0	a
a * 1	a
a * 0	0
x << 0	x
0 >> y	0

Operations subject to identity collapsing include:

- addition or subtraction
- multiplication or division
- bitwise left or right shift
- bitwise `and`, `xor`, or `or`

Constant Propagation

Programs often contain computations that produce the same value each time the program is executed. Constant propagation involves tracking constant values through the computations in a program. In arithmetic or conditional operations, the compiler can sometimes eliminate less efficient memory or register instructions, replacing them with an instruction sequence that uses constant values. The compiler performs the following types of instruction replacement:

- An integer arithmetic instruction that always produces the same constant value result is replaced by a single instruction (commonly `lda` or `mov`) that copies the constant value into the destination register of the original instruction. For example, this program fragment uses an `addo` to put the sum of `2` and `4` into `g4`:

  ```
  mov    2,  g2
  mov    4,  g3
  addo   g2, g3, g4
  ```

- After constant propagation, the code contains these optimized instructions:

  ```
  mov    2, g2
  mov    4, g3
  mov    6, g4
  ```

- Dead code elimination deletes the first two now-unused `mov` instructions.
- A conditional branch instruction for which the condition is known is deleted. For example, this program fragment sets `x` equal to `y+z` if `2` and `4` are equal, which is never true:

  ```
  a=2; b=4;
  ...
  if (a==b)
         x=y+z;
  else
  x=y-z;
  ```

- After constant propagation, the code contains these optimized instructions:

  ```
  a=2; b=4;
  ...
  if (0)
  ```

```
           x=y+z;
    else
    x=y-z;
```

- Dead-code elimination further reduces the instruction sequence by removing the test and unreachable “then” part, leaving:

```
    a=2; b=4;
     x=y-z;
```

- A conditional branch instruction for which the condition is found to always be true is changed to an unconditional branch. For example, this program fragment branches to `L1` if `2` is less than or equal to `4`, which is always true:

```
    Before         After
    mov             2,  g2
    mov             4,  g3
    cmpi            g2, g3
    ble             L1
    addi            g4, g5, g6
    b               L2
L1:
    subi            g4, g5, g6 subi  g4, g5, g6
L2:                    L2:
```

- A load operation from a memory location found to contain a constant value is replaced by a copy of the constant value into the destination register of the original instruction. For example, the following program fragment loads the constant value `5` from the memory location `_i` into `g3`:

```
lda     5,  g2
st      g2, _i
ld      _i, g3
st      g3, _j
```

- After constant propagation, the code contains these optimized instructions:

```
lda     5,  g2
st      g2, _i
lda     5,  g3
st      g3, _j
```

- Complex memory-addressing modes are sometimes reduced to less complex addressing modes when registers that are components of a memory reference contain constant integer values. For example, this code fragment contains a complex memory-addressing mode in the third instruction:

```
mov     2,  g2
lda     _i, g3
ld      10(g3)[g2*4],g4
```

- After constant propagation, the code contains these optimized instructions:

```
mov     2,  g2
lda     _i, g3
ld      18(g3),g4
```

Calls, Jumps, and Branches

For some branches or function calls, the compiler can replace the original instructions with more efficient instructions to lower execution time or with fewer instructions to reduce program size. Optimizations that perform such restructuring include:

- branch optimization
- branch prediction for i960 Cx and Hx processors
- leaf-function identification
- inline function expansion
- tail-call elimination

Branch Optimizations

Branch optimizations streamline the flow of program control by performing the following actions:

- collapsing branch chains
- eliminating branch-to-next-line sequences
- eliminating branch-around-branch sequences

The following program fragments show branch optimizations.

- This program fragment contains a branch directly to another branch instruction. It doesn't matter whether the branch is conditional or unconditional. After branch optimization, the branch chain is collapsed to a single branch.

Before **After**

```
cmpi    g1, g2      cmpi     g1, g2
bl     .L1          bl      .L2
...                       ...
.L1:                .L1:
 b      .L2         b       .L2
```

The final branch might be eliminated by the dead code optimization.

- This program fragment contains an unconditional branch to the label directly following the branch. After branch optimization, the branch-to-next-line sequence is eliminated:

Before After

```
b  .L1                .L1:
.L1:
```

- In the next program fragment, an unconditional branch follows a conditional branch. The compiler optimizes this branch sequence by removing the unconditional branch and reversing the test on the conditional branch.

Before After

```
cmpi    g1, g2      cmpi    g1, g2
be     .L1          bne     L2
b .L2               .L1:
.L1:
```

Branch Prediction

The i960 Cx and Hx processors provide a branch-prediction bit in conditional branch instructions. If the prediction is correct, the branch takes no cycles to execute; otherwise, the branch takes one or more cycles. For further information on execution speed during branch prediction, refer to the *i960 Cx Microprocessor User's Manual*.

If not profiling, the compiler uses these heuristics to set the branch-prediction bit:

- For backward branches (likely a loop), the compiler predicts that the branch is taken so that the loop is executed more than once.
- For forward branches (conditional operations such as `if-then` statements), the compiler predicts that the branch is not taken.

During profile-driven compilation, each branch's observed behavior is used to set the prediction bit.

Identification of Leaf Functions

The compiler identifies functions that can be called with branch-and-link instruction sequences. The compiler then generates the correct function prologue, epilogue, and symbol table information for the assembler. When this function is called, the compiler generates the `callj` pseudo-instruction. The linker optimizes the call to use branch-and-link instruction sequences. A function called with branch-and-link instruction sequences does not allocate a new stack frame, does not create a new register frame, and thus executes faster than a function invoked with a `call` instruction.

Neither the compiler nor the linker can absolutely identify a function called indirectly through a function pointer as a leaf function. Therefore, the compiler does not optimize such indirectly called functions to branch-and-link instruction sequences.

For an explanation of the two entry points generated for leaf procedures, see the *i960 Processor Assembler User's Guide* and the *i960 Processor Software Utilities User's Guide*.

Inline Function Expansion

Using calls to a function within a program usually takes less space but requires longer execution time than repeating the function body each time it is needed. Inline function expansion replaces a function call with the called function body expanded in place. The inlining optimization increases speed by eliminating call overhead and creates opportunities for further optimization.

The compiler provides user-controllable inlining using pragma `inline`, and with the `__inline` storage class. Additionally, at ic960 optimization level 2, or gcc960 optimization level 3, the compiler performs more automatic procedure inlining, based on heuristics.

In the following example, the `swap` function switches two numbers. The source text contains a function call:

```
void swap(x,y)    /* function body */
    int *x, *y;
    {
    int temp;
    temp = *x; *x = *y; *y = temp;
    }
main()
    {
    ...
    if (a > b) swap(&a, &b);  /* function call */
    printf("The smaller number is %d\n",a);
    ...
    }
```

After inline function expansion, the function body replaces the call:

```
main()
    {
     ...
    if (a > b)
        {
        int temp;
        temp = a; a = b; b = temp;
        }
    printf("The smaller number is %d\n",a);
     ...
    }
```

Tail-call Elimination

When a call directly precedes a return from a function, optimization can sometimes replace the call with an unconditional branch to the called function. This replacement saves execution time since a branch executes faster than a call.

For example, the following algorithm for Ackermann's function uses tail calls:

```
/* Ackermann's function with tail recursion */
int ack(int m,int n)
{
if (m == 0)
    return n+1;
else
    if (n == 0)
        return ack(m-1,1);
    else
        return ack(m-1,ack(m,n-1));
}
```

Tail-call recursion elimination produces the following:

```
/* Ackermann's function with tail recursion eliminated
*/
int ack(int m,int n)
{
label:
if (m == 0)
   return n+1;
else
   if (n == 0)
   {
      n=1;
      m--;
      goto label;
   }
   else
   {
      n = ack(m,n-1);
      m--;
      goto label;
   }
}
Here is C code to illustrate a simple tail recursion.
print_bool (int v)
{
   if (v== 0)
      printf ("FALSE");
   else
      printf ("TRUE");
   return;
}
Here is the generated assembly code.
   cmpibne0,g0,L4
   lda        LC0,g0
   b          _printf
L4:
   lda        LC1,g0
   b          _printf
```

Loop Optimizations

Movement of Loop-invariant Code

Loops are the bodies of `do`, `while`, and `for` statements. The loop-invariant code optimization identifies computations that do not change within a loop (loop-invariant code) and moves them to a point before the entry to the loop.

Induction Variable Elimination

Loops that traverse arrays occur in many programs. To compute the address for references in these arrays the compiler must multiply the array subscript by the size of an array element.

Multiplication is a time-consuming operation. To generate faster code, the compiler can sometimes replace the multiply operation with an add operation.

These methods improve the performance of the code whenever a value computed in a loop is a linear function of a loop iteration variable. Indexing arrays is the most common case.

Loop Unrolling

When the number of times a loop executes can be determined either at compile time, or prior to executing the loop at run time, then this optimization may be performed. Loop unrolling involves duplicating the body of a loop 1 or more times, and changing the loop conditions so that the same number of executions of the loop body occur. This optimization is chosen based on many factors. Two such factors are the size of the loop body and the complexity of the loop termination condition.

Memory Optimizations

Global Alias Analysis

The compiler gathers information about the interaction between loads and stores in the program. With this information, the compiler can remove some of the redundant load-store operations. Assignments into an array are one applicable case.

Two names are aliases when they both reference the same memory location. Without tracing the relationships of values and names, the compiler must treat any value stored through a pointer, called an indirect store, as if it affected any memory location.

Variable Shadowing

The compiler may place a memory object in a register throughout a single-entry, single-exit region (such as a loop) when it can determine that the following are all true:

- There are no references to memory within the region that could overlap the candidate memory object.
- The address of the candidate is a compile-time constant, or it is constant throughout the single-entry, single-exit region and a reference to the object's address is guaranteed to happen at least once whenever the code for the region is executed.
- There are no calls within the region.

In the following example, global migration causes `p` to be loaded once at the beginning of the loop and stored once at the exit point.

```
static int*p;
while (*p != ’\0’)
        p++;
```

Without this optimization, the program loads and stores `p` once for each iteration of the loop.

Register Use

The compiler can use registers to speed up data access. Register optimizations are as follows:

- local variable promotion
- register management
- register spilling

Local Variable Promotion

The compiler promotes a local variable to a register location when the variable’s address is not taken and its storage class is `auto` or `register`.

Local variables stay in their register location through the life of the function. Optimization level `0` suppresses local variable promotion and assigns all variables with `auto` storage class to stack locations.

Register Management

The register allocator phase of the compiler assigns all register operands to the physical registers. For the KB/SB processors, the physical registers available for assignment include the four floating-point registers. For all i960 processors, the physical general-purpose registers available for assignment include `r3` through `r15`, `g0` through `g11`, and `g13`. You must specify the compiler option for position-independent data (gcc960’s `mpid` or `mpid-safe` option or ic960’s `Gpd` or `Gpr` option) to make `g12` unavailable for assignment. Due to the standard calling conventions, `g14` is not available for register-operand assignment.

Register Spilling

Portions of the compiler that run before register allocation can produce code that needs more physical registers than are available in the processor. The register allocator must fit each function’s arbitrarily large burden of register demands into the physical registers implemented in the hardware. To allocate available registers, the compiler must reuse each physical register many times.

When the physical registers cannot meet the demands of a particular function, the register allocator must insert a sequence of instructions, known as spill code, to transfer long-lived values from some of the registers in order to free the registers for more immediate demands.

Instruction Selection and Sequencing

In addition to other optimizations, the compiler can reduce or eliminate instructions that have become redundant or useless. The compiler can also eliminate less efficient instructions or replace them with instruction sequences and addressing modes that take advantage of i960 processor features. These instruction optimizations include:

- code compression
- code scheduling
- specialized instruction selection

Code Compression

The i960 architecture provides complex addressing-mode instructions that enable denser code generation. By default, the compiler tries to pick addressing modes to maximize run-time performance, generally using a mix of complex and simple addressing modes. You can control this optimization with `#pragma compress`, as described in Chapter 7, “Optimization”.

Code Scheduling

In code scheduling, the compiler modifies the sequence of instructions to increase parallel execution. Although the effect of the code does not change, code scheduling can often improve code performance.

Since different members of the i960 family of processors provide varying levels of hardware parallelism, the compiler orders the instructions differently according to the specific processor for which code is being generated.

For example, on the i960 KA, KB, SA, and SB processors, the execution of a memory operation can overlap the execution of an arithmetic instruction, provided the memory operation occurs in the instruction stream first. The following code computes the expression`(b*13) + c` with these instructions:

```
ld   _b, r4
muli r4, 13, r4
ld   _c, r5
addi r5, r4, r4
```

To optimize this computation, the compiler moves the instruction that fetches the value of `c` ahead of the multiply instruction:

```
ld   _b, r4
ld   _c, r5
muli r4, 13, r4
addi r5, r4, r4
```

When this rearranged code executes, part of the instruction `ld _c, r5` executes in parallel with the multiplication. The instruction `ld _b, r4` also executes partly in parallel with the instruction `ld _c, r5`.

The same sort of rearrangement can improve performance on the CA and CF processors, but more parallelism is possible because the CA and CF can issue multiple instructions at one time and can execute more instruction categories in parallel than the KA or KB.

For example, on the CA and CF processors, the compiler can also substitute one instruction for another that has the same effect but executes in a different internal unit of the processor. The most common examples of such substitution are conversions of `mov` instructions to `lda` instructions, and vice versa.

Specialized-instruction Selection

A number of i960 processor instructions can help optimize code in special situations. The special code sequences recognized by the compiler, and the replacements used are as follows:

- A bitwise `or` instruction for which one of the operands is a constant with value `2`^{n}, for some *n*, can become `setbit`.
- A bitwise `and` instruction for which one of the operands is a constant with value `~(2`^{n}`)`, for some *n*, can become `clrbit`.

The i960 processor has a complete set of bitwise-boolean instructions. The compiler takes advantage of this in translating expressions involving bitwise-boolean operations in which the operands or the results are negated. For example, the operations in the expression `~(a & b)` become a single `nand` instruction. Similarly, `(a | ~b)` can use an `ornot` instruction.

Multiplication of an integer or unsigned integer by a constant power of 2 becomes a left-shift operation. Similarly, division of an integer or unsigned integer by a constant that is a power of 2 becomes a right-shift operation.

Program-level Optimization

After program development is complete, it is possible to use the compiler’s profile-driven optimizations to achieve the highest level of program optimization, based on the program’s execution-time profile.

Inter-module Function Inlining

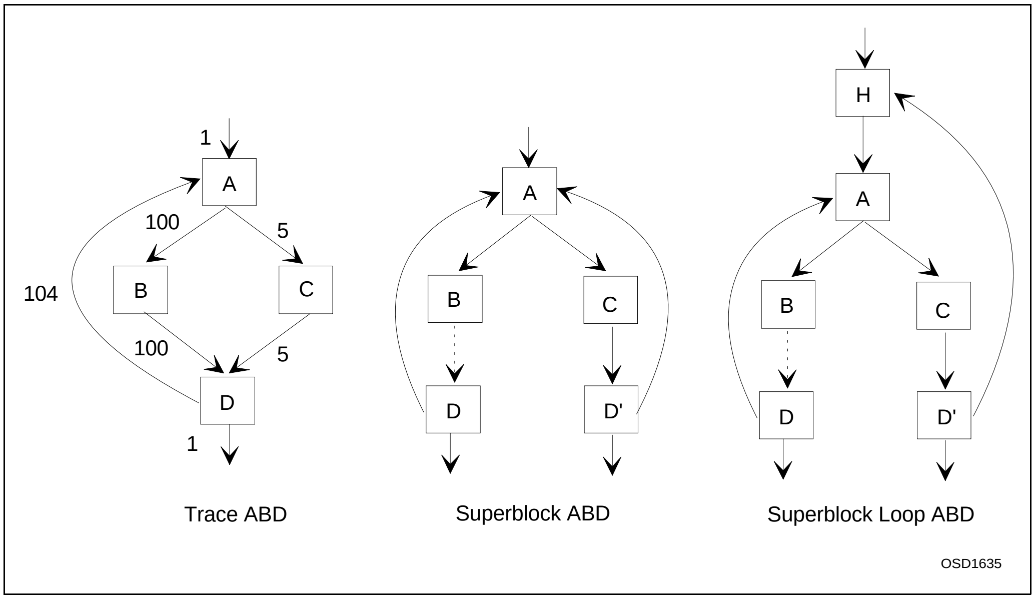
Given program profile data describing the typical behavior of the program, the compiler knows what functions the program calls, from which call sites, and how many times calls are made. Intelligent decisions can be made about which functions to inline at which specific call sites. If a function is called from multiple sites, it is better to inline the function at frequently executed call sites. The inlining decisions are made by the gcdm960 program during the profiling decision-making step. After the decisions have been made, the compiler performs the inlining during profile-driven recompilation.

Superblock Formation

A superblock is a group of basic blocks that tend to execute in sequence (a path) and can be entered only from their initial block. A superblock loop is a superblock whose first block is the header of a loop, and for which execution flow out of the last block usually goes to the first block. In other words, a superblock loop is a heavily iterated loop where a single path through the loop is taken quite frequently.

These concepts are illustrated in Figure 12-1:

Figure 12-1 Superblock Formation Process



The left diagram shows that path A➠B➠D is heavily traveled and would thus be detected as a superblock candidate. To form a superblock from this candidate, it is necessary to remove the arc C➠D. This is done as shown in the middle diagram. Block D is duplicated, and block C is altered to flow to D’. The dashed arc from block B to block D indicates that it is likely that these two blocks will be merged into a single block. This merging increases the scope of the local optimizer and of the scheduler, optimizations that work on a single block at a time. The superblock loop containing only blocks A, B, and D is formed in the diagram on the right. An empty header block, H, has been created, and the original single loop in the middle diagram now becomes two loops, a nested superblock loop headed by A, and an outer loop headed by H.

The fundamental advantage that superblock formation yields is the removal of data dependencies. In the diagram on the left, any data modifications in block C must be considered when optimizing the loop. These modifications often have a negative effect, inhibiting the classic loop optimizations. For example, if block C contains a procedure call, it appears to modify all memory variables. Optimizations involving memory references are inhibited in this case. In the diagram on the right, data modifications in block C do not effect loop optimizations in the superblock loop ABD.

Profile-based Branch-prediction Bit Setting

Without program profile data, the compiler uses a fixed rule for setting the branch-prediction bits for the processor.

With program profile data, the branch-prediction bits are set based on that profile data. This setting is better for a given program.

Optimizing Virtual Function Dispatch

Generally, invoking a virtual function is more expensive than invoking a non-virtual function in C++. Also, other function related optimizations such as inlining cannot be performed on virtual functions. In many situations, the call to the virtual function can be replaced by a direct call to a member function, and if possible it can be inlined at the call site. This improves the runtime performance of the code. Consider the following program segment:

```
class A
        {
public:
        virtual void f(int i)
             { printf("Function A::f called with %d\n,i");}
        } *a;

class B : public A
```

```
        {
public:
        virtual void f(int i)
             { printf("Function B::f called with %d\n",i);}
} B;

main()
{
   a = &B;
   a->f(10);
}
```

The virtual function call `a->f()` always resolves at run time to the function `B::f`. The virtual function optimization phase of the compiler not only resolves this at compile time, it also inlines `B::f` into the function `main.` This improves the runtime performance.

This optimization is not enabled by default. It is performed only if invoked with the appropriate switches. The two-pass framework is needed for this optimization.

This optimization will not work correctly if

- The C++ code is not type safe. Suppose that you have a class D that is derived from class B, then the code is not type safe if a pointer to an object of type B is used as a pointer to an object of type D.
- If a C++ object that has a virtual function associated with it is used, or created in either C or assembly code.
- A C++ file that is a part of the application is not included in the two-pass optimization scheme, or if the two-pass optimization is performed incrementally.

Caveats 13

This chapter provides useful programming tips on:

- “Aliasing Assumptions”
- “Alignment Assumptions”
- “Volatile Objects”C
- “Known Problems Using the Compiler”
- “C Version Incompatibilities”
- “Troubleshooting”

Aliasing Assumptions

Some compiler optimizations (for example, `fshadow-mem`) use type information as the basis for several assumptions. These assumptions exclude some pairs of memory references as possible alias candidates.

If your program violates these assumptions, the compiler may generate code that does not function as you intended.

Here are the rules the compiler uses:

character (i.e., `char`, `unsigned char`, `signed char`) lvalues can access all objects, regardless of type.

ordinal (e.g., `int`, `short`, `long`, `enum`) lvalues can access only ordinal objects of the same size (regardless of sign) or character objects.

real (e.g., `float`, `double`, `long double`) lvalues can access only real objects of the same size, or character objects.

pointer — lvalues can access only objects of pointer type (regardless of the types pointed to) or character objects.

structure — lvalues can access only the objects that can be accessed by the members of the structure, or `struct` objects of the same size, or character objects.

union — lvalues can access only the objects that can be accessed by the members of the union, or union objects of the same size, or character objects.

These rules are not as strict as those allowed by the relevant portion of the ANSI standard (section 3.3), but they are still aggressive enough to cause some problems with code developed for some compilers.

The `fint-alias-ptr`, `fint-alias-real`, and `fint-alias-short` compiler options relax these restrictions. See Chapter 2, "gcc960 Compiler Driver" and Chapter 3, "ic960 Compiler Driver" for more information.

To make use of the higher optimization levels, you should examine your code carefully and ensure that these rules are not violated.

Consider this code fragment:

```
double *pq, *pr, *ps;
int* pi, *pj;
*pq = *pr;
*pi = *pj;
*ps = *pr;
```

The compiler might conclude that the value of `*pr` is unaffected by the assignment to `*pi`, because double objects cannot legally be referenced by `int` lvalues.

It might then use this conclusion to rewrite the above code as follows:

```
register double t = *pr;
*pq = t;
*pi = *pj;
*ps = t;
```

This is fine as long as `*pi` really doesn’t overlap `*pr`, but if your program does something like:

```
double d;
pi = (int *) &d;
pr = &d;
```

before it executes the second fragment, the wrong value would get stored in `*ps`.

Alignment Assumptions

The compiler sometimes uses pointer type information when deciding whether or not memory references are properly aligned for some optimizations.

Thus, the compiler assumes that all pointer expressions are aligned as their pointed-to types would indicate. For example, `((double *) e)` is treated as an assertion that the low 3 bits of `e` are `0`.

The compiler also infers more stringent alignment for individual variables than would be indicated by their types alone, since it assumes that the allocation is aligned according to the compiler's rules.

So, if your program defines global variables in assembly code that are referenced by C routines, or if it has its own memory manager (*e.g.*, `malloc`), the allocations must be aligned according to the compiler's rules or unaligned references may result.

Here is an example of how these assumptions are used:

```
#include <string.h>
 ...
struct {
  int s1;
  int s2;
  int s3;
} *s;      /* (1) *s is assumed to be 16 byte aligned */
extern char mybuf[23];
 /* (2) mybuf is assumed to be 16 byte aligned */
memcpy (mybuf, s, sizeof (*s));
```

The compiler would generate:

```
ldt (s), r
stt r, mybuf
```

in lieu of the call to `memcpy`; the memory references would be unaligned should the assumptions mentioned above prove false.

Volatile Objects

The compiler aggressively attempts to remove redundant memory references (both loads and stores), and it attempts function inlining across multiple `.c` files. If your program expects actual memory references to be made at certain points in the program, you must make those references volatile. Volatile objects are guaranteed to be updated at certain sequence points in the program (e.g., between semicolons, `&&`, `||`, `?:`, and before calls).

Volatile objects are also presumed to have been changed in unknowable ways between such points.

Here is an example of a program that fails because of a memory reference that needs to be made volatile:

```
fiddle.c:
  #define MY_PORT *((int *) 0x10000)
   ...
  int read_my_port ()
  { return MY_PORT;
  }
faddle.c:
   ...
while (read_my_port() == 0)
  /* do nothing */;
ok_go_do_something ();
```

This program is incorrect, but it functions as intended when compiled with compilers that do not attempt inlining across `.c` files.

When these two files are compiled with global inlining, the compiler translates the program to:

```
(1)   while (MY_PORT == 0)
       /* do nothing */;
       ok_go_do_something ();
```

And, since `MY_PORT` appears to be loop invariant (because it isn't volatile), we then get:

```
(2)   t = MY_PORT;
       while (t == 0)
         ;
```

which loops forever if the first value read from `*0x1000` is `0`.

All that is needed here is to make `MY_PORT` volatile, as follows:

```
#define MY_PORT *((volatile int *) 0x10000)
```

This suppresses (2), as `MY_PORT` must be considered to have changed between iterations of the loop.

Known Problems Using the Compiler

Here are some of the things that have caused trouble for people using the compiler.

Type Promotion

Users often think it is a bug when the compiler reports an error for code like this:

```
int foo (short);
int foo (x)
     short x;
{...}
```

The error message is correct: this code really is erroneous, because the old-style non-prototype definition passes subword integers in their promoted types. In other words, the argument is really an `int`, not a `short`. The correct prototype is this:

```
int foo (int);
```

Prototype Scope

Users often think it is a bug when the compiler reports an error for code like this:

```
int foo (struct mumble *);
struct mumble { ... };
int foo (struct mumble *x)
{ ... }
```

This code really is erroneous, because the scope of the `struct mumble` prototype is limited to the argument list containing it. It does not refer to the `struct mumble` defined with file scope immediately below — they are two unrelated types with similar names in different scopes.

But in the definition of `foo`, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype do not match, and you get an error.

longjmp and Volatile Data

If you use `longjmp`, beware of automatic variables. ANSI C says that automatic variables that are not declared `volatile` have undefined values after a `longjmp`. And this is all the compiler promises to do, because it is very difficult to restore register variables correctly, and one of the compiler's features is that it can put variables in registers without being asked.

Incorrect debug information generated for arrays with unspecified bounds.

Consider the following example

```
int arr[];
```

The compiler generates debug information for the above declaration as if `arr` were an array of 1 integer. As a result, when you do a `ptype arr` in gdb960 the type of `arr` is displayed as `int [1]`.

C Version Incompatibilities

There are several noteworthy incompatibilities between Intel C for the 80960 architecture and some (non-ANSI) versions of C.

String Constants Read-only

The compiler normally makes string constants read-only. If several identical-looking string constants are used, the compiler stores only one copy of the string.

If this is a problem for your application, the best solution is to change the program to use `char`-array variables with initialization strings for these purposes instead of string constants. But if this is not possible, you can use the `fwritable-strings` flag, which directs the compiler to handle string constants the same way most C compilers do. `ftraditional` also has this effect, among others.

No Macro Argument Substitution in Strings

The compiler does not substitute macro arguments when they appear inside of string constants. For example, the following macro:

```
#define foo(a) "a"
```

produces output `“a”` regardless of what the argument `a` is.

The `ftraditional` option directs the compiler to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.

External Variables and Functions in Blocks

Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

In some other C compilers, an `extern` declaration affects all the rest of the file even if it happens within a block.

The `ftraditional` option directs the compiler to treat all `extern` declarations as global, like traditional compilers.

Combining long with typedef Names

In traditional C, you can combine `long`, etc., with a typedef name, as shown here:

```
typedef int foo;
typedef long foo bar;
```

In ANSI C, this is not allowed: `long` and other type modifiers require an explicit `int`. Because this criterion is expressed by grammar rules rather than C code, `ftraditional` cannot alter it.

Using typedef Names in Function Parameters

Some C compilers allow typedef names to be used as function parameters. Because this criterion is expressed by grammar rules rather than C code, `ftraditional` cannot alter it.

Whitespace in Compound Assignment Operators

Some C compilers allow whitespace in the middle of compound assignment operators such as `+=`. The CTOOLS960 and GNU/960 compiler, following the ANSI standard, does not allow this. Because this criterion is expressed by grammar rules rather than C code, `ftraditional` cannot alter it.

Flagging Unterminated Character Constants

The compiler flags unterminated character constants inside of preprocessor conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, the compiler will probably report an error. For example, this code produces an error:

```
#if 0
You can’t expect this to work.
#endif
```

The best solution to such a problem is to put the text into an actual C comment delimited by `/*...*/`. However, `ftraditional` suppresses these error messages.

Disguised varargs or stdarg Routines

Disguised `varargs` routines (those that do not use `varargs.h` or `stdarg.h` but that increment through a pointer assigned from the address of an argument) do not work.

Troubleshooting

Undefined References

When trying to compile a program, a user may get error messages similar to the following:

```
crt960.o: undefined reference to ‘heap_size’
crt960.o: undefined reference to ‘__setac’
crt960.o: undefined reference to ‘__LL_init’
_filbuf.c:47: (_filbuf): undefined reference to ‘_read’
exit.c:31: (_exit_init): undefined reference to
```

```
‘__exit_create’
exit.c:39: (exit): undefined reference to ‘__exit_ptr’
fflush.c:38: (fflush): undefined reference to ‘_write’
_flsbuf.c:105: (_flsbuf): undefined reference to ‘_write’
fclose.c:43: (fclose): undefined reference to ‘_close’
malloc.c:82: (malloc): undefined reference to ‘_sbrk’
malloc.c:60: (malloc): undefined reference to ‘_brk’
```

Problem:

When invoked with `gcc960 -ACA -o` *`filename filename`*`.c`, the compilation system tries to construct a b.out format executable file, fully linked. A fully linked file implies a C-runtime startup file and several runtime libraries. If the proper library list (in the proper order) is not added to the invocation command, the error messages listed above may result.

Solution:

The preferred method of creating fully linked executables is to use the target configuration files, e.g., `gcc960 -o` *`filename filename`*`.c -T`*`arg`*. The `-T`*`arg`* option instructs the compiler to parse the file `$G960BASE/lib/`*`arg`*`.gld`, which contains definitions for the i960 architecture flag, C-runtime filename, library lists, and section load addresses. Target configuration files are supplied for all the i960 processor evaluation boards, and adding your own description file is as easy as renaming and modifying an existing description file. Do not confuse gcc960’s `-T` option with ic960’s and gld960's `-T` option.

C Interrupt Service Routine Failures

An application that uses interrupts extensively may have hand-built assembler wrappers for each interrupt type, with each wrapper calling specific C interrupt service routines. Some of the C interrupt service routines may fail in mysterious ways, often in an operation fault.

Problem:

The C function calling convention requires that the i960 processor register `g14` contain the value zero for all functions that take fewer than 14 words of parameters and are non-leaf procedures. Because of this, for most functions, the compiler assumes `g14` to contain zero, and uses that register as a zero constant. If your application happens to be interrupted with `g14` containing a non-zero value, then your C interrupt service routine is called with `g14` containing a non-zero, but used as a zero constant.

Solution:

When calling any C function from assembly source, always zero `g14` prior to the function call. Also, be sure to save all global registers prior to calling your C function, and restore those registers prior to returning from the interrupted state.

Preventing Structure Padding

You may be using an i960 processor to communicate with another processor. The communication involves passing structures between the two processors. The Intel compiler pads the structures, but the compiler for your other processor does not, causing passed structure members to contain incorrect values. It is necessary to prevent the Intel compiler from padding your structures and unions.

Problem:

The Intel compiler uses fairly strict data-type alignment rules, which take advantage of the i960 processor features supporting memory references. This increases the performance of programs running on the i960 processor, but makes it more difficult to interface through structs/unions to other processor types or to read binary data from a file.

Solution:

gcc960's `#pragma-align` lets you control the compiler's alignment rules for aggregate data types on a per-definition basis, and therefore control the padding added to the end of structures and unions.

In this case, `#pragma-align 1` could be added to your code before the structure definition to remove trailing structure pads and properly match structure members. `#pragma-align 0` could then be added after the structure definition to return to normal alignment rules, thereby reducing its impact on the performance of the entire program.

However, #pragma-align has limitations. Although it can be used to restrict the padding of aggregate data types (and arrays of those types) it does not change the alignment rules for individual structure members. For information on alignment rules for structure members, see the discussion of pragma pack in Chapter 7, "C Language Implementation".

Consider the following example:

```
struct test {
   char       first;
   int        second;
   short      third;
};
```

If you compiled the above structure without modification, the structure size would be 16 bytes. If you defined pragma align 1 before the structure definition, the structure size would be 12 bytes - four pad bytes removed. In both cases, however, the position of the elements would not have changed, with element "first" at address offset zero, element "second" at address offset 4, and element "third" at address offset 8. This element placement effectively creates three pad bytes between the first and second structure elements.

To work around the limitations of intra-structure padding, consider the case where the above structure must be read in from a binary file written by a processor/tool pair that inserted zero (intra-struct) pad bytes.

The following code demonstrates one way to perform that function:

```
#include <unalign.h>
/* The following structure is what gcc960 compiles.
 * The buffer, when filled, contains the same
 * structure in packed format - all pad bytes removed. */
struct test {
   char    first;
   int     second;
   short   third;
} 960_struct;
unsigned char packed[7];
/* sum of 960_struct element sizes */
/* Read binary data from a file and copy into a
 * structure that has different alignment rules. */
main()
{
   int           fdesc;
   unsigned char *ptr;
   /* Assume file opened and ready for reading...
    * Then read one struct’s worth of bytes. */
   if (read(fdesc, packed, 7) != 7) {
      /* Handle read error. */
   }
   /* Fill up structure. Done. */
   ptr = packed;
   960_struct.first = *(char *)ptr;
   ptr += sizeof(960_struct.first);
   960_struct.second = GET_UNALIGNED(ptr,int);
                                     /*  *(int *)ptr; */
   ptr += sizeof(960_struct.second);
   960_struct.third =  GET_UNALIGNED(ptr,short);
                                    /* *(short *)ptr; */
}
```

Although the code shown above is expensive in terms of performance, using `#pragma align` also has a significant performance penalty. To get the best performance, use the default alignment rules and use pragmas only where absolutely necessary. See the discussions of gcc960’s `pragma align` and `pragma pack` in Chapter 7, “C Language Implementation” for a detailed discussion of alignment.

Breakpoints Inside Interrupt Handlers

If your application uses interrupts extensively, when debugging interrupt handlers with gdb960, breakpoints set inside the handlers may not work and may result in operation faults.

Problem:

When the i960 processor invokes an interrupt handler, it first disables tracing by saving, then clearing, the state of the trace-enable bit and the trace-fault-pending flag. On return from the interrupt handler, the processor restores the process-controls register to its state prior to the interrupt. This restores the state of the trace-enable bit and the trace-fault-pending flag; therefore, standard interrupt handlers cannot contain breakpoints.

Solution:

To set breakpoints inside an interrupt handler, you can modify that handler, probably in the assembler wrapper, adding code to change the state of the trace-enable bit.

Messages 14

This chapter describes the diagnostic messages that the compiler produces when invoked with the ic960 driver, or with the gcc960 driver and the `ffancy-errors` option. (Invoking the compiler with `ic960 -Jgd` produces the corresponding gcc960-style message format and output.)

On UNIX systems, the compiler displays error messages, along with the erroneous source line, on the standard error device. In Windows systems, messages appear on the standard output device. However, if `I960ERR` is defined, messages appear on the standard error device. To display or suppress warning messages, use the `w` (Diagnostic-level) compiler option. Additionally, the `h` (Help), `v` (Verbose), and `V` (Version) options display more information about the compiler, assembler, and linker invocations and about the host system.

Diagnostic messages provide syntactic and semantic information about source text. Syntactic information can include, for example, syntax errors and use of non-ANSI C. Semantic information includes, for example, unreachable code. If a source listing is requested, the compiler puts diagnostic messages in the program listing, as well as displaying them to the standard error device.

Several levels of diagnostic messages can occur:

Command-line diagnostics	report improper command-line options or arguments.
Warning messages	report legal but questionable use of C. The compiler displays some warnings by default. To suppress all warning messages, set the diagnostic level to `2`. To enable all warning messages, set the diagnostic level to `0`. Warnings do not stop translation and linking, nor do they interfere with any output files.
Error messages	report syntactic or semantic misuse of C. The compiler always displays error messages. Errors do not stop translation but do suppress object code for the module containing the error. Errors also prevent linking.
Catastrophic error messages	report occurrences of the `#error` macro, unrecognized command-line options, and file input/output errors. Catastrophic error conditions stop translation and linking. If a catastrophic error ends compilation, the compiler displays a termination message on the standard error device.
Internal error messages	If a compilation produces any internal errors, contact Customer Support.

Messages on the Standard Error Device

Command-line messages appear on the standard error device in this form:

```
ic960 [ ERROR | WARNING ]: message
```

Other diagnostic messages appear on the standard error device in this form:

```
source-line
diagnostic-pointer
diagnostic-message
```

source-line	is the line containing the error being reported.
diagnostic-pointer	is a caret (^) located below the beginning of the token that the diagnostic refers to.
diagnostic-message	has this form:
ic960 *level* *filename*, line *lnn*, -- *message*	
level	is the type of diagnostic message: WARNING, ERROR, CATASTROPHIC ERROR, or INTERNAL ERROR.
filename	names the source file currently being processed.
lnn	is the line number, if available, where the compilation system detects the condition.
message	explains the diagnostic.

The *source-line* and *diagnostic-pointer* may be absent for those messages that are not associated with any particular source code line.

The *diagnostic-pointer* may be absent when the *source-line* is present if the precise column for the error is not available.

Messages in the Listing File

In a source listing, diagnostic lines follow the erroneous source lines. The diagnostic lines in a source listing have this form:

```
>>>>>  source-line
>>>>>  diagnostic-pointer
>>>>>  diagnostic-message
```

source-line	is the line containing the error being reported.
diagnostic-pointer	is a caret (^) located below the beginning of the token that the diagnostic refers to.
diagnostic-message	has this form:
ic960 *level filename*, line *lnn*, -- *message*	
level	is the type of diagnostic message: WARNING, ERROR, CATASTROPHIC ERROR, or INTERNAL ERROR.
filename	names the source file currently being processed.
lnn	is the line number, if available, where the compilation system detects the condition.
message	explains the diagnostic.

The *source-line* and *diagnostic-pointer* may be absent for those messages that are not associated with any particular source code line.

The *diagnostic-pointer* may be absent when the *source-line* is present if the precise column for the error is not available.

If *source-line* is shown, and the error being reported starts and ends on that line, the filename and line number does not appear in the diagnostic message. This is an example of a listing file containing diagnostic messages:

```
              ic960 5.0, Tue Nov  9 08:45:17 PST 1995 "ex_err.c"

Include  Line
 Level  Number  Source-lines
======= ======  ============
    0*      1   #include "ex_err.h"

        >>>>>   struct foo bar {
        >>>>>
        >>>>>   ic960 ERROR: "ex_err.h", line 2 -- syntax error before ’{’

    0       2
    0       3   main ()
    0       4   {
    0       5       struct foo bar;
    0       6       bar.x=3;

        >>>>>       bar.x=3;
        >>>>>           ^      ^
        >>>>>   ic960 ERROR: invalid use of undefined type ’struct foo’

    0       7   }
```

Glossary

arithmetic control (AC) register
For processors with on-chip floating-point support, the register that contains the floating-point exception flags, floating-point exception masks, and rounding-mode bits. For processors without on-chip floating-point support, the AC register is implemented as a predefined variable (`fpem_CA_AC`).

basic block
An assembly language sequence of code that has one entry point and one exit point.

calling convention
The rules that specify the use of registers and the stack for parameter passing and return values in function calls.

command-option file
DOS command-line file, containing command-line options, input filenames, and comments, to be specified on the command line.

common subexpression elimination (CSE)
Avoid recomputing an expression if the compiler can reuse a previously computed value of the same expression.

conditional compilation
Compiling only part of the source code, depending on the preprocessor’s evaluation of conditions you specify.

constant folding

Deducing at compile time that the value of an expression is a constant and using the constant in place of the expression.

constant propagation

Replacing use of variables known to have a constant value with the constant value.

dead function

A function which cannot be referenced during the profile recompilation step. If a function has been in-lined at all known call sites, or if the function is never referenced, then the function is dead.

execution environment

The hardware and software of the system on which your program executes.

floating-point registers

Registers `fp0` through `fp3`, available on processors with on-chip floating-point support.

gcdm960

The decision-making tool that analyzes profile data to make optimization decisions.

global registers

Registers `g0` through `g15`.

gmpf960

The utility that merges execution profiles for use by gcdm960.

inline assembly language

Assembly-language statements or functions in the C source text.

inline function expansion

Replacing a function call with the instructions that comprise the function, rather than calling the function.

instruction set

The set of all possible executable instructions.

instrument

Insert new code into an existing program so that execution data is recorded at runtime.

instrumented program

A program that has had record keeping code inserted to allow creation of a run-time profile of the program's execution.

interrupt handler

A function to be called when an interrupt occurs.

leaf function

A function that is called with a branch-and-link instruction sequence.

macro

An identifier that the preprocessor replaces with C source text that you specify.

object module

The formatted object code resulting from compilation and assembly.

padding

Interleaving unused bytes between struct/union members and at the tail of structs/unions to ensure that struct/union members are properly aligned.

preprocessor file

A text file generated by the compiler, containing the intermediate source code after macro expansion, file inclusion, and conditional compilation.

primary source file

A file that contains C source text, has a `.c` filename extension, and is specified as an input file on the command line.

primary source text

The contents of the primary source file, without any text from include files.

profile-based

Optimizations that depend on profile information gathered by execution of an instrumented program. The term is interchangeable with profile-driven.

profile data

Both static and dynamic program level data.

static profile data

Information that the compiler derives at compile time about the program (e.g., which functions are defined in a module, which functions are called from within a specific function, which variables are defined in a module, which variables have had their addresses used).

strength reduction

An optimization that substitutes expensive operations such as multiplications with low-cost operations such as addition or subtraction. Strength reduction also eliminates unnecessary induction variables. For example, consider the following C code fragment:

```
int v, a[10], j, t4, t5;
. . .
do {
j = j - 1;
t4 = 4 * j;
t5 = a[t4];
} while ( t5 > v );
```

Note that the values of `j` and `t4` remain in lock-step; every time the value of `j` decreases by 1, that of `t4`

tail call

A call that immediately precedes the return to the calling function.

unreachable code

Code that can never execute because the flow-of-control bypasses it.

Index