Getting Started with the 80960 QUICKval Kit

i960® Microprocessor Evaluation Kit

Order Number: 632708-005

Revision	Revision History	Date
-001	Original Issue.	12/94
-002	80960Hx and PCI80960DP chapters added. Additional examples for all processors added.	12/95
-003	CTOOLS 5.0 support.	02/96
-004	Information on 80960RP, IQ80960RP evaluation board, and CTOOLS 5.1 support added.	02/97
-005	CTOOLS 6.0 support	01/98

Contents

Chapter 1 Introduction

About this Kit 1-1
What's New In CTOOLS 1-2
About This Manual 1-2
Intel Support Services 1-4
Notation Conventions 1-4

Chapter 2 Software Installation

System Requirements 2-1
Installing CTOOLS 2-2
Installing CTOOLS in Windows 95/Windows NT 4.0 2-2
Windows NT 3.51 Installation 2-2
AUTOEXEC.BAT Change Summary 2-3
Installing the QUICK*val* Example Programs 2-4
Installing QUICK*val* Example Programs in Windows 95/ Windows NT 4.0 2-4
Where Do You Go From Here? 2-4

Chapter 3 Hardware Installation

Inspecting Your Board 3-1
Installing Your CPU module 3-2
Setting Your Base Board Switches 3-3
Setting Your CPU Module Switches 3-3

Setting the CPU Module Frequency Switch 3-3
Setting Your CPU Module Interrupt Switch 3-4
Installing MON960 on the IQ80960Rx Platform 3-5
Items Needed 3-5
If You Are Using an IQ80960RP (5V) Evaluation Board... 3-5
If You Are Using an IQ80960RD/RPLV (3.3V) Evaluation Board... 3-6
Connecting the Evaluation Base Board to the Host............... 3-6
PCI80960DP 3-7
Connecting the Serial Communications Cable (Optional) 3-7
Connecting the Parallel Communications Cables (Optional) 3-7
IQ80960Rx... 3-8
Connecting the Serial Communications Cable (Optional) 3-8
EP80960BB 3-9
Where Do You Go From Here?... 3-10
Troubleshooting the PCI-SDK and IQ-SDK Platforms 3-10
Verifying the PCI-SDK and IQ-SDK Platforms 3-11

Chapter 4 The i960 Rx CPU Example Programs

System Validation 4-5
Hello World.. 4-5
Memory Test.. 4-6
If Something Goes Wrong.. 4-8
MON960 Debug Monitor is Not Responding... 4-8
Invoking the gcc960 Compiler Resulted in Errors... 4-8
Invoking the gld960 Linker Resulted in Errors... 4-9
Invoking the gdb960 Debugger Resulted in Errors....... 4-9
Data Cache Tutorial 4-9
Instruction Cache Tutorial 4-11

DMA Example 4-12
- How Does This Program Work? 4-12
- Positive Address Decoding 4-13
- Writing the Destination... 4-13
- What Does This Example Do?.................................. 4-13

DMA Tutorial... 4-13
Messaging Unit Example... 4-14
Example Description 4-14
- Address Translation 4-14
- Example Structure 4-15
- Rp_code.c.. 4-16

Messaging Unit Tutorial 4-16
- Host Program DOS Window 4-16

i960 Rx Processor Program Command Prompt Window 4-16
- Program Termination 4-18

Static, Global, and Profile-Driven Optimizations............. 4-18
- C No Optimization.. 4-19
- C Static Optimization 4-19
- C++ No Optimization 4-20
- C++ Static Optimization 4-20
- C Global Optimization... 4-21
- C++ Global Optimization... 4-22

Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation.............................. 4-23
C++ Virtual Function Optimizations............................... 4-24
Building Self-contained Profiles with gmpf960 4-26
Compression Assisted Virtual Execution (CAVE)........... 4-30
- C Example 4-31

C++ Compression Assisted Virtual Execution (CAVE)... 4-34
XLATE960 Tutorial... 4-38
- Looking at the xlt.s File 4-39

Using xlate960 4-41
Running the New Rx-compatible Source Code.......... 4-46
Assembler Pseudo-instruction Tutorial........................... 4-48
What Are Pseudo-instructions?.................................. 4-48
pseudop.c: Editing the File for the Cx Microprocessor 4-48
Running pseudop.c For the Cx Microprocessor............. 4-50
pseudop.c: Migrating the File to the Jx/Hx/Rx Microprocessor 4-52
Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors... 4-53
pseudop.c: Adding Pseudo-Ops to the Program........ 4-56
Running pseudop.c with Pseudo-instruction 4-57
Debugging with gdb960 4-61
Debugging Optimized Code... 4-64
Debugging Optimized C++ Code Tutorial....................... 4-68
Writing Flash on the IQ80960RP Evaluation Board........ 4-78
Writing Flash on the IQ80960RD Evaluation Board........ 4-80
80960Rx Initialization Example...................................... 4-83
Module: INIT.S.. 4-83
Module: MAIN.C... 4-84
MODULE: CYRP_HW.C.. 4-84
MODULE: MONITOR.C 4-85
Other i960 Processor Choices and the Remote Evaluation Facility... 4-85

Chapter 5 The i960 Hx CPU Example Programs

System Validation 5-6
Hello World... 5-6
Memory Test... 5-7
If Something Goes Wrong... 5-8
MON960 Debug Monitor is Not Responding.............. 5-8

- Invoking the gcc960 Compiler Resulted in Errors........ 5-9
- Invoking the gld960 Linker Resulted in Errors... 5-9

Data Cache Tutorial.. 5-11
Instruction Cache Tutorial.. 5-12
Register Cache.. 5-13
External Interrupts Tutorial 5-15
Internal Interrupts Tutorial.. 5-16
Fault Handling 5-17
Static, Global, and Profile-Driven Optimizations............. 5-19

- C No Optimization... 5-20
- C Static Optimization 5-20
- C++ No Optimization 5-21
- C++ Static Optimization 5-21
- C Global Optimization... 5-22
- C++ Global Optimization... 5-23

Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation............................... 5-23
C++ Virtual Function Optimizations............................... 5-24
Building Self-contained Profiles with gmpf960 5-26
Profiling a Program in Pieces 5-30
Compression Assisted Virtual Execution (CAVE)........... 5-35

- C Example 5-36

C++ Compression Assisted Virtual Execution (CAVE)... 5-38
Linker Consumption... 5-43
Assembler Pseudo-instruction Tutorial 5-46

- What Are Pseudo-instructions? 5-46
- pseudop.c: Editing the File for the Cx Microprocessor.. 5-46

Running pseudop.c for the Cx Microprocessor.............. 5-48

- pseudop.c: Migrating the File to the Jx/Hx/Rx Microprocessor.. 5-50
- Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors 5-51

pseudop.c: Adding Pseudo-Ops to the Program 5-54
Running pseudop.c with Pseudo-instruction 5-55
Debugging with gdb960 5-59
Debugging Optimized Code 5-62
Debugging Optimized C++ Code Tutorial........................ 5-66
Writing Flash 5-75
How to Add Benchmarking Routines to Your Code......... 5-78
Other i960 Processor Choices and the Remote
Evaluation Facility 5-79

Chapter 6 The i960 Jx CPU Example Programs

System Validation 6-5
Hello World.. 6-5
Memory Test.. 6-7
If Something Goes Wrong... 6-8
MON960 Debug Monitor is Not Responding... 6-8
Invoking the gcc960 Compiler Resulted in Errors... 6-9
Invoking the gld960 Linker Resulted in Errors... 6-9
Data Cache Tutorial 6-11
Instruction Cache Tutorial 6-12
Register Cache 6-13
External Interrupts Tutorial.. 6-15
Internal Interrupts Tutorial 6-16
Halt Mode.. 6-17
Fault Handling 6-19
Static, Global, and Profile-Driven Optimizations 6-21
C No Optimization 6-22
C Static Optimization.. 6-22
C++ No Optimization.. 6-23
C++ Static Optimization 6-23
C Global Optimization 6-24
C++ Global Optimization 6-25

Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation 6-25
C++ Virtual Function Optimizations 6-26
Building Self-contained Profiles with gmpf960 6-28
Profiling a Program in Pieces 6-32
Compression Assisted Virtual Execution (CAVE) 6-37
C Example 6-38
C++ Compression Assisted Virtual Execution (CAVE) ... 6-41
Linker Consumption 6-45
XLATE960 Tutorial 6-48
Looking at the xlt.s File 6-49
Using xlate960 6-51
Running the New Rx-compatible Source Code 6-57
Assembler Pseudo-instruction Tutorial 6-59
What Are Pseudo-instructions? 6-59
pseudop.c: Editing the File for the Cx Microprocessor 6-59
Running pseudop.c for the Cx Microprocessor 6-61
pseudop.c: Migrating the File to the Jx/Hx/Rx Microprocessor 6-63
Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors 6-65
pseudop.c: Adding Pseudo-Ops to the Program 6-67
Running pseudop.c with Pseudo-instruction 6-69
Debugging with gdb960 6-73
Debugging Optimized Code 6-76
Debugging Optimized C++ Code Tutorial 6-80
Writing Flash 6-89
How to Add Benchmarking Routines to Your Code 6-92
Other i960 Processor Choices and the Remote Evaluation Facility 6-93

Chapter 7 The i960 Cx CPU Example Programs

System Validation 7-5
    Hello World... 7-5
    Memory Test... 7-7
If Something Goes Wrong.. 7-8
    MON960 Debug Monitor is Not Responding... 7-8
    Invoking the gcc960 Compiler Resulted in Errors... 7-8
    Invoking the gld960 Linker Resulted in Errors... 7-9
    Invoking the gdb960 Debugger Resulted in Errors... 7-9
Data Cache Tutorial (80960CF Only).............................. 7-10
Instruction Cache Tutorial (80960CF Only)..................... 7-12
External Interrupts Tutorial... 7-13
Fault Handling... 7-15
DMA Tutorial 7-16
Static, Global, and Profile-Driven Optimizations 7-17
    C No Optimization 7-18
    C Static Optimization.. 7-19
    C++ No Optimization.. 7-19
    C++ Static Optimization 7-20
    C Global Optimization 7-20
    C++ Global Optimization 7-21
Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation 7-22
C++ Virtual Function Optimizations 7-23
Building Self-contained Profiles with gmpf960................ 7-24
Profiling A Program In Pieces 7-29
Compression Assisted Virtual Execution (CAVE) 7-34
    C Example... 7-35
C++ Compression Assisted Virtual Execution (CAVE) ... 7-37
Linker Consumption 7-41

Assembler Pseudo-instruction Tutorial 7-44
What are Pseudo-instructions?................................. 7-44
pseudop.c: Editing the File for the Cx Microprocessor... 7-45
Running pseudop.c for the Cx Microprocessor.............. 7-47
pseudop.c: Migrating the File to the Jx/Hx/Rx Microprocessor... 7-48
Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors... 7-50
pseudop.c: Adding Pseudo-Ops to the Program 7-53
Running pseudop.c with Pseudo-instruction............. 7-54
Debugging with gdb960... 7-58
Debugging Optimized Code 7-61
Debugging Optimized C++ Code Tutorial....................... 7-66
Writing Flash 7-75
How to Add Benchmarking Routines to Your Code........ 7-78
Other i960 Processor Choices and the Remote Evaluation Facility 7-79

Chapter 8 The i960 Sx CPU Example Programs

System Validation... 8-4
Hello World 8-4
Memory Test... 8-5
If Something Goes Wrong 8-7
MON960 Debug Monitor is Not Responding... 8-7
Invoking the gcc960 Compiler Resulted in Errors...... 8-7
Invoking the gld960 Linker Resulted in Errors... 8-8
Invoking the gdb960 Debugger Resulted in Errors... 8-8
External Interrupts Tutorial 8-9
Static, Global, and Profile-Driven Optimizations............. 8-11
C No Optimization... 8-11
C Static Optimization 8-12

C++ No Optimization 8-12
C++ Static Optimization 8-13
C Global Optimization 8-14
C++ Global Optimization 8-14
Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation 8-15
C++ Virtual Function Optimizations 8-16
Building Self-contained Profiles with gmpf960 8-17
Compression Assisted Virtual Execution (CAVE) 8-22
C Example 8-23
C++ Compression Assisted Virtual Execution (CAVE) ... 8-25
Debugging with gdb960 8-29
Debugging Optimized Code 8-33
Debugging Optimized C++ Code Tutorial 8-37
Writing Flash 8-46
How to Add Benchmarking Routines to Your Code 8-49
Other i960 Processor Choices and the Remote Evaluation Facility 8-50

Appendix A Communicating with MON960 via Serial Port

Appendix B The Saxsoft Webster Browser

Tables

3-1 Cyclone Kit Contents 3-2
3-2 CPU Module Frequency Switch Settings 3-4
3-3 CPU Module Interrupt Switch Settings 3-4
3-4 Chapter Roadmap 3-10
4-1 QUICK*val* i960 Processor Sample Programs 4-2
4-2 i960 Processor Optimization Results 4-25
4-3 Uncompressed Text Sections 4-33
4-4 After Function Compression 4-33
4-5 Improvement 4-33

4-6 Uncompressed Text Sections 4-37
4-7 After Function Compression 4-38
4-8 Improvement 4-38
5-1 QUICK*val* i960 Processor Sample Programs 5-2
5-2 i960 Hx Processor Fault Types and Subtypes 5-18
5-3 i960 Hx Processor Optimization Results 5-25
5-4 Uncompressed Text Sections 5-38
5-5 After Function Compression 5-38
5-6 Improvement 5-38
5-7 Uncompressed Text Sections 5-42
5-8 After Function Compression 5-42
5-9 Improvement 5-42
6-1 QUICK*val* i960 Processor Sample Programs 6-1
6-2 i960 Jx Processor Fault Types and Subtypes 6-20
6-3 i960 Processor Optimization Results 6-27
6-4 Uncompressed Text Sections 6-40
6-5 After Function Compression 6-40
6-6 Improvement 6-41
6-7 Uncompressed Text Sections 6-45
6-8 After Function Compression 6-45
6-9 Improvement 6-45
7-1 QUICK*val* i960 Processor Sample Programs 7-2
7-2 i960 Cx Processor Fault Types and Subtypes 7-15
7-3 i960 Processor Optimization Results 7-24
7-4 Uncompressed Text Sections 7-36
7-5 After Function Compression 7-37
7-6 Improvement 7-37
7-7 Uncompressed Text Sections 7-41
7-8 After Function Compression 7-41
7-9 Improvement 7-41
8-1 QUICK*val* i960 Processor Sample Programs 8-2
8-2 i960 Processor Optimization Results 8-17

8-3 Uncompressed Text Sections 8-24
8-4 After Function Compression 8-25
8-5 Improvement 8-25
8-6 Uncompressed Text Sections 8-29
8-7 After Function Compression 8-29
8-8 Improvement 8-29

Examples

4-1 person Class 4-72
4-2 professor Class 4-73
5-1 person Class 5-69
5-2 professor Class 5-70
6-1 person Class 6-83
6-2 professor Class 6-84
7-1 person Class 7-69
7-2 professor Class 7-70
8-1 person Class 8-40
8-2 professor Class 8-41

Introduction 1

In this highly-charged, fast-paced, cut-throat race to market, you want a world-class support team backing you every step of the way. You want a team that can help you push the limits of performance when you design technically-advanced, high-quality, “second-to-none” products — even when you’re confronted with limited time and limited resources. And that’s what you get when you choose Intel’s i960® processor and CTOOLS development toolset.

The 80960 QUICK*val* kit provides everything necessary to evaluate the i960 processor and CTOOLS software development suite. Also included is an on-line tutorial with development tool and architecture examples to help jump start your evaluation process.

About this Kit

The 80960 QUICK*val* Kit includes:

- Installation CD-ROM with example programs that highlight features of the i960 processor and CTOOLS.
- CTOOLS, a software development toolset that includes the gcc960 compiler, assembler, utilities, gdb960 debugger, and complete documentation.
- The Cyclone evaluation platform, which includes an evaluation base board and an i960 CPU module.

What's New In CTOOLS

Release 6.0 features support for C++. This means that you can now use the enhancements of the C++ language with CTOOLS' powerful development features such as:

- Whole program and profile driven optimizations
- Position independent data, position independent code
- Compression Aided Virtual Execution (CAVE), to reduce the physical memory requirements of ROM-based applications
- Symbolic debug of optimized code using the DWARF debug format

CTOOLS 6.0 also improves support for the i960 Rx processor as follows:

- xlate960 assembly language converter: converts assembly language code from 80960 core processors (e.g., i960 Cx, Jx, and Hx processors) to its CORE0 (e.g., 80960Rx) equivalent. xlate960 performs both instruction and addressing-mode translations.
- Improved Assembler Pseudo-Instruction Support: A number of pseudo-instructions have been added to the CTOOLS assembler to ease migration between processors. These pseudo-ops provide an architecture-independent method for performing some of the more common low-level processing operations.

Finally, CTOOLS 6.0 includes support for the i960 JT and RD processors. All tools support code generation for these new i960 processor family members.

About This Manual

This manual is the only reference that you need for the QUICK*val* kit. Information from the other manuals included in this kit is incorporated into this manual for your convenience. For further details on any topic, please refer to the appropriate manual or contact the 80960 technical support personnel by phone or E-mail as described below.

This manual includes the following information.

Chapter 1, Introduction	Introduces the 80960 QUICK*val* Kit and its features, describes how to receive technical support, and defines the various typeface conventions used in this manual.
Chapter 2, Software Installation	Describes how to install CTOOLS and QUICK*val*.
Chapter 3, Hardware Installation	Describes how to set-up your Cyclone stand- alone or PCI evaluation platform.
Chapter 4, The i960 Rx CPU Example Programs	Describes the example programs provided for evaluating the i960 Rx processor and CTOOLS.
Chapter 5, The i960 Hx CPU Example Programs	Describes the example programs provided for evaluating the i960 Hx processor and CTOOLS.
Chapter 6, The i960 Jx CPU Example Programs	Describes the example programs provided for evaluating the i960 Jx processor and CTOOLS.
Chapter 7, The i960 Cx CPU Example Programs	Describes the example programs provided for evaluating the i960 Cx processor and CTOOLS.
Chapter 8, The i960 Sx CPU Example Programs	Describes the example programs provided for evaluating the i960 Sx processor and CTOOLS.
Appendix A, Communicating with MON960	Describes how to use MON960, the debug monitor resident on the Cyclone evaluation board, to download and execute programs.
Appendix B, The Saxsoft Webster Browser	Tells you how to use the integrated web browser to find files on the World-Wide Web including specification updates, technical support, and many other resources.

Intel Support Services

This QUICK*val* Kit includes 90 days of free software and hardware support by phone or E-mail. To receive this support, register with the 80960 technical support engineers in one of the following ways:

1. Fill out and return the enclosed registration card.
2. Call the 80960 Technical Support Group at 1-800-628-8686 between 7 am and 5 pm Pacific Time or, for non-USA customers, contact your local technical support group.
3. E-mail the registration information to the 80960 Technical Support Group at 960tools@intel.com.

The 80960 Technical Support Group information can also provide information on how to access application support through the FaxBACK, Bulletin Board, Internet, and World Wide Web. Additionally, you can order data sheets, fact sheets, manuals, and application notes by calling the Intel Literature Center at 1-800-548-4725.

Notation Conventions

Bold	Indicates user entry and/or commands.
Italics	Indicates a variable.
`monospace fonts`	Indicates code examples, directories and filenames, and development tool output.
asterisks	On non-Intel company and product names, a trailing asterisk indicates the item is a trademark or registered trademark. Such brands and names are the property of their respective owners.

Software Installation 2

This chapter describes how to install the software components of the QUICK*val* Kit. Installation should be completed in the following order:

- Make sure your host system meets the minimum requirements.
- Install CTOOLS.
- Install the QUICK*val* example programs.

A section on each of these installation steps follows.

System Requirements

The QUICK*val* kit provides a Microsoft Windows* 95/Windows NT*-based tutorial.[1] Your PC system must have:

- An Intel386™, Intel486™, Pentium®, or Pentium Pro processor (or compatible).
- At least 8 Mbytes of RAM.
- Up to 47 Mbytes of available hard disk space.
- For the EP80960BB evaluation platform, one available serial port, and optionally one available parallel port.
- For the PCI80960DP evaluation platform, one empty full length PCI slot, and optionally one available parallel or serial port.
- For the IQ80960Rx evaluation platform, one empty full length PCI slot, and optionally one available serial port.

1. The CTOOLS toolset also supports other host environments including Sun-4*, RS-6000, and HP9000-300/700, and more.

Installing CTOOLS

The CTOOLS R6.0 toolset installation requires approximately 22 MB of available hard disk space. To install an item that was not selected during this installation, re-run the installation and select the item to install.

Installing CTOOLS in Windows 95/Windows NT 4.0

1. Insert the CTOOLS CD-ROM into your CD-ROM drive.
2. From the Windows task bar (Start button), choose Run.
3. Type **`drive:setup`**. For example, if you inserted the CD-ROM into drive E, type:

   `e:\setup`
4. Follow the on-screen instructions. When installing CTOOLS, make sure you make the following selections:
   a. Select gcc960 interface
   b. Install All Components
   - Help Files
   - Library Files
   - GUI
   - .gld Files
   - .ld Files

   c. Select All Library Components
   - KA/SA
   - KB/SB
   - Hx/Jx/Cx
   - RP

Windows NT 3.51 Installation

1. Insert the CTOOLS CD-ROM into your CD-ROM drive.
2. From Windows Program Manager, open the File menu and choose Run.
3. Type **`drive:setup`**. For example, if you inserted the CD-ROM into drive E, type:

   `e:\setup`

4. Follow the on-screen instructions. When installing CTOOLS, make sure you make the following selections:
   a. Select gcc960 interface
   b. Install All Components
   - Help Files
   - Library Files
   - GUI
   - .gld Files
   - .ld Files

   c. Select All Library Components
   - KA/SA
   - KB/SB
   - Hx/Jx/Cx
   - RP
5. Edit the `autoexec.bat` file as indicated in the `autoexec.new` file that was created during installation.

If you made changes to your `autoexec.bat` file, you must re-boot your system now so that the changes made are implemented.

AUTOEXEC.BAT Change Summary

With Windows NT, it is important that the path name for the CTOOLS `BIN` directory is included in the `autoexec.bat` `PATH` definition. If you selected the default directory during installation, verify that these path names have been included.

```
C:\INTEL960\BIN
```

In addition, verify that the following variable is set:

```
SET G960BASE=C:\INTEL960
```

You must now re-boot your system if you made any changes to your `autoexec.bat` file.

NOTE. *If you did not use the default directories on installation, please make sure the G960BASE environment variable is assigned appropriately.*

Installing the QUICK*val* Example Programs

You can install the QUICK*val* example programs for use with Windows 95 or NT 4.0. These files require 25 MB of disk space. Although you can use CTOOLS V6.0 with Windows NT 3.51, QUICK*val* supports Windows 95 and NT 4.0 only.

Installing QUICK*val* Example Programs in Windows 95/ Windows NT 4.0

1. Insert the QUICK*val* CD-ROM into your CD-ROM drive.
2. From the Windows task bar (Start button), choose Run.
3. Type `drive:setup`. For example, if you inserted the CD-ROM into drive E, type:

   ```
   e:\setup
   ```
4. Follow the on-screen instructions.

Where Do You Go From Here?

Now that your software is installed, you are ready to configure you evaluation platform and connect it to the host PC. Chapter 3 tells you how.

Hardware Installation 3

Now that you have installed the required software as directed in Chapter 2, you are ready to connect the Cyclone base board to your host system.

In this chapter, you complete these steps:

- Inspecting the board for any defects.
- Installing the CPU Module (PCI80960DP and EP80960BB evaluation boards only)
- Configuring the processor module switch settings.
- Configuring the Cyclone base board switch settings.
- Connecting the board to the host PC.

This chapter also provides troubleshooting information to help you with any installation problems that may arise. This information appears near the end of this chapter.

Inspecting Your Board

NOTE. *Use the ground strap supplied with this kit and handle electronic components in a static-free area.*

1. Verify that you received every item on the packing list. The kit contents are shown in the table below:

Table 3-1 Cyclone Kit Contents

IQ80960Rx	EP80960BB	PCI80960DP
• Base board with CPU	• Base board	• Base board
• Serial cable	• CPU module	• CPU module
• Connector adapters	• Parallel cable	• Parallel cable
	• Serial cable	• Serial cable
	• Connector adapters	• Connector adapters

2. Visually inspect the board for any damage that may have occurred during shipment. If there are any visible defects, follow the return procedure in the Trouble Sheet to get a replacement. If there is no damage, place the board in a static-free area and take precautions to minimize static electricity (e.g., wear the provided ground strap).

Installing Your CPU module

A CPU module is a smaller board that attaches directly onto the Cyclone EP80960BB and PCI80960DP evaluation base boards. If you are using an IQ80960Rx evaluation board, you may skip this section.

NOTE. *Make sure the power is OFF before you install or remove a CPU module. Also, do not "peel" connectors by lifting one end of the connector before the other. This can bend or break the pins and connectors.*

Line up the alignment holes of the CPU module with the stand-off posts in the center of the base board with the i960 processor facing away from the base board. Press down firmly on the edges of the CPU module making sure that the CPU module remains parallel with the base board at all times. Use the plastic bolts provided to secure the CPU module in place.

Setting Your Base Board Switches

If you are using the Cyclone EP80960BB, PCI80960DP, or IQ80960RP evaluation boards, make sure all four-position DIP switches located at S1 on the base board are set to the OFF position. If you are using the Cyclone IQ80960RD evaluation board, make sure that switch SW1.3 is set to the ON position and SW1.1, SW1.2, and SW1.3 are set to the OFF position. For further details on the function of these switches, refer to the *Cyclone User's Guide*.

Setting Your CPU Module Switches

The sections that follow describe setting the clock frequency for the Cyclone EP80960BB and PCI80960DP evaluation boards. If you are using an IQ80960Rx evaluation board, you may skip to "Installing MON960 on the IQ80960Rx Platform" on page 3-5.

Setting the CPU Module Frequency Switch

Your CPU module may have either one or two four-position DIP switches. The switch located on the CPU module allows you to set the clock frequency. Table 3-2 outlines the processor frequency switch settings.

NOTE. *Do not set a clock frequency that is faster than the installed processor is capable of running. Remove power before changing the switch settings.*

Table 3-2 CPU Module Frequency Switch Settings

Frequency	FREQ2	FREQ1	FREQ0
16 MHz	ON	OFF	ON
20 MHz	ON	OFF	OFF
25 MHz	OFF	ON	ON
33 MHz	OFF	ON	OFF
40 MHz	OFF	OFF	ON
50 MHz	OFF	OFF	OFF

NOTE. *The CPU module frequency switch position 1 is the V_{PP} switch. It is recommended that you leave it OFF.*

Setting Your CPU Module Interrupt Switch

The sections that follow describe setting the CPU interrupt switch for the Cyclone EP80960BB and PCI80960DP evaluation boards. If you are using an IQ80960Rx evaluation board, you may skip to "Installing MON960 on the IQ80960Rx Platform" on page 3-5.

Your CPU module may have a four-position DIP switch located on the lower left corner. This DIP switch is provided to map interrupt sources to the four direct-mapped interrupt inputs. Table 3-3 outlines two settings that are used with the QUICK*val* kit. For further details on the function of these switches, refer to the Cyclone User's Guide.

Table 3-3 CPU Module Interrupt Switch Settings

Interrupt Sources	Position 1	Position 2	Position 3	Position 4
UART/PCI (Default)	ON	OFF	OFF	ON
UART/PPIRQ	ON	OFF	ON	OFF

Installing MON960 on the IQ80960Rx Platform

By default, a preprogrammed 256K Flash device containing IxWorks by Wind River Systems ships on the IQ80960Rx evaluation boards. However, the CTOOLS development tool suite and the 80960 QUICK*val* Kit require MON960 on the IQ80960Rx board in place of IxWorks.

A MON960 Flash device is included with the IQ80960Rx kit in an anti-static box. The following instructions describe how to remove the IxWorks Flash device on the IQ80960Rx evaluation boards and how to install the MON960 Flash device in its place.

Items Needed

- MON960 Flash device that ships in an anti-static box.
- Extraction tool.
- IQ80960Rx board.

WARNING. *When performing the steps below, make sure you wear a ground strap and handle electronic components in a static-free area.*

For help using the extraction tool or inserting the 256K Flash device:

- Technical Support Group
  1-800-628-8686
- 80960 Technical Support Group
  960tools@intel.com

If You Are Using an IQ80960RP (5V) Evaluation Board

1. Power down the computer.
2. Remove the IQ80960RP evaluation board as you would an expansion card in your host system.
3. Locate the 256K Flash device containing IxWorks in socket U4.
4. With the extraction tool, remove the IxWorks Flash device.
5. Replace the 256K MON960 Flash device with the IxWorks Flash device in the anti-static box.

6. Insert the 256K MON960 Flash device in the U4 socket on the IQ80960RP evaluation platform.
7. Insert the IQ80960RP evaluation board in the computer as you would an expansion card.
8. Power up the computer.

If You Are Using an IQ80960RD/RPLV (3.3V) Evaluation Board

1. Power down the computer.
2. Remove the IQ80960RD evaluation board as you would an expansion card in your host system.
3. Locate the 256K Flash device containing IxWorks in socket U10.
4. With the extraction tool, remove the IxWorks Flash device.
5. Replace the 256K MON960 Flash device with the IxWorks Flash device in the anti-static box.
6. Insert the 256K MON960 Flash device in the U10 socket on the IQ80960RD evaluation platform.
7. Insert the IQ80960RD evaluation board in the computer as you would an expansion card.
8. Power up the computer.

With MON960 installed on your IQ80960Rx evaluation platform, you are ready to use the powerful CTOOLS code development tool suite and the QUICK*val* code examples and tutorials.

Connecting the Evaluation Base Board to the Host

Now that the processor module is configured, you are ready to connect the evaluation base board to the host PC.

PCI80960DP

1. Install PCI-SDK platform as you would an expansion card in your host system.
2. Observe the LEDS on the board you should see the following sequence:
   - The red Fail LED located at CR6 on the base board should turn OFF, indicating the processor has passed the self test.
   - The green Run LED located at CR5 should turn ON, indicating that the processor is performing bus cycles.
   - The green LEDs located at CR1 and CR4 should also be lit indicating that the +5 VDC and +3.3 VDC power supplies are within tolerance.

NOTE. *The PCI-SDK platform is a plug-in board and therefore draws power through the PCI bus. No external power source is required.*

Connecting the Serial Communications Cable (Optional)

If you do not want to use the PCI local bus for communicating with the evaluation board, you can use a serial port: however, this method is much slower.

Connect the RS-232 cable from COM1 or COM2 on your host system to J5 on the PCI-SDK platform. Your system has either a DB-9 (9-pin) or DB-25 (25-pin) connector for its RS-232 port. Both connectors are provided.

Connecting the Parallel Communications Cables (Optional)

If you do not want to use the PCI Local Bus to download code, you can use the parallel port, however, this method is much slower.

Connect a 25-pin to 25-pin parallel port cable from an open parallel port on your system to J1 on the PCI-SDK platform.

IQ80960Rx

1. Install the IQ80960Rx platform as you would an expansion card in your host system.
2. Observe the LEDS on the board you should see the following sequence:
   - The red Fail LED located at CR3 on the base board should turn OFF, indicating the processor has passed the self test.
   - The green Run LED located at CR4 should turn ON, indicating that the processor is performing bus cycles.
   - The red user LEDs located at CR1 and CR2 should also be lit indicating the status of the MON960 on-board monitor. MON960 is the monitor software installed on the evaluation board.

NOTE. *The IQ80960Rx platform is a plug-in board and therefore draws power through the PCI bus. No external power source is required. Further, when Windows 95 detects the new adapter board, it asks you if you want to install a driver for that device. Choose the No option.*

Connecting the Serial Communications Cable (Optional)

If you do not want to use the PCI local bus for communicating with the evaluation board, you can use a serial port: however, this method is much slower.

Connect the RS-232 cable from COM1 or COM2 on your host system to J2 on the IQ80960Rx platform. Your system has either a DB-9 (9-pin) or DB-25 (25-pin) connector for its RS-232 port. Both connectors are provided.

EP80960BB

1. Connecting the serial cable

   The serial port is used for communicating and downloading. Connect the RS-232 cable from COM1 or COM2 on your system to J5 on the Cyclone base board. Your system has either a DB-9 (9-pin) or DB-25 (25-pin) connector for its RS-232 port. Both 9-pin and 25-pin connectors are provided.
2. Connecting the parallel cables

   The parallel port can be used to significantly increase download speed. Connect a 25-pin to 25-pin parallel port cable from an open parallel port on your system to J1 on the Cyclone base board.
3. Powering up the board
   - Using the power supply provided with the Cyclone evaluation board, plug the power supply into a standard power socket and the power supply cable into connector J7. The power supply operates with 120 VAC @ 60 Hz.
   - Upon power up, the Fail LED located at CR6 on the base board should turn OFF, indicating the processor has passed its self test. The green Run LED located at CR5 should light, indicating that the processor is performing bus cycles. The green LEDs located at CR1 and CR4 should also be lit indicating that the +5 VDC and +3.3 VDC power supplies are within tolerance.

Where Do You Go From Here?

Congratulations! You have successfully installed one of the most advanced software development toolsets available for the i960 processor. Your development environment is now ready to use for evaluating the i960 architecture and CTOOLS. The table below tells you where to go for tutorials on each i960 architecture.

Table 3-4 Chapter Roadmap

Chapter 4. The i960 Rx CPU Example Programs	Describes the sample programs provided for evaluating the i960 Rx Processor and CTOOLS.
Chapter 5. The i960 Hx CPU Example Programs	Describes the sample programs provided for evaluating the i960 Hx Processor and CTOOLS.
Chapter 6. The i960 Jx CPU Example Programs	Describes the sample programs provided for evaluating the i960 Jx Processor and CTOOLS.
Chapter 7. The i960 Cx CPU Example Programs	Describes the sample programs provided for evaluating the i960 Cx Processor and CTOOLS.
Chapter 8. The i960 Sx CPU Example Programs	Describes the sample programs provided for evaluating the i960 Sx Processor and CTOOLS.

Troubleshooting the PCI-SDK and IQ-SDK Platforms

If the host computer does not boot with the PCI-SDK or IQ-SDK platform installed, do the following:

1. Verify the version of the MON960 debug monitor on your CPU or baseboard is R3.2.3. To do this, check the label on socket U4 on the IQ80960RP board, socket U10 on the IQ80960RD board, or for the PCI80960DP platform socket U5 of the CPU module.

   Also, make sure that you are using gdb960 v6.0 for the PCI communications.
2. If you have the correct version of the monitor, turn off your PC and take out the PCI- or IQ-SDK platform.

3. If you are using the PCI-SDK platform, remove the CPU module from the platform and check that the connector pins are straight. Replace the CPU module.
4. Re-insert the board and make sure that it is firmly seated.
5. Power up your PC.

Verifying the PCI-SDK and IQ-SDK Platforms

mondb and the `-pcil` option make it possible to verify that the SDK platforms are installed correctly. For information on installing MON960 and MONDB on your host, refer to the *MON960 Debug Monitor User's Guide*. Once you have installed MON960/MONDB, At a command prompt, type:

`mondb -pcil`

This option displays the first 64 bytes of PCI configuration space for each PCI device found.

For the PCI-SDK, the following information should be displayed along with other information for other PCI cards which may be installed in your system:

```
MONDB 3.2.3, Fri Feb 28 10:26:19 1998 (WIN_32), Copyright 1997,
Intel Corp.

PCI CONFIGUARTION for DEVICE at BUS=00, DEV=13, FUNC=00
00:  0001113c  02800107  ff000013  00004200
10:  fffbfc00  0000fc81  ffc00000  00000000
20:  00000000  00000000  00000000  00000000
30:  00000000  00000000  00000000  0000010a
       VENDOR_ID = 113c, DEVICE_ID = 0001

Bus# Dev# Fcn# VendId DevId StsReg CmdReg ClsCde Rev Hdr
0    13   0    113c   1     280    107    ff0000 13  0  [*]

[*]  Cyclone PCI Evaluation Target
```

The Vendor ID (VenID) for the PCI-SDK platform is 113c. The bus, device, and function values vary with the installed BIOS. If you see a listing for the Cyclone PCI Evaluation Target, then the PCI-SDK has been installed properly.

For the IQ-SDK, the following information should be displayed along with other information for other PCI cards which may be installed in your system:

```
MONDB 3.2.3, Fri Feb 28 10:26:19 1998 (WIN_32), Copyright 1997,
Intel Corp.

PCI CONFIGUARTION for DEVICE at BUS=00, DEV=13, FUNC=00
00:  09608086  02800106  06040000  00814008
10:  00000000  00000000  00010100  22800000
20:  ff90ff90  ff80ff80  00000000  00000000
30:  00000000  03a2113c  00000000  00030000
       VENDOR_ID = 8086, DEVICE_ID = 0960

PCI CONFIGUARTION for DEVICE at BUS=00, DEV=13, FUNC=01
00:  19608086  02800116  05800000  80804008
10:  fffbf008  00000000  00000000  00000000
20:  00000000  00000000  00000000  03a2113c
30:  00000000  00000000  00000000  0000010a
       VENDOR_ID = 8086, DEVICE_ID = 1960

Bus# Dev# Fcn# VendId DevId StsReg CmdReg ClsCde Rev Hdr
0    13   0    8086   960   280    106    060400 0   81
0    13   1    8086   1960  280    116    058000 0   80 [#]

[#]  Cyclone Rx80960 Evaluation Target
```

The Vendor ID (VenID) for the IQ-SDK platform is 8086. The i960 Rx microprocessor is assigned two Device IDs (DevId). The Address Translation Unit and Messaging Unit have a DevID of 1960, and the i960 Rx microprocessor itself has a DevID of 960.

The bus, device, and function values vary with the installed BIOS. If you see a listing for the Cyclone Rx80960 Evaluation Target, then the IQ-SDK has been installed properly.

If you are interested in the source code for mondb and have installed the MON960 software to the default directory, look for the file `mondb.c` in:

```
c:\intel960\src\mondb\common
```

NOTE. *If you receive the message "NT PCI device driver not found" complete the following. In the CTOOLS installation directory,* `c:\intel960\bin`, *you will find the file* `reg_ntdd.exe`, *which is used to register the device driver* `PCI_WNT.SYS`. *Run* `reg_ntdd.exe` *using the syntax:*

```
reg_ntdd ctools_path NT_SYS_DIR_PATH
```

where `ctools_path` *represents the CTOOLS installation directory, and* `NT_SYS_DIR_PATH` *represents the Windows NT system driver directory. For example:*

```
reg_ntdd c:\intel960\bin c:\winnt
```

or

```
reg_ntdd %G960BASE% %SystemRoot%
```

The i960 Rx CPU Example Programs 4

The i960Rx processor effectively removes the I/O bottleneck in network computing. In today's PC servers the I/O subsystem's performance and bandwidth have not kept pace with today's powerful microprocessors.

The i960 Rx processor is a single-chip intelligent I/O subsystem for PC servers in the enterprise computing environment. I/O subsystems based on the i960 Rx processor will improve the speed at which users access and manipulate text, graphic, video and audio data from PC servers, maximizing the performance of the server. The i960 Rx processor combines the top performance of the i960 Jx processor core with a fully integrated PCI bridge. It will free the host CPU from handling many interrupt-driven I/O processing tasks and allow the host CPU to address secondary PCI devices through the PCI-to-PCI bridge.

Additionally, you can optimize your system’s performance with CTOOLS, which includes a profile-driven compiler that can automatically optimize your code based on its runtime behavior.

Table 4-1 provides descriptions of the tutorials included in the QUICK*val* kit. Each example highlights a feature of the architecture or CTOOLS and provides you with source code that can help shorten your software development cycle.

NOTE. *The 80960Rx QUICKval kit includes tutorials that highlight features of the i960 Rx architecture. If you would like to explore some of the features of the development tools, double-click on the Hx Jx Cx & Sx QUICKval icon in the QUICKval program group and use the tools tutorials provided there that are described in Table 4-1. When you select an architecture to be used for the tutorials, choose 80960Jx. For your base board, choose the PCI80960DP. Note, however, that you should not attempt to use any of the i960 Jx architecture-specific tutorials described in Table 6-1.*

Additionally, you can optimize your system's performance with CTOOLS, which includes a profile-driven compiler that can automatically optimize your code based on its runtime behavior.

The following pages describe the example programs included with this kit. Each example highlights a feature of the architecture or CTOOLS and provides you with source code that can help shorten your software development cycle. Table 4-1 provides descriptions of the tutorials included in the i960 Rx QUICK*val* kit.

Table 4-1 QUICK*val* i960 Processor Sample Programs

Tutorial Description	Source Files
Hello World: Uses simple printf statement to verify system integrity.	`hello.c`: source file `system.c`: system file
Memory Test: Used for system verification of external memory. The programs perform byte, short, or word writes to external memory, and then they check the addresses written for correctness.	`memtst8.c:` 8 bit memory test `memtst16.c`: 16 bit memory test `memtst32.c`: 32 bit memory test `system.c`: system file

continued ☛

Table 4-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
Data Cache: Uses the minimum edit distance algorithm to demonstrate the effectiveness of the on-chip data cache. This example also shows how to enable and disable the data cache and how to configure an area of memory for caching.	`dcache.c`: source file `system.c`: system file
Instruction Cache: Uses a simple loop to demonstrate how to enable and disable the instruction cache. It also highlights the performance advantage obtained when using the on-chip instruction cache.	`loop.c`: source file `system.c`: system file
DMA Controller (i960 Rx): Demonstrates how to set-up the DMA controller, the Primary Address Translation Unit (ATU), the Secondary ATU, and the PCI-to-PCI Bridge Unit.	`rpdma.c`: source file
Messaging Unit: Demonstrates the messaging unit of the i960 Rx processor	`hostcode.c`: source file `rp_code.c`: source file
C Local Optimizations: Shows how to use the C compiler with high levels of static optimization for improved runtime performance.	`chksum.c`, `system.c`: source files
C Global Optimizations: Shows how to use program-wide optimizations of the C compiler for increased performance.	`chksum.c`, `system.c`: source files
C++ Local Optimizations: Shows how to use the C++ compiler with high levels of static optimization for improved runtime performance.	`optimize.cpp`: source file
C++ Global Optimizations: Shows how to use program-wide optimizations of the C++ compiler for increased performance.	`optimize.cpp`: source file

continued ☛

Table 4-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
C++ Virtual Function Optimizations: Shows how a call to a virtual function can be replaced by a direct call to a member function, and, if possible, it may be inlined at the call site. This improves the runtime performance of the code.	`optimize.cpp`: source file
Profiling Lab: Teaches you how to use some of CTOOLS advanced profiling features.	`chksum.c`: source file
Self-Contained Profile: Shows how to create a self-contained profile that captures the program structure and associates it with the program counters from a raw profile. When the source program changes, the global decision making step interpolates or stretches the counters in the self-contained profile to fit the changed program.	`quick.c`: source file
C Cave: Uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.	`ttt.c`: source file
C++ Cave: Shows how to reduce target memory requirements. The text sections of compressed and uncompressed C++ executables are compared. This example also shows how to specify functions for compression.	`cavecpp.cpp`: source file
Linker Directive Language: Provides a hyperlinked manual that describes the linker command options. This tutorial is found in the online help only, not in this manual.	
xlate960 Assembly Language Converter: Shows how to use xlate960 to convert assembly language code written for one i960 processor family member to that of another.	`xlt.s`: source file

continued ☛

Table 4-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
i960 Processor Assembler Pseudo-Instruction Support: Shows how to use the new assembler pseudo-ops.	`pseudop.c`: source file
Debugging with gdb960: Uses the Go Fish card game to teach a few useful debugger commands.	`fish.c`: source file `system.c`: system file
ELF/DWARF Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to set a breakpoint on an in-line function.	`swap.c`: source file
C++ DWARF-2 Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to debug a C++ application.	`cppdwarf.cpp`: source file
Retargeting MON960: Provides steps for retargeting MON960. This tutorial is found in the online help only, not in this manual.	
Writing Flash: Demonstrates how to update the version of MON960 on your evaluation board.	
i960 Rx Processor Initialization Code: Shows the Memory Controller, System Init, and Hardware Init Code.	

System Validation

Hello World

The program `hello.c` is used to verify your software and hardware system integrity. The following steps provide instructions on how to compile, link, download, and execute this program.

1. Power up the host system.
2. Double-click on the **Rx QUICK*val*** icon in the QUICK*val* program group.

3. Configure you hardware.
    - Select the **80960Rx Architecture** tab.
    - Select the i960 Rx processor that you are using.
    - Select the IQ80960Rx tab.
    - Configure the software communication options to match those of your evaluation board.
    - Choose **OK**
4. Choose **Hello World**.
5. Choose **Make** to compile, link, and download the program automatically.
6. Use the gdb960 debugger to execute hello. Type:
   `run`
7. The gdb960 debugger responds by displaying:

```
Hello...Welcome to the 80960Rx QUICKval Kit!
SYSTEM CHECK COMPLETED!!
Now you may proceed with our Example Programs.
Program Exit: 01
(gdb960)
```

8. To exit the debugger, type: **`quit`**

CONGRATULATIONS! You have successfully installed your software and your hardware, compiled a program using gcc960, and downloaded and executed the program on your evaluation board using the gdb960 debugger.

If you received any error messages during this process, refer to “If Something Goes Wrong” on page 4-8.

Memory Test

The programs `memtst8.c`, `memtst16.c`, and `memtst32.c` are used to test the external memory on the Cyclone base board.

Depending on the test that is run, an 8, 16, or 32-bit test is run on an area of memory. The program writes F's and 0's to a memory location and reads the location to verify the integrity of what was written. All three programs are almost identical, with the exception of the casting of the variable *ADDR, which allows you to perform different test types.

NOTE. *Below,* `memtst*.c` *refers to either the byte, short, or word memory test example.*

1. Choose **Memory Test**.
2. Choose a memory test. The options are, **8-bit Memory Test**, **16-bit Memory Test**, or **32-bit Memory Test**.
3. Choose **Make** to compile, link, and download the program automatically.
4. Use the gdb960 debugger to execute memtst. Type:

   `run`
5. For the 8-bit test, `memtst8.c`, the gdb960 debugger responds by displaying:

```
This program will run a 8-bit test on the external memory.

Test to be implemented is byte test.
Starting address = a000dfb0
Ending address = a000ec30

Press enter to begin test with 0's.
Number of errors that occurred is 0.

Begin test for f's.

Press enter to continue.
Number of errors that occurred is 0.

All tests are complete.
Program exited with code 030.
(gdb960)
```

6. Exit the debugger. Type:

   `quit`

If Something Goes Wrong

The following section describes a few actions that may help resolve errors that may have occurred when invoking one of the tools. If you were unable to get the proper response from the gdb960 debugger after executing the above programs and the trouble-shooting hints described below do not help, contact the 80960 Technical Support Group by phone at 1-800-628-8686 or by E-mail at 960tools@intel.com.

MON960 Debug Monitor is Not Responding...

If the red FAIL LED (CR3) on the base board is lit, the monitor may not have booted up correctly. Reset the host PC. If the red FAIL LED remains lit, contact the 80960 Technical Support Group.

Invoking the gcc960 Compiler Resulted in Errors...

The environment must be set-up as described in Chapter 2. If you chose the default directories while installing CTOOLS, verify that the path names `C:\INTEL960\BIN` have been added to your PATH variable and that the following statement is in your `autoexec.bat` file. If you did not install these tools using the default directories, make the appropriate change.

```
SET G960BASE=C:\INTEL960
```

NOTE. *You did not use the default directories on installation, please make sure the G960BASE environment variable is assigned appropriately.*

NOTE. *Don't forget to re-boot your system once you have made any necessary changes to your* `autoexec.bat` *file.*

Invoking the gld960 Linker Resulted in Errors...

Verify that the directory that contains the `hello.c` and `memtst*.c` example programs also now has the object files, `hello.o` and `memtst*.o`. If `hello.o` and `memtst*.o` do not exist, then the gcc960 compiler command did not successfully create an object file. Re-compile `hello.c` and `memtst*.c` to see if an error occurred during the compilation.

If `hello.o` and `memtst*.o` do not exist, make note of the error message and contact the 80960 Technical Support Group.

Invoking the gdb960 Debugger Resulted in Errors...

NOTE. *When using the IQ-SDK evaluation platform, you may specify* `-pci` *for PCI download and PCI communication.*
For a list of all the gdb960 command line options, at a command prompt, enter:
`gdb960 -h | more`

Serial communication error

A serial communication error causes the gdb960 debugger to respond by displaying:

```
HDIL error (10), communication failure
HDIL error (10), communication failure
You can’t do that when your target is ‘exec’
```

Verify that the serial port you are using is the one you specified in the gdb960 command line. Verify that your serial cable is properly connected to the board and to your PC.

Data Cache Tutorial

The i960 Rx processors feature a 2-Kbyte, direct-mapped data cache that is write-through and write-allocate. These processors have a line size of four words. Each line in the cache has a valid bit; to reduce fetch latency on cache misses, each word within a line also has a valid bit.

The purpose of the `dcache.c` program is to show the performance advantage that can be obtained by the use of the data cache on the i960 Rx microprocessor.

This example uses the Minimum Edit Distance (MED) algorithm in order to show the effectiveness of using the data cache. The MED algorithm finds the minimum number of edit steps required to change one string into another.

This example is a real world example of using the data cache. This algorithm maintains a cost matrix to determine which change to the string being edited would incur the least cost. The cost matrix is a 2-D array [1..n][1..m], where n and m are the sizes of the two strings.

The algorithm really shows the speed of the data cache due to three reads for each write to the cost matrix. The algorithm reads from the cache to determine which step to take next, then writes its choice in the cost matrix. Since the writes to the data cache are write-through, there is no improvement for writes to the data cache. The Write-Through feature maintains coherency between the data cache and external memory.

The source code includes system files, `system.c` and `system.h`, that includes a macro and an assembly function that simplifies issuing data cache control instructions.

Also, the example shows how to define an area of memory to make data cacheable by using the Logical Memory Configuration (LMCON) registers. The address of the area to make cacheable is programmed into the Logical Memory Address Register (LMADR). The mask is programmed into the Logical Memory Mask Register (LMMR).

1. Choose **Data Cache**.
2. Choose **Qv Code**.
3. Scroll through the `dcache.c` code to see the calls to the macro, `dcctl_contrl`.
4. Open and scroll through the `system.h` and `system.c` code to see the macro and assembly function, `dcctl_control` and `i960_dcctl`.
5. Choose Make to compile, link, and download the program automatically.
6. Use the gdb960 debugger to execute `dcache`. Type:

   `run`

The debugger responds by displaying:

```
Minimum Edit Distance algorithm makes reads from the data cache.
This routine will determine how many steps are needed to convert:
StringA:  80960 QUICKval EvalKit
TO StringB:  i960(R) HxJxCxSx & Kx
Starting timed routine with data cache on ...
RESULT: 18 moves are required to convert string A to string B
Elapsed Time On = 0.002956 seconds
Elapsed Time for routine with data cache off ...
RESULT: 18 moves are required to convert string A to string B.
Elapsed Time Off = 0.003391 seconds
IMPROVEMENT: 12.8 percent
(gdb960)
```

7. Type: `quit`
8. Select **Results**.

Instruction Cache Tutorial

The i960 Rx processor comes equipped with 4 KB of two-way set-associative instruction cache. The instruction cache boosts your application’s performance by reducing the number of instruction fetches from external memory. The cache provides fast execution of code and loops of code in the cache.

The loop.c program demonstrates the performance boost obtained by running a loop completely within versus outside of the instruction cache.

The source code includes system files, `system.c` and `system.h` that includes a macro and an assembly function that simplifies issuing instruction cache control instructions.

1. Choose **Instruction Cache**.
2. Choose **Instruction Cache**.
3. Choose **Qv Code.**
4. Scroll through the `loop.c` code to see the calls to the macro, `icache_control`.
5. Open and scroll through the `system.h` and `system.c` code to see the macro and assembly function, `icache_control` and `i960_icctl`.

6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `loop`. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/loop
   Simple loop timed with instruction cache off ...
   Elapsed Time Off = 4.200 seconds
   Simple loop timed with instruction cache on ...
   Elapsed Time On = 2.076 seconds
   IMPROVEMENT : 50.6 percent
   Program exited with code 01
   (gdb960)
   ```
8. Type: **`quit`**
9. Select **Results**.

DMA Example

The purpose of this example is to demonstrate how to set-up the DMA controller, the Primary Address Translation Unit (ATU), the Secondary ATU, and the PCI-To-PCI Bridge Unit.

How Does This Program Work?

A DMA is initiated on Channel 2 on the Secondary PCI bus. The Chain Descriptor for the DMA is set-up such that the destination of the DMA is the first Index Register of the Messaging Unit. (Note that this is a basic example that shows how to use several features of the i960 Rx microprocessor.)

In the Chain Descriptor, the DMA source is some 80960 local memory value (0x12345678), and the type of transaction is PCI WRITE. This means the DMA writes the value 0x12345678 to the first Index Register.

The DMA starts on the Secondary Bus from Channel 2. The PCI-To-PCI Bridge claims the transaction since the destination PCI address is on the primary bus. The PCI-To-PCI Bridge forwards the transaction to the primary bus.

Positive Address Decoding

In this example, the PCI-To-PCI Bridge is set up for Positive Address Decoding. With Positive Address Decoding the PCI addresses within the address range of the Secondary Memory Base Register (SMBR) and the Secondary Memory Limit Register (SMLR) of the PCI-To-PCI Bridge Unit are forwarded through the bridge. Inverse Decoding is disabled.

Writing the Destination

The destination address of the DMA is the Primary Inbound ATU Base Address (PIABAR) + INDEXREG_OFFSET (0x50). This means the DMA is the first Index Register in the Messaging Unit. The DMA is then verified by checking the 80960 local bus address of the first Index Register. The Index Registers on the 80960 local bus can be found at the address:

Primary Inbound ATU Translate Value Register (PIATVR) + INDEXREG_OFFSET.

What Does This Example Do?

This example configures the following parts of the 80960Rx:

1. Primary ATU
2. Secondary ATU
3. PCI-to-PCI Bridge unit
4. DMA Controller for Channel 2
5. Chain Descriptor

Note that the Chain descriptors must be aligned on an 8-word boundary.

DMA Tutorial

1. Choose **Rx DMA**.
2. Choose **Make** to compile, link, and download the program automatically.
3. At the (gdb960) prompt, enter:

   `run`

   Notice the destination value changes from 0 to 0x12345678.
4. Enter **`quit`**.

NOTE. *You should read “How Does This Program Work” before running the tutorial to familiarize yourself with the concepts of the example. Note also that the code rpdma.c is fully commented. Look at the source code for further explanations.*

Messaging Unit Example

This example demonstrates the messaging unit of the i960 Rx microprocessor. The host processor, for example a Pentium processor, and the 80960Rx can pass messages to one another via the messaging unit.

Example Description

This example demonstrates the messaging unit of the i960 Rx microprocessor. The host processor, for example a Pentium processor, and the Rx can pass messages to one another via the messaging unit.

In this example, such messages include:

- From Host to Rx - "toggle your LEDs"
- From Rx to Host - "show various graphics applications"
- Synchronous counting by the Host and Rx

The message passing is done by using various inbound and outbound registers of the Rx.

Address Translation

To access the inbound and outbound registers of the messaging unit, a device driver is used. For Windows 95, the device driver is PCI_W95.VXD. For Windows NT 4.0, the device driver is PCI_WNT.SYS.

The device driver takes a physical address of the 80960Rx’s registers obtained from the system BIOS and returns a usable linear address. The host processor can then use the linear address in a program to talk to the Rx through its registers.

Example Structure

The example is broken into two programs: `hostcode.c` runs on the host processor, `rp_code.c` runs on the i960 Rx processor.

Hostcode.c

`Hostcode.c` begins by printing some system information about the host computer. Depending upon which operating system the host system is running, a device driver is loaded for physical address translation. The program scans the PCI bus for a PLX chip or an Rx chip. Since the Rx chip has two functions, the Address Translation Unit (ATU) and the PCI-To-PCI Bridge, `hostcode.c` searches for both functions.

Note that the bridge function is not found under Windows NT because the operating system hides this information.

If the ATU of i960 Rx processor is found, the System BIOS is read to determine the physical address of the Primary Inbound ATU Base Address Register. The physical address is passed to the device driver, which returns a linear address.

`hostcode.c` translates this virtual address to determine the addresses of the Inbound Message Registers and Outbound Message Register 0.

NOTE. *In this example, Inbound Message Registers 0 and 1 are used; however, MON960 requires exclusive use of Inbound Message Register 1. When developing your code, it is wise not to use the registers used by MON960. Please refer to the MON960 Debug Monitor Release Notes for more information.*

The program displays a menu of possible messages to send to the Rx. The messages are binary codes that are delivered to the Rx via Inbound Message Register 0. The Rx sends binary messages to the host via Outbound Message Register 0. When the Rx and Host processor count together, Inbound Message Register 1 is additionally used.

Rp_code.c

The Rx enters a while loop that it stays in until the user sends a "quit" message via the host. The Rx looks for messages from the Inbound Message Register 0. The messages range from 0x1 to 0x9.

The Rx communicates back to the host by sending messages through the Outbound Message Register 0. Also, the Rx uses the Inbound Message Register 1 in order to synchronize the counting between processors for the counting message.

Messaging Unit Tutorial

The example spawns two DOS windows: one for the host program and one for the i960 Rx program.

NOTE. In order for this example to work, the Rx must be ready to receive messages from the host. See below how to set-up the host to send messages and set-up the Rx to receive messages.

Host Program DOS Window

Follow the instructions on the screen. The `hostcode.c` program explains what is happening. Press **`<Enter>`** repeatedly until the Rx Message Options Menu appears.

i960 Rx Processor Program Command Prompt Window

1. Choose **Messaging**.
2. Choose **Make** to compile, link, and download the program automatically.
3. At the (gdb960) prompt, enter:

   `run`

   The program responds with:

```
"I am waiting for the Host Processor to talk with me .... ".
```

   The i960 Rx microprocessor is now ready to send and receive messages with the host processor.

4. Use the Rx Message Options menu in the Host Program DOS window to communicate with the Rx. Try all the messages and see the result.
   - In some cases, the Rx asks a question. You must supply the Rx with an answer in order to continue the example.
   - Click on the Rx Program DOS window in order to give it focus to respond to the 80960Rx's questions.
   - Remember to click on the Host Program DOS window to interact with the Rx Message Option menu.

General Notes Concerning the Random Scene Generator (Message 7)

- Press **`<Enter>`** to make the random scene generator show pictures faster.
- Pressing **`<Esc>`** to end the scene generator.

Issues with the random scene generator (Message 7):

Windows 95: The random scene generator starts out running in a DOS window, and its color palette is not correct.

1. Press **`<Alt>+<Enter>`**, and the scenes zoom to full screen size and the color palette corrects itself.
2. You can then press **`<Alt>+<Enter>`** again to toggle back to the DOS window; however, this time the color is correct.

Windows NT: After the i960 Rx processor sends the message back to the host saying it wants to see the random scene generator, nothing happens. This is because the Host Program DOS window does not have focus.

1. Click on the Host Program DOS window.

   The random scene generator then starts; however, it makes itself full-screen. Under Windows NT, let the scene generator run full-screen.
2. After viewing the scenes, press **`<Esc>`**. To toggle to the DOS window, press **`<Alt>+<Enter>`**.

Program Termination

To exit the example:

1. Send message 9 , "QUIT", from the Host Program DOS window.
   The Rx program also terminates.
2. In the Rx Program DOS window enter
   `quit`
   at the (gdb960) prompt

NOTE. The code `hostcode.c` and `rp_code.c` are fully documented. Look at the code for further explanations.

Static, Global, and Profile-Driven Optimizations

Optimizing compilers provide you with a means of developing high performance code without detailed knowledge of the architecture. Engineers who understand the features of the i960 architecture developed gcc960 to provide optimizations that take full advantage of the i960 processor. In general, optimizing compilation takes more time and may require more memory for large functions. However, the benefit in runtime performance is well worth it.

There are several levels of optimization available. Typically, low levels of optimizations are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once your application is functioning properly, you can increase its runtime performance by using a higher level of optimization.

Release 5.0 and later of the development tools support the ELF object module format and DWARF version 2.0 debug information format. The new format enables more accurate mapping between source and object code at higher optimization levels and ease debugging of production code.

The C optimization example uses a program called `chksum.c`. The C++ examples use a program called `optimize.cpp`

C No Optimization

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.
4. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 36.903947 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**

C Static Optimization

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

Use the following commands to compile the `chksum.c` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.
4. Choose **Make -O4** to compile with optimizations, link, and download the program automatically.

5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 5.1887223 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ No Optimization

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.
5. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 10.2445 seconds.
   Program exited normally
   ```
7. Type: **`quit`**

C++ Static Optimization

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

Use the following commands to compile the `optimize.cpp` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.
5. Choose **Make -O4** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 6.4918 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

C Global Optimization

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

Use the following commands to compile the `chksum.c` with program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C Global Optimizations**.
4. Choose **Make +O5** to compile with optimizations, link, and download the program automatically.

5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 4.154432 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ Global Optimization

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

Use the following commands to compile the `optimize.cpp` program using the program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Global Optimizations**.
5. Choose **Make+05** to compile with optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 6.4584 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation

An 88% improvement in C code performance is significant, but there is another level of optimization that is uniquely available through Intel's CTOOLS compilers: profile-driven optimization. This two-pass compilation procedure allows the compiler to make optimizations based on runtime behavior as well as the static information used by conventional 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. Standard (i.e., non-program-wide) optimizations are referred to as 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 before the end of the linking phase.

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.

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Profiling Lab**.
4. Follow the **Profiling Tutorial** link in the online help.

Using profile-driven optimization, an increase in runtime performance of 20% is obtained. The average 80960 application can expect to gain 15 to 30% performance improvement through the use of this technology. This boost in performance is available to you without any further investment in hardware.

C++ Virtual Function Optimizations

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, however, 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.

Use the following commands to compile the `optimize.cpp` program.

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Virtual Opts**.
5. Choose **Make -NoVOpt** to compile without virtual function optimizations, link, and download the program automatically.

6. To execute the program, type:

   `run`

   The debugger responds by displaying:

```
Starting program: C:\QUICKVAL/optimize
Now starting C++ routine ...
Time for C++ routine was 6.4584 seconds.
Program exited normally
```

7. Type: **`quit`**
8. Choose **Make -VOpt** to compile with virtual function optimizations, link, and download the program automatically.
9. To execute the program, type:

   `run`

   The debugger responds by displaying:

```
Starting program: C:\QUICKVAL/optimize
Now starting C++ routine ...
Time for C++ routine was 5.6307 seconds.
Program exited normally
```

10. Type: **`quit`**
11. Choose **Results**.

The virtual function optimizations yielded a 12.8% improvement.

Note the runtime performance at each optimization level as shown below.

Table 4-2 i960 Processor Optimization Results

Optimization Level	C Execution Time	C++ Execution Time
no optimization (-O0)	36.903947 seconds	10.2445 seconds
maximum static (-O4)	5.1887223 seconds	6.4918 seconds
global optimization	4.154432 seconds	6.4584 seconds
profile-driven	4.153173seconds	NA
Virtual Function Optimization	NA	5.6307 seconds

Building Self-contained Profiles with gmpf960

A *raw* profile contains program counters that record 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 has a very short useful life. When changes are made in the 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 used to optimize a program 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.

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Self-Contained**.
4. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
5. Specify the program database directory.

   The PDB can be specified by setting the environment variable `G960PDB`.

For example, if you chose the default directory during installation, enter:

```
SET G960PDB=C:\quickval\prof_lab\lab_pdb
```

Or, specify the PDB at compiler invocation time with the `Z`*`dir`* option, as shown in the example below.

```
gcc960  -Zmypdb  foo.o
```

6. Compile for profile instrumentation.

   Insert profile instrumentation into *quick* so that when the linked program is executed, a profile can be collected. Type:

   ```
   gcc960 -Fcoff  -T{Link-dir} -A{arch} -fdb
   -gcdm,subst=:*+fprof -o quick quick.c
   ```

   The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyrx` specifies `mcyrx.gld`
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-gcdm,subst=:*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only eligible modules not linked in from libraries.
`+fprof`	causes generation of profile instrumentation
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

7. Collect 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. Type:

```
gdb960 -t mon960 -b 115200 -r com1 -D lpt1 quick
```

The options in this gdb960 compiler command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`quick`	the executable file

8. Use the gdb960 debugger to execute `quick`. Enter:

   `run`

9. Exit the debugger. Enter:

   `quit`

10. Enter the command:

    `gmpf960 -spf quick.pf default.pf`

    The options in this gmpf960 compiler command are:

`-spf`	causes a self-contained profile, `quick.pf`, to be produced as output
`default.pf`	The input profile.

11. Recompile the `quick.c` source code using the profiling information obtained by the instrumentation. Type:

    `gcc960 -Fcoff -T`*`{Link-dir}`*` -A`*`{arch}`*` -fdb`
    `-gcdm,iprof=quick.pf -o quick quick.c`

    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyrx` specifies `mcyrx.gld`
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=quick.pf`	This supplies a profile file `quick.pf` to the global decision making and optimization step.

`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

12. Change the control structure of `quick.c.`

    Edit `quick.c`. Find the procedure called QUICK. In this procedure, there is a control structure:

    ```
    for(i = 2; i <= SORTELEMENTS; i+=1)
    {
        (LOGIC)
    }
    ```

    Change the control structure to:

    ```
    i = 2;
    while (i <= SORTELEMENTS)
    {
        (LOGIC)
        i+=1;
    }
    ```

13. Compile the new `quick.c` using the interpolated profile. Type:

    `gcc960 -Fcoff -T`*`{Link-dir}`*`-A`*`{arch}`*`-fdb`
    `-gcdm,iprof=quick.pf -o quick quick.c`

    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyrx` specifies `mcyrx.gld`
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=quick.pf`	This supplies a profile file `quick.pf` to the global decision making and optimization step.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

Notice that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`.

NOTE. *The beauty of this example is that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`, not the results of running this example.*

Compression Assisted Virtual Execution (CAVE)

This CTOOLS feature allows non-critical parts of an application's machine code to be stored in memory in compressed form resulting in reduced target memory requirements. The code is expanded into native machine code on demand for 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 of 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, 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, that 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.

Note that due to the overhead of decompressing code at runtime, only non-performance critical code should be secondary functions, such as error handling code or initialization code. You can use runtime profile information generated by gcov960 to aid in selecting the set of secondary functions.

This example uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.

For the sake of demonstration, we compress performance-critical code in the tic-tac-toe program. The purpose of this example is to show the reduced text section of the executable, not demonstrate run times.

C Example

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Compiler**.
2. Choose **C Cave**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Use the gcc960 `mcave` option or `#pragma cave` to designate the specified functions as secondary. In the tic-tac-toe example, `ttt.c`, the following `#pragma` has been added:

   ```
   #pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)
   ```

   where `Initialize, Winner, Other, Play, Evaluate, Best_Move, Describe, Move,` and `Game` are all functions to be compressed.

5. Edit `ttt.c`. Make sure the `#pragma cave` program line is commented out:

   ```
   /*#pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)*/
   ```

6. Compile the tic-tac-toe program. Enter:

`gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c`

   The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyrx` specifies mcyrx.gld.
`-o ttt`	names the executable file ttt
`ttt.c`	input file

7. Check the text section size of the uncompressed program. Enter:

   `gsize960 ttt`

   The option in this command is:

`ttt`	name of the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.

8. Edit `ttt.c`. Make sure the `#pragma cave` program line is uncommented:

   ```
   #pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)
   ```

9. Compile the tic-tac-toe program with the pragma program line. Enter:

   `gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c`

   The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyrx` specifies mcyrx.gld.

`-o ttt`	names the executable file ttt
`ttt.c`	input file

10. Check the text section size of the compressed program. Enter:

    `gsize960 ttt`

    The option in this command is:

`ttt`	executable file

    The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 4-3 Uncompressed Text Sections

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	33,764	32,944	32,768	32,976	31,600

Table 4-4 After Function Compression

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	31,908	30,832	30,816	30,832	29,648
Cave Section	1,818	1,770	1,746	1,800	1,776
Total	33,726	32,602	32,562	32,632	31,424

Table 4-5 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
0.1%	1.0 %	0.6 %	1.0 %	0.6 %

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

C++ Compression Assisted Virtual Execution (CAVE)

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Compiler**.
2. Choose **C++ Cave**.
3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Use the *gcc960* `mcave` option or `#pragma cave` designate the specified functions as secondary. In the C++ example, `cavecpp.cpp`, the following `#pragma` has been added:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

   where `initSetName`, `initSetDept`, `initSetGpa`, `initSetNumPubs`, `isOutstanding`, `printName`, and `InitializeRecords` are all functions to be compressed, i.e., all functions are secondary functions. All other functions of the program are primary functions.

   The primary set should contain performance-critical functions that are not to be affected by the CAVE mechanism; the secondary set is subject to compression. Secondary functions are compressed by the linker and reside in memory in compressed form.

   The C++ compiler behaves in essentially the same manner as the C compiler when the mcave or Gcave options are used **-** generating all functions in the compilation unit for which this option is in effect as secondary.

   A user typically designates a single function as secondary through the use of `pragma cave`. The following statement for example designates the function max as secondary.

   ```
   # pragma cave max
   ```

   However in C++ overloaded functions have the same name. Member functions of two different classes are also allowed to have the same name and these member functions can in turn have the same name as a function with file scope.

When a user specifies a function as secondary through the use of `pragma cave`, the C++ compiler treats all functions with this name as secondary. To illustrate, consider the following example:

```
# ifdef PRAGMA
# pragma cave max
# endif

int max(int a, int b)
{
return a > b ? a : b;
}

float max(float a, float b)
{
return a > b ? a : b;
}

class Tclass1 {
int a, b;
public:
int max();
};

int Tclass1::max()
{
return a > b ? a : b;
}

class Tclass2 {
float a, b;
public:
float max();
};


float Tclass2::max()
{
return a > b ? a : b;
}

Tclass1 t1;
Tclass2 t2;
```

The Compiler treats all the following functions as secondary.

```
int max(int, int);
float max(float, float);
int Tclass1::max();
float Tclass2::max();
```

5. Choose **Qv Code**. Edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is commented out:

   ```
   //#pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

6. Compile the C++ program. Enter:

   `gcc960 -A{`*`arch`*`} -Felf -T{`*`Link-dir`*`} -stdlibcpp -o cavecpp cavecpp.cpp`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A{`*`arch`*`}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{`*`Link-dir`*`}`	specifies the linker directive file. For example, `-Tmcyrx` specifies `mcyrx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `cavecpp`
`cavecpp.cpp`	input file

7. Check the text section size of the uncompressed program. Enter:

   ```
   gsize960 cavecpp
   ```

   The option in this command is:

`cavecpp`	specifies the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.

8. Choose **Qv Code** and edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is uncommented:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

9. Compile the C++ program with the pragma program line. Enter:

   `gcc960 -A{`*`arch`*`} -Felf -T{`*`Link-dir`*`} -stdlibcpp`
   `-o cavecpp cavecpp.cpp`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, -AHD specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyrx` specifies `mcyrx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `ttt`
`cavecpp.cpp`	specifies the input file

10. Check the text section size of the compressed program. Enter:

    `gsize960 cavecpp`

    The option in this command is:

`cavecpp`	executable file

The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 4-6 Uncompressed Text Sections

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	89,788	84,196	83,512	84,196	81,764

Table 4-7 After Function Compression

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	87,612	81,892	81,512	81,892	79,796
Cave Section	1,920	1,546	1,514	1,546	1,512
Total	89,532	83,438	83,026	83,438	81,308

Table 4-8 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
1%	1%	1%	1%	1%

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

XLATE960 Tutorial

This tutorial shows how to use the xlate960 utility provided with CTOOLS release 6.0. xlate960 is the 80960 translation utility that generates i960 Rx-compatible code sequences to replace instructions and addressing modes that are only available on other i960 processors.

1. If you are using the **Hx Jx Cx & Sx QUICK*val*** software, Choose Linker and Utilities. If using the **Rx QUICK*val*** software, this step is not necessary.
2. Choose **xlate960 Tutorial**
3. Choose **Qv Code**.

The assembly file, `xlt.s`, is loaded into the editor shown on your screen. This program is a contrived example that really does not do any useful work. It was written to help demonstrate how to migrate assembly code to the Rx Strategy. This program supports i960 processor functionality that is not available when using the Rx Strategy. `xlt.s` has two complex addressing modes:

- indexed
- ip-relative

and three classes of instructions

- arithmetic (scanbit)
- triple word /quad word instructions (quad word move)
- integer/overflow behavior (addi)

that demonstrate behavior not supported under the Rx Strategy.

If `xlt.s` were compiled with the `-AJF` architecture option, there would be no compilation errors. However, if `xlt.s` were compiled with the `-ARD` or `-ARP` architecture options, compilation errors would stop the build. The offending instructions and addressing modes would have to be translated to Rx Strategy compatible instructions and addressing modes. xlate960 can do this automatically for you, with only a little user interaction.

Looking at the xlt.s File

To understand what the `xlt.s` file is doing, please review the `xlt.s` file in an editor. The lines that violate the Rx Strategy are detailed below:

Line 83: `bx 24(ip)`

IP-relative addressing is not available when specifying an i960 Rx processor-based target.

The xlate960 utility replaces the above `bx 24(ip)` operation with the following instruction sequence that duplicates the functionality of the `bx 24(ip)` operation:

```
#xlate-beginbx 24(ip)
#xlate-err“Fill in register for E0”
#xlate-warn“Verify use of local labels ‘8’ and ‘9’”
#xlate-err“Verify that register g14 can be clobbered”
bal .+4
8:  lda         24+9f-8b(g14),E0; 9:
bx  (E0)
#xlate-end
```

The line beginning with `#xlate-begin` marks the start of the code added by the xlate960 utility to replace the `bx 24(ip)` instruction, and the line beginning with `#xlate-end` marks the end of the code. All translation errors are marked with a comment of the form `#xlate-err`. More subtle translation incompatibilities are flagged with a `#xlate-warn` comment.

Above, three non-comment lines were added to replace the `bx 24(ip)` instruction. However, based on the suggestions of the comments, these lines may require manual editing. Manual translation is demonstrated later in the tutorial.

Line 126: `st r9,_VariableArray[r11*8]`

Indexed addressing modes are not available when specifying an i960 Rx processor-based target. The xlate960 utility replaces the above `st r9,_VariableArray[r11*8]` operation with the following instruction sequence that duplicates the functionality of the `st r9,_VariableArray[r11*8]` operation:

```
#xlate-beginst r9,_VariableArray[r11*8]
#xlate-err“Fill in register for E1”
shlo3,rll,E1
st  r9,_VariableArray(E1)
#xlate-end
```

Two instructions were inserted by the xlate960 utility to replace the `st r9,_VariableArray[r11*8]` operation. Also, as before, it may be necessary to edit these two instructions to complete code migration.

Line 160: `scanbit r9,r8`

The `scanbit` instruction is not guaranteed to set the condition code with the Rx Strategy.

Line 208:	`addi r10,r11,r8`

The `addi` instruction is not supported with the i960 Rx architectures.

Line 239:	`movq r8,g8`

The `movq` instruction is not supported with the i960 Rx architectures. The instruction sequence inserted by xlate960 to replace the `movq` instruction does not test for unaligned or overlapping registers. It is left to the programmer to ensure that the registers used do not overlap and that the registers are aligned. The programmer can do this by making sure the code is compatible with existing i960 processors before running the code through xlate960. The programmer should not experience unaligned or overlapping registers if the code has been assembled for another processor prior to running it through xlate960.

Using xlate960

To prove that `xlt.s` compiles unaltered as code designed for earlier i960 processors, complete the following steps:

1. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
2. Enter the following command in the Command Prompt window provided:

   ```
   gcc960 -AJF -Fcoff -Tmcyjx -o xlt xlt.s
   ```

   The options in this command are:

`-AJF`	sets the target architecture for the compiler.
`-Fcoff`	sets the object file type as COFF.
`-Tmcyjx`	uses the linker directive file for the Jx architecture.
`-o xlt`	sets the object file name as `xlt` (optional).
`xlt.s`	specifies the input source file.

3. To run the program, enter:

   ```
   gdb960 -t mon960 -b {baudrate} -r {comport} -pci xlt
   ```

The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`xlt`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out:

   - the value of register r11 before and after the ip-relative branch.
   - the value of the displacement in the index with displacement addressing mode.
   - the condition code before and after the `scanbit` instruction.
   - the condition code before and after the `add` instruction.
   - the result of performing the `movq` instruction.

NOTE. *The significance of this example is not in the results of the running program, but in the code translation performed by xlate960 in the next few steps.*

5. At the (gdb960) prompt, enter: **`quit`**

To prove that `xlt.s` does **not** compile unaltered using the Rx Strategy, complete the following steps:

6. Enter the following command in the Command Prompt window provided:

   ```
   gcc960 -AR{P|D} -Fcoff -Tmcyrx -o xlt xlt.s
   ```

`-AR{P|D}`	sets the target architecture for the compiler. Since you are compiling for the Rx Strategy, use the available i960 Rx architecture options `-ARP` or `-ARD`.
`-Fcoff`	sets the object file type as COFF.
`-Tmcyrx`	uses the linker directive file for the i960 Rx architecture.
`-o xlt`	sets the object file name as `xlt` (optional).
`xlt.s`	specifies the input source file.

   There are errors during the compilation. The errors are:

   ```
   xlt.s:83: Register is not in target architecture:
   "(ip)".
   xlt.s:126: indexed addressing mode not available
   xlt.s:208: Opcode is not in target architecture:
   "addi".
   xlt.s:239: Opcode is not in target architecture:
   "movq".
   ```

   These errors must be resolved before the program compiles using the `-ARD` or `-ARP` architecture flags.

xlate960 generates Rx-compatible code sequences to replace those instructions and addressing modes that appear in the JF processor causing errors above.

7. Enter the following command in the Command Prompt window provided:

   ```
   xlate960 xlt.s
   ```

   The previous command converts instructions in `xlt.s` to Rx-compliant instructions, placing the output into the file `xlt.xlt`.

   The output in the Command Prompt window is:

   ```
   C:\INTEL960\BIN\XLATE960.EXE:  Output file
   'xlt.xlt' requires further manual translation.
   ```

This message above means you must edit the output file `xlt.xlt` to finish the translation to i960 Rx-compliant code.

You Are now ready to Edit the `xlt.xlt` file.

8. Open `xlt.xlt` in an editor.

   The output file produced by xlate960 is identical to the input file except for the instances where translation occurred. Each instruction that was translated is replaced with a sequence of the following format in the output file:

   ```
   #xlate-beginoriginal instruction
   <translation errors or warnings, marked by xlate-err
   or xlate-warn>
   <translation routine>
   #xlate-end
   ```

9. Find the translation points in `xlt.xlt` by searching the file for `#xlate-begin` flags.

   There are five translation points in the file.

   At the first translation point beginning on line 84 of `xlt.xlt`, note two `#xlate-err` translation errors and the `#xlate-warn` translation warning. The first translation error is:

   ```
       #xlate-err  "Fill in register for E0"
   ```

   The following two instructions are found on lines 89 and 90 of the translation routine:

   ```
   lda 24+9f-8b(g14),E0; 9:
   bx  (E0)
   ```

   Fill in a register that can be used for the place holder E0 that does not affect the program logic (i.e., choose a register that is not being used). In our example, it is all right to use register r13. So, edit the code and change E0 to r13:

   ```
   lda 24+9f-8b(g14),r13; 9:
   bx  (r13)
   ```

   The next translation error is:

   ```
   #xlate-err"Verify that register g14 can be clobbered"
   ```

   The translation routine uses register `g14` on line 89. Since `g14` can be overwritten, it does not need to be changed. The translation warning reported is:

   ```
   #xlate-warn"Verify use of local labels '8' and '9' "
   ```

The translation routine uses the local labels '8' and '9'. Since they do not conflict with other local labels used in the program, no change is needed.

Lastly, the original program, `xlt.s`, made a branch ahead by 24 plus the contents of the ip-register. The translation routine discredits the displacement number due to added instructions, and it is now necessary to change the displacement to 28.

10. So, edit the translation routine and change 24 to 28 to maintain the correct logic:

    ```
    lda 28+9f-8b(g14),r13; 9:
    ```

11. Find the next translation point; it is the following:

    ```
    #xlate-beginst r9,_VariableArray[r11*8]
    ```

    The translation error reported is:

    ```
    #xlate-err"Fill in register for E1"
    ```

    Like previously, all that is necessary is to use a register for the placeholder E1 that is not used and that does not affect the logic of the program. This time, register r15 is all right.

12. Edit the code on lines 138 and 139 from:

    ```
    shlo3,r11,E1
    st  r9,_VariableArray(E1)
    ```

    to the following:

    ```
    shlo3,r11,r15
    st  r9,_VariableArray(r15)
    ```

13. In order for the program to print the correct displacement after the translation, the code needs a little more editing. On line 128 of the `xlt.xlt` file, the following code segment begins:

    ```
    lda LC9,g0
    mov r15,g1
    callj_printf
    mov g0,g4
    ```

    Move this code segment to line 139 of the file. The segment thus occupies lines 139 through 142. Make sure to delete the code segment from lines 128 through 131.

14. Translation point three concerning the scanbit instruction had no translation warnings or errors.

15. View translation point four; it starts with the following:

    `#xlate-beginaddi r10,r11,r8`

    The translation warning for this translation routine is:

    `#xlate-warn"Loss of faulting behavior"`

    and the translation routine is:

    `addor10,r11,r8`

    xlate960 uses the xlate-warn comment lines to indicate instances where the translated code has subtle differences from the original code. Here, the `addo` instruction differs from the `addi` instruction because it does not fault when an overflow is generated. If overflow behavior is important to the program's operation, you would need to rewrite the code to manually check for an overflow condition.

16. Finally, view translation point five; it starts with the following:

    `#xlate-beginmovq r8,g8`

    The translation warning for this translation routine is:

    ```
    #xlate-warn"Does not test for unaligned or
    overlapping registers"
    ```

    and the translation routine is:

    ```
    mov r8,g8
    mov r9,g9
    mov r10,g10
    mov r11,g11
    ```

    Because our original code was 80960-compatible, the movq instruction was aligned and did not access overlapping registers. However, the translator draws our attention to the fact that invalid code would be generated when either of these conditions were present. Since neither are, you can ignore this warning.

17. The program has been manually translated. Close the `xlt.xlt` file.

Running the New Rx-compatible Source Code

1. Copy the xlt.xlt file to another file. At the command prompt, enter:

   `copy xlt.xlt xltconv.s`

2. To compile the Rx-compatible code, enter:

   `gcc960 -AR{`*`P|D`*`} -Fcoff -Tmcyrx -o xlt xltconv.s`

   The options in this command are:

`-AR{`*`P`*`/`*`D`*`}`	sets the target architecture for the compiler. Since you are compiling for the Rx Strategy, use the available i960 Rx architecture options `-ARP` or `-ARD`.
`-Fcoff`	sets the object file type as COFF.
`-Tmcyrx`	uses the linker directive file for the i960 Rx architecture.
`-o xlt`	sets the object file name as `xlt` (optional).
`xltconv.s`	specifies the input source file.

3. To run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci xlt`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`xlt`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out:

   - the value of register r11 before and after the ip-relative branch.
   - the value of the displacement in the index with displacement addressing mode.

- the condition code before and after the `scanbit` instruction.
- the condition code before and after the add instruction.
- the result of performing the `movq` instruction.

5. At the (gdb960) prompt, enter: **`quit`**

CONGRATULATIONS! You have translated source code written for earlier i960 processors. The source code is now Rx-compatible!

Assembler Pseudo-instruction Tutorial

This tutorial demonstrates the use of pseudo-instructions that have been added to the CTOOLS assembler to ease migration between processors. The tutorial that follows demonstrates how to enable and disable the instruction cache for the i960 Cx, Hx, Jx, and Rx microprocessors using microprocessor specific instructions. The tutorial then demonstrates how easy it is to enable and disable the instruction cache using only one pair of pseudo-instructions.

What Are Pseudo-instructions?

A number of pseudo-instructions (pseudo-ops) have been added to the CTOOLS assembler to ease the migration between processors. These pseudo-ops provide an architecture-independent method for performing some of the more common low-level processing operations. Using these pseudo-ops should reduce the number of changes required when moving assembly code from one i960 processor to another.

When you use any of the new i960 pseudo-instructions, you are required to re-assemble your source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best processor instructions to replace the pseudo-instructions based on the processor targeted.

pseudop.c: Editing the File for the Cx Microprocessor

1. If you are using the **Hx Jx Cx & Sx QUICK*val*** software, choose Linker and Utilities. If using the **Rx QUICK*val*** software, this step is not necessary.
2. Choose **Pseudo-op Tutorial**.

3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`. Both procedures contain no code initially. `cache_off()` looks like:

```
cache_off()
{

}
```

5. Add the code necessary to disable the instruction cache for the Cx microprocessor. Between the brackets of the `cache_off()` procedure, add the following line exactly:

```
__asm__ __volatile__("sysctl %0,%1,%2" ::"d"
(((CONFIGURE_ICACHE)<<8)|
(DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
```

   The `cache_off()` procedure should look like this:

```
cache_off()
{
__asm__ __volatile__("sysctl %0,%1,%2" ::"d"
(((CONFIGURE_ICACHE)<<8)|
(DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
}
```

   This procedure, `cache_off()`, uses the instruction cache control processor instruction `sysctl`. This instruction is valid in the i960 Cx processor for managing and controlling the instruction cache. `sysctl` is used above to disable the instruction cache. Also, the `CONFIGURE_ICACHE` and `DISABLE_ICACHE` constants are found in the `system.h` file that is included in the `pseudop.c` file.
6. Likewise, edit the `cache_on()` procedure adding the following line exactly:

```
__asm__ __volatile__("sysctl %0,%1,%2" ::"d"
(((CONFIGURE_ICACHE)<<8)|
(ENABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
```

   The `cache_on()` procedure should look like this:

```
cache_on()
```

```
{
__asm__ __volatile__("sysctl %0,%1,%2" ::"d"
(((CONFIGURE_ICACHE)<<8)|
(ENABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
}
```

Similarly, the `cache_on()` procedure for the Cx microprocessor uses the instruction cache control processor instruction `sysctl`. `sysctl` is used directly above to enable the instruction cache.

7. Save the `pseudop.c` file.

Running pseudop.c For the Cx Microprocessor

1. Compile and run the `pseudop.c` program to show that it works as desired.

NOTE. *If you do not have an i960 Cx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

2. In the Command Prompt window, enter the following commands:

```
gcc960 -AC{F/A} -Fcoff -Tmcycx -o pseudop pseudop.c
```

The options in this command are:

`-AC{F/A}`	sets the target architecture for the compiler. For this example, choose the Cx architecture, `-ACF` or `-ACA`
`-Fcoff`	sets the object file type as coff.
`-Tmcycx`	sets the linker directive file for the Cx architecture.
`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

If you have a Cx microprocessor and want to run the program, enter:

`gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

3. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works with instructions chosen specifically for the Cx architecture. Of course, this is what is expected. This program becomes more interesting when you start using pseudo-instructions.*

4. At the (gdb960) prompt, enter: **`quit`**

pseudop.c: Migrating the File to the Jx/Hx/Rx Microprocessor

Since the i960 Jx, Hx, and Rx microprocessors use the same processor instruction to enable and disable the instruction cache, this migration supports all three processors.

In order to use the program, `pseudop.c`, modified in the first part of this tutorial to support the Jx, Hx, or Rx microprocessor, it must first be migrated to those processors since they do not use the `sysctl` instruction to enable and disable the instruction cache.

1. Choose Qv Code. View the file `pseudop.c` loaded into the editor. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`.

   `cache_off()` contains the Cx specific code and looks like:

   ```
   cache_off()
   {
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   }
   ```

2. Change the code to disable the instruction cache for the i960 Jx/Hx/Rx microprocessors. Between the brackets of the `cache_off()` procedure, delete the previously added line and insert the following line exactly:

   ```
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   ```

   The `cache_off()` procedure should now look like this:

   ```
   cache_off()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   }
   ```

   This procedure, `cache_off()`, uses the instruction cache control processor instruction `icctl`. This instruction is valid in the 80960 Jx/Hx/Rx processors for managing and controlling the instruction cache. `icctl` is used above to disable the instruction cache. Also, the `ICACHE_OFF` constant is found in the `system.h` file that is included in the `pseudop.c` file.

3. Likewise, edit the `cache_on()` procedure deleting the previously added line and inserting the following line exactly:

   ```
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_ON),"d"(0),"d"(0));
   ```

   The `cache_on()` procedure should now look like this:

   ```
   cache_on()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_ON),"d"(0),"d"(0));
   }
   ```

   Similarly, the `cache_on()` procedure for the Jx/Hx/Rx microprocessors use the instruction cache control processor instruction `icctl`. `icctl` is used directly above to enable the instruction cache.
4. Save the `pseudop.c` file.

Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors

1. Compile and run the `pseudop.c` program to show that it works as desired.

NOTE. *If you do not have an i960 Jx, Hx, or Rx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

2. In the Command Prompt window, enter the following commands:

   For the Jx Microprocessor:

   `gcc960 -AJ{F/D/A} -Fcoff -Tmcyjx -o pseudop pseudop.c`

   The options in this command are:

`-AJ{AF/D/T}`	sets the target architecture for the compiler. For this example, to choose the Jx architecture, `-AJA`, `-AJF`, `-AJD`, or `-AJT`
`-Fcoff`	sets the object file type as coff.
`-Tmcyjx`	sets the linker directive file for the Jx architecture.

`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

For the Hx Microprocessor:

`gcc960 -AH{D|A} -Fcoff -Tmcyhx -o pseudop pseudop.c`

The options in this command are:

`-AH{`*`D/A`*`}`	sets the target architecture for the compiler. For this example, to choose the Hx architecture, `-AHD` or `-AHA`
`-Fcoff`	sets the object file type as coff.
`-Tmcyhx`	sets the linker directive file for the Hx architecture.
`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

For Rx Microprocessor:

`gcc960 -AR{`*`P/D`*`} -Fcoff -Tmcyrx -o pseudop pseudop.c`

The options in this command are:

`-AR{`*`P/D`*`}`	sets the target architecture for the compiler. For this example, to choose the Rx architecture: `-ARP` or `-ARD`
`-Fcoff`	sets the object file type as coff.
`-Tmcyrx`	sets the linker directive file for the Rx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

3. If you have a Jx, Hx, or Rx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works with instructions chosen specifically for the architecture in question. Of course, this is what is expected. This program becomes more interesting when* you *start using pseudo-instructions.*

5. At the (gdb960) prompt, enter: **`quit`**

pseudop.c: Adding Pseudo-Ops to the Program

As can be seen, it is neither easy nor fun migrating code from one processor to another, especially when your code is many thousands of lines long. Fortunately, pseudo-instructions have been added to the CTOOLS assembler to ease migration between processors.

1. Choose Qv Code. View the file `pseudop.c` loaded into the editor. You are ready now to rewrite this program using pseudo-instructions. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`.

   `cache_off()` contains the i960 Jx/Hx/Rx microprocessor specific code:

   ```
   cache_off()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   }
   ```

2. Change the code to disable the instruction cache for ALL processors. Between the brackets of the `cache_off()` procedure, delete the previously added line and insert the following line exactly:

   ```
       __asm__ __volatile__("ic_disable r5");
   ```

   The `cache_off()` procedure should now look like this:

   ```
   cache_off()
   {
       /* local register r5 is used to hold the status
   returned */
       __asm__ __volatile__("ic_disable r5");
   }
   ```

   This procedure, `cache_off()`, uses the pseudo-instruction `ic_disable`. When this program, `pseudop.c`, is compiled for a specific 80960 processor by using a `-A` architecture flag, the best instructions for that architecture are chosen to replace the `ic_disable` pseudo-op. Thus, pseudo-ops ease migration between processors. Also, notice only one argument to the pseudo-op is necessary. The `icctl` instruction requires three arguments. Programming with pseudo-ops can be simpler. Pseudo-instructions are also available to perform the other instruction cache management and controlling functions, such as cache invalidation.

3. Likewise, edit the `cache_on()` procedure deleting the previously added line and inserting the following line exactly:

   ```
   __asm__ __volatile__("ic_enable r5");
   ```

   The `cache_on()` procedure should now look like this:

   ```
   cache_on()
   {
         /* local register r5 is used to hold the status
   returned */
       __asm__ __volatile__("ic_enable r5");
   }
   ```

   Similarly, `cache_on()` uses a pseudo-instruction: `ic_enable`. When this program, `pseudop.c`, is compiled for a specific 80960 processor, the best instruction for that architecture is chosen to replace the `ic_enable` pseudo-op.
4. Save the `pseudop.c` file.

Running pseudop.c with Pseudo-instruction

1. Compile and run the `pseudop.c` program to show that the pseudo-instructions work as desired. To prove that the best instruction is chosen for the architecture, compile the code for the Cx microprocessor and then the Jx, Hx, or Rx microprocessor.
2. In the Command Prompt window, enter the following command:

   ```
   gcc960 -AC{F/A} -Fcoff -Tmcycx -o pseudop pseudop.c
   ```

   The options in this command are:

`-AC{F/A}`	sets the target architecture for the compiler. For this example, choose the Cx architecture, `-ACF` or `-ACA`
`-Fcoff`	sets the object file type as coff.
`-Tmcycx`	sets the linker directive file for the Cx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

   When compiled, warnings may be generated. The warnings are generated just to point out this simple fact: when you use any of the new i960 pseudo-instructions, you are required to re-assemble your

source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best instructions to replace the pseudo-instructions based on the processor targeted.

3. If you have a Cx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works as expected with pseudo-instructions.*

The result of this example is similar to using instructions specifically chosen for the Cx architecture. So, using pseudo-instructions can maintain the logic of your code, while easing migration to future i960 microprocessors.

5. At the (gdb960) prompt, enter: `quit`

NOTE. *If you do not have an i960 Jx, Hx, or Rx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors

Now you are ready to compile the code for the Jx, Hx, or Rx microprocessor to demonstrate similar results on a different processor.

1. In the Command Prompt window, enter the following commands:

   For the Jx Microprocessor:

   `gcc960 -AJ{F|D|A} -Fcoff -Tmcyjx -o pseudop pseudop.c`

   The options in this command are:

`-AJ{A|F|D|T}`	sets the target architecture for the compiler. For this example, to choose the Jx architecture, `-AJA`, `-AJF`, `-AJD`, or `-AJT`
`-Fcoff`	sets the object file type as coff.
`-Tmcyjx`	sets the linker directive file for the Jx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

   For the Hx Microprocessor:

   `gcc960 -AH{D|A} -Fcoff -Tmcyhx -o pseudop pseudop.c`

   The options in this command are:

`-AH{D|A}`	sets the target architecture for the compiler. For this example, to choose the Hx architecture, `-AHD` or `-AHA`
`-Fcoff`	sets the object file type as coff.

`-Tmcyhx`	sets the linker directive file for the Hx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

For Rx Microprocessor:

`gcc960 -AR{`*`P|D`*`} -Fcoff -Tmcyrx -o pseudop pseudop.c`

The options in this command are:

`-AR{`*`P|D`*`}`	sets the target architecture for the compiler. For this example, to choose the Rx architecture, `-ARP` or `-ARD`
`-Fcoff`	sets the object file type as coff.
`-Tmcyrx`	sets the linker directive file for the Rx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

2. If you have a Jx, Hx, or Rx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.

`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

3. At the (gdb960) prompt, enter: **`run`**

   The result of this example is the same as using instructions specifically chosen for the Jx, Hx, or Rx architecture. So, using pseudo-instructions does not change the logic of the program. It only eases future migration of your code to future i960 microprocessors.

4. At the (gdb960) prompt, enter: **`quit`**

   When compiled, warnings may be generated. The warnings are generated just to point out this simple fact: When you use any of the new i960 pseudo-instructions, you are required to re-assemble your source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best instructions to replace the pseudo-instructions based on the processor targeted.

CONGRATULATIONS! You can now start using pseudo-instructions in your code to ease migration of your code to future i960 processors.

Debugging with gdb960

A software debugger is a useful tool that allows you to learn more about the behavior of an application program while it is running on a target or simulator. gdb960 is a source-level debugger that allows you to interact with your application program running on a target system through the debug monitor, MON960. MON960 is resident on the Cyclone CPU module.

This example uses the card game, Go Fish, and is designed to teach you a few debugger commands so that you can further examine the example programs provided with this kit or your own programs. In the card game, Go Fish, you and the computer each get several cards. You take turns guessing which cards are in each other’s hands. When you guess correctly, you acquire that card. If you don’t guess correctly, you need to “Go Fish” and draw another card from the pack. When you get four-of-a-kind, you

remove those cards from your hand. The objective of the game is to have the most sets of four-of-a-kind when either you or the computer has no cards remaining in your hands.

NOTE. *This example uses the command line interface to gdb960. The program also features a Graphical User Interface in both Windows and UNIX. See The gdb960 User's Manual for more information.*

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Debugger**.
2. Choose **gdb960 Tutorial**.
3. Choose **Make** to compile, link, and download the program automatically.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

NOTE. *DEBUGGING SHORTCUTS*
Abbreviations for gdb960 commands are accepted as long as they are unambiguous.
*To **run,** enter: **r***
*To **break,** enter: **br***
*To **list,** enter: **l***
*To **continue,** enter: **c***
*To **print,** enter: **p***
*To **clear,** enter: **cl***
*To **quit,** enter: **qu***
*For **help,** enter: **he***

4. **DO NOT TYPE RUN!** First, use the gdb960 debugger to set a breakpoint at function `main()`. Type:

   `break main`

   The debugger responds by displaying:

```
Breakpoint 1 set at 0xa0008570: file fish.c, line 209.
```

5. Set a second breakpoint at line 275. Type:

   `break 275`

   The debugger responds by displaying:

```
Breakpoint 2 set at 0xa0008bc4: file fish.c, line 275.
```

6. To execute the program from the beginning, type:

   `run`

   The debugger responds by displaying:

```
Starting program: C:\QUICKVAL/fish
Breakpoint 1, main() at fish.c, 209.
209    srand();
```

7. To display the code at the breakpoint, type:

   `list`

   The debugger displays lines 204-213 of the fish.c source. To see the next ten lines, type **list** again.

8. To continue executing the program from this location, type:

   `continue`

   The debugger responds by displaying:

```
Continue.
Would you like instructions[n]?
```

9. Reply by typing **y** for yes or <Enter> or **n** for no.

```
your hand is: A A 6 6 8 8 9
Breakpoint 2, game() at fish.c:275.
275    if(!move(yourhand,myhand,g=guess(),0))break;
```

10. In the source code in step 9, there are two variable arrays, myhand and yourhand. Myhand is the computer's hand and yourhand is yours. To look at the card in the computer's hand, type:

    `print myhand`

    The debugger responds by displaying:

```
$1="000\000\000\001\000\002\000\001\000\000\001\002\000"
```

    `myhand[0]` does not represent a card.

    `myhand[1]` represents the number of Aces.

`myhand[2]` represents the number of 2s, and so on.

The same order of cards is represented in the array, `yourhand`.

If a King is drawn by either player, `myhand[13]` or `yourhand[13]` will appear when you print the array.

11. Using the ability to see the computer's hand, you are able to beat the computer every time. Clear the first breakpoint at the function `main()` and continue playing the game, looking at the computer's hand any time you need to. To clear the breakpoint at `main()`, type:

    `clear main`

    The debugger responds by displaying:

    ```
    Deleted breakpoint 1
    ```

12. To continue executing the program, type:

    `continue`

13. If you need further assistance beating the computer, contact the 80960 Technical Support Group for more hints.
14. Type: **`quit`**

Debugging Optimized Code

CTOOLS can use the ELF object module format and DWARF Version 2 debug information format as described in the *80960 Embedded Application Binary Interface (ABI) Specification* (order number 631999). The new formats enable more accurate mapping between source and object code at higher optimization levels and ease production code debugging.

This example shows that at the highest level of module-local optimization, it is possible to set a breakpoint on an inline function using ELF/DWARF, while with COFF this is not possible.

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Debugger**.
2. Choose **C ELF/DWARF Format**.
3. Choose **Make**.

    The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

4. Compile *swap.c* with no module-local optimizations (no inlining). This shows that the procedure *swap* is not inlined. Enter:

   gcc960 -Felf -T*{Link-dir}* **-A***{arch}* **-O0 -S swap.c**

   The options in this command are:

`-Felf`	creates an ELF format output file
`-A`*{arch}*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*{Link-dir}*	specifies the linker directive file. For example, `-Tmcyrx` specifies `mcyrx.gld`.
`-O0`	no module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

5. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, see the call to the procedure swap:

   ```
   callj _swap
   ```

   This is an out-of-line call to the procedure swap. The function swap has not been inlined.

6. Now, compile `swap.c` with the highest level of module-local optimizations. This inlines the procedure `swap`.

   gcc960 -Felf -T*{Link-dir}* **-A***{arch}* **-O4 -S swap.c**

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*{arch}*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*{Link-dir}*	specifies the linker directive file. For example, `-Tmcyrx` specifies `mcyrx.gld`.
`-O4`	highest level of module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

7. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, note the call to the procedure `swap` does not exist:

   ```
   callj _swap  /* Does Not Exist*/
   ```

   The procedure swap has been inlined.

8. Recompile using the `-O4` optimization level, the ELF/DWARF format, and add debugging information.

```
gcc960 -Felf -T{Link-dir} -A{arch} -O4 -g -o swap swap.c
```

The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyrx` specifies mcyrx.gld.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o swap`	names the executable file swap
`swap.c`	input file

9. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

```
gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap
```

The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`swap`	the executable file

10. **DO NOT TYPE RUN!**

First, set a breakpoint on the procedure *swap.* Enter:

```
break swap
```

The debugger responds by displaying:

```
breakpoint 1 @0xa00080f0:file swap.c, line 43
breakpoint 2 @0xa0008148:file swap.c, line 54
```

Breakpoint 1 is the out-of-line reference to the procedure `swap`. Breakpoint 2 is the inline reference to the procedure `swap`.

`Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is still possible to set a breakpoint on the inline function. Breakpoint 2 stops program execution.

11. To execute the program, enter:

    `run`

    The debugger responds by displaying:

    ```
    Breakpoint 2, main() @ swap.c: 54
    54 printf(ìThe smallest number is %d\nî,a);
    ```

12. To continue the program, enter:

    `c`

    When the program has finished, enter:

    `quit`

13. Compile using the `-O4` optimization level, the COFF format, and add debugging information.

`gcc960 -Fcoff -T`*`{Link-dir}`*` ``-A`***`{arch}`*** **`-g -O4 -o swap swap.c`**

The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyrx` specifies `mcyrx.gld`.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o swap`	names the executable file `swap`
`swap.c`	input file

14. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

    `gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap`

    The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1

`-D lpt1`	use parallel port 1
`swap`	the executable file

15. **DO NOT TYPE RUN!!**

    First, set a breakpoint on the procedure `swap`. Enter:

    `break swap`

    The debugger responds by displaying:

    ```
    breakpoint 1 @0xa00080f0
    ```

    Breakpoint 1 is the out-of-line reference to the procedure `swap`. Notice that no inline breakpoint has been set. This breakpoint does not stop execution of the program.

    `Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is not possible to set a breakpoint on the inline function. Program execution does not stop.

16. To execute the program, enter:

    `run`

    The debugger responds by displaying the smallest number from the swap. There is no break in program execution.

17. When the program has finished, enter:

    `quit`

    You have now seen that with the ELF/DWARF format, it is now possible to debug your production code, even after high levels of program optimization.

Debugging Optimized C++ Code Tutorial

The C++ compiler generates debug information using the DWARF format when the `-g` option is specified with the `-Felf` option. This debug information format is richer than that of other supported OMFs, and allows more reliable debugging under optimization.

This tutorial demonstrates that at the highest level of module-local optimization, debugging a C++ application is still possible due to the DWARF debug format.

In this example, you compile a C++ program using the -O0 optimization compiler option, which disables all optimizations, including those that may interfere with debugging. The same C++ program is then compiled using the highest-level of module-local optimization, -O4.

There are several levels of program optimization available with the CTOOLS development tool suite. Typically, low levels of optimization are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once the application is functioning properly, the application's performance may be increased by using a higher level of optimization. The static optimization options are:

`O0`	Turn optimization off
`O1`	Basic optimization
`O2`	strength-reduction, instruction scheduling for pipelining, etc...
`O3`	`O2` plus `fconstprop, finline-functions`, etc...
`O4`	`O3` plus `fsplit-mem, fmarry-mem, fcoalesce`

Level O4 is the highest level of static optimization. Please refer to the *i960 Processor Compiler User's Guide* for more information on ELF/DWARF and compiler optimizations.

In this tutorial, you compile and debug a C++ program, `cppdwarf.cpp`, that contains many of the advanced features of the C++ language, including:

- Classes
- Public, protected, and private variable accessibility
- Virtual functions
- Scope operators
- Overloaded functions
- Class inheritance

Using ELF/DWARF, both levels of optimization, `-O0` and `-O4`, retain the C++ program structure so that the above features may be investigated.

This example is found in the **Hx Jx Cx & Sx QUICK*val*** software. To use this tutorial, choose Jx as the architecture and PCI80960DP for the evaluation board.

1. Choose **Debugger**
2. Choose **C++ ELF/DWARF Format**

3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Compile the program using the `-O0` optimization level. In the Command Prompt window, enter the following command:

   `gcc960 -Felf -A{`*`arch`*`} -T{`*`Link-dir`*`} -stdlibcpp -O0 -g -o cppdwarf cppdwarf.cpp`

   The options in this command are:

`-Felf`	creates an ELF format output file.
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD.
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcyrx` specifies mcyrx.gld.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-O0`	specifies the lowest level of module-local optimizations.
`-g`	includes debug information in object file.
`-o cppdwarf`	specifies the executable file `cppdwarf`.
`cppdwarf.cpp`	specifies the input file `cppdwarf.cpp`.

5. Run the program using the debugger, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -D {`*`parallel port`*`} -pci cppdwarf`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.

`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, ... com99.
`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are lpt1 and lpt2.
`-pci`	sets the code download option for the PCI bus (optional). When no serial port is specified, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`cppdwarf`	specifies the executable file `cppdwarf`.

6. **Do Not Enter Run!**

   Now you are ready to examine some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes.

   At the (gdb960) prompt, enter:

   ```
   ptype person
   ```

The following data type information concerning the class `person` appears:

Example 4-1 person Class

```
  type = class person {
    protected:
      char name[40];
      char dept[40];
    public:
      void setName ();
      void setName (char *);
      void setDept ();
      void setDept (char *);
      void printName ();
      virtual int isOutstanding ();
      virtual char * getDept ();
}
```

Please note the following concerning the above output:

- The entire class information for person is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.

Another C++ class in the program is `professor`, which inherits from the person class. Again, you use the gdb960 command `ptype` to display a description of the `professor` class.

7. At the (gdb960) prompt, enter:

   `ptype professor`

The following data type information concerning the class professor appears:

Example 4-2 professor Class

```
type = class professor : public person {
  private:
    int numPubs;
  public:
    void setNumPubs ();
    void setNumPubs (int);
    virtual int isOutstanding ();
}
```

Please note the following concerning the above output:

- The entire class information for `professor` is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.
- `type = class professor : public person` indicates that the `professor` class inherits from the `person` class.

8. You are ready to set some breakpoints.
   a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

      `break professor::setNumPubs`

The following information concerning breakpoints is displayed:

```
[0] cancel
[1] all
[2] professor::setNumPubs(int) at
cppdwarf.cpp:125
[3] professor::setNumPubs(void) at
cppdwarf.cpp:118
```

Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.

b. Set a breakpoint on all `professor::setNumPubs` functions. At the `>` prompt, enter: **`1`**

   The following information about breakpoints is displayed:

```
Breakpoint 1 at 0xa00083d0: file cppdwarf.cpp,
line 125.
Breakpoint 2 at 0xa0008358: file cppdwarf.cpp,
line 118.
```

c. Set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

   `break professor::isOutstanding`

   The following information concerning breakpoints is displayed:

```
Breakpoint 3 at 0xa0009080: file cppdwarf.cpp,
line 110.
```

9. You are now ready to start the program. At the (gdb960) prompt, enter:

   `run`

   Notice that the program stops at all three of the breakpoints.

10. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.

11. At the (gdb960) prompt, enter: **`quit`**

The results of the debug session were as expected because no optimizations had been performed on the source code during compilation. You can now recompile the `cppdwarf.cpp` program using the highest-level of module-local optimization and repeat the previous debug session.

12. Compile the program using the `-O4` optimization level. In the Command Prompt window, enter the following command:

    `gcc960 -Felf -A{`*`arch`*`}-T{`*`Link-dir`*`} -stdlibcpp -O4 -g -o cppdwarf cppdwarf.cpp`

    The options in this command are:

`-Felf`	create an ELF format output file
`-A{`*`arch`*`}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{`*`Link-dir`*`}`	specifies the linker directive file. For example, `-Tmcyrx` specifies `mcyrx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o cppdwarf`	specifies the executable file `cppdwarf`
`cppdwarf.cpp`	input file

13. Run the program using the debugger, enter:

    `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -D {`*`parallel port`*`} -pci cppdwarf`

    The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.

`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, ... com99.
`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are: lpt1 and lpt2.
`-pci`	sets the code download option for the PCI bus (optional). When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used.)
`cppdwarf`	specifies the executable file.

14. **Do Not Enter Run!**

    You are now ready to investigate some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes. At the (gdb960) prompt, enter:

    `ptype person`

    Please note, the output matches that of Example 4-1, “person Class”. Optimizations did not affect the `person` class output. It is the same as the first debug session.

15. Another C++ class in the program is `professor`, which inherits from the person class. Once again, you use the gdb960 command `ptype` to display a description of the `professor` class. At the (gdb960) prompt, enter:

    `ptype professor`

    Again please note, the output matches that of Example 4-2, “professor Class”. Optimizations did not affect the `professor` class output. It is the same as the first debug session.

16. You are now ready to set some breakpoints.
    a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

       `break professor::setNumPubs`

       The following information concerning breakpoints is displayed:

       ```
       [0] cancel
       [1] all
       [2] professor::setNumPubs(int) at
       cppdwarf.cpp:125
       [3] professor::setNumPubs(void) at
       cppdwarf.cpp:118
       ```

       Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 only sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.
    b. Set a breakpoint on all `professor::setNumPubs` functions, so At the `>` prompt, enter: **`1`**.

       The following information about breakpoints is displayed:

       ```
       Breakpoint 1 at 0xa00082e4: file cppdwarf.cpp,
       line 125.
       Breakpoint 2 at 0xa0008294: file cppdwarf.cpp,
       line 118.
       ```
    c. Finally, set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

       `break professor::isOutstanding`

       The following information concerning breakpoints is displayed:

       ```
       Breakpoint 3 at 0xa0008960: file cppdwarf.cpp,
       line 111.
       ```

17. You are now ready to start the program. At the (gdb960) prompt, enter:

    `run`

    Notice that the program does not stop at all three of the breakpoints. As can be seen, the DWARF debug information format is very rich, 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.
18. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.
19. At the (gdb960) prompt, enter: **`quit`**

CONGRATULATIONS! You may now know how to use ELF/DWARF to debug your optimized C++ code.

Writing Flash on the IQ80960RP Evaluation Board

This example teaches you the following:

- Writing to Flash on the Cyclone base board.
- Booting off of the Flash in socket U3 of the Cyclone base board, as opposed to the Flash on the CPU Module.
- Setting the Cyclone base board to 12 volts.
- Using mondb.exe as a simple utility to download and execute an application program on the target board running MON960.
- Using mondb.exe to write Flash.
- Building a new monitor.

Complete these steps to write the Flash:

1. Identify the Flash on the Cyclone base board

   A blank Flash ships on each Cyclone board in socket U3. If your board did not ship with a Flash in socket U3, insert an Intel N28F020 Flash chip.
2. Set the Cyclone eval base board voltage to 12 volts

   Locate the four-position DIP switch labeled SW1. Flip S1.1 to the *ON* position. This enables VPP to the Cyclone base board Flash.
3. Power up or reset the host PC to reset Cyclone board.

   On the IQ80960RP Platform, +12 VDC and +5 VDC power is supplied through the edge connector.

4. Edit `Version.c.`
   a. Change directories to where the version.c file resides. The default installation directory for CTOOLS is:

      `c:\intel960\src\mon960\common`

      Version.c contains the following information:

      ```
      const char mon_version_byte =  nn;   /* version
      n.n = nn */
      const char base_version[] = "MON960 n.n.n";
      const char build_date[] = __DATE__;
      ```
   b. Change the file contents to reflect that this is your version of MON960. For example, change

      ```
      const char base_version[] = "MON960 n.n.n";
      ```

      to:

      ```
      const char base_version[] = "MY MON960";
      ```
   c. Save Version.c.
5. Build the new MON960

   By default the source for MON960 is located at:
   `c:\intel960\src\mon960\common`

   You may use the pre-built version of MON960 there, or build a custom version. To create a custom version:

   a. Copy `makefile.xxx` to `c:\intel960\src\mon960\common\makefile`.

      where xxx is one of the following make files

   - `makefile.ic` (ic960 interface, COFF format)
   - `makefile.ice` (ic960 interface, ELF format)
   - `makefile.gnu` (gcc960 interface, COFF format)
   - `makefile.gne` (gcc960 interface, ELF format)

   b. Issue the commands:

   ```
   nmake -s makefile
   cyrp
   ```

   This creates a file called `cyrp.fls`.

6. Writing the Flash

   To write the Flash, use the mondb.exe utility located in the `intel960\bin\` directory. If you are going to use the pre-built MON960 files, they are located in the `intel960\roms` directory. For example, if you used the default installation directory and are using the pre-built MON960 files for the 80960RP, enter:

   ```
   mondb -pci -ef -ne c:\intel960\roms\cyrp.fls
   ```

   The options in this command are:

`-pci`	Use PCI bus for communication
`-ne`	no execute
`-ef`	erase Flash
`cyrp.fls`	input Flash filename

   Note also that if you built a version of MON960 from the source code as described previously, the `cyrp.fls` file will be located in the `c:\intel960\src\mon960\common\` directory.

7. Set board voltage back to +5 VDC

   Locate the four-position DIP switch labeled SW1. Set S1.1 to the *OFF* position. This disables VPP to Cyclone RP base board Flash and protects the Flash.

8. Set the board to boot from U3 socket

   Locate the four-position DIP switch labeled SW1. Set SW1.2 ROMSWAP to the *ON* position. This exchanges the addresses of the U4 and U3 ROMs. When the switch is *OFF* the processor boots from the U4 ROM. When the switch is *ON* the processor boots from the U3 ROM.

9. Reboot the base board by rebooting the host PC. There is no reset switch on the IQ80960RP evaluation board.

Writing Flash on the IQ80960RD Evaluation Board

This example teaches you the following:

- Writing to Flash on the IQ80960RD66 Evaluation Platform.
- Booting off the Flash in socket U9 of the Cyclone base board, as opposed to the Flash in socket U10.
- Using mondb.exe to write Flash.
- Building a new monitor.

Complete these steps to write the Flash:

1. Identify the Flash on the Cyclone base board

   A blank Intel 28F008SA Flash chip ships on each RD66 Cyclone board in socket U9. On the IQ80960RD66 Platform, +12 VDC power is supplied through the edge connector for writing the blank Flash chip. Note: The IQ80960RD66 Cyclone base board is always set at +12 VDC power.

2. Edit `Version.c`.

   a. Change directories to where the version.c file resides. The default installation directory for CTOOLS is:

      `c:\intel960\src\mon960\common`

      `Version.c` contains the following information:

      ```
      const char mon_version_byte = nn; /* version n.n
      = nn */
      const char base_version[] = "MON960 n.n.n";
      const char build_date[] = __DATE__;
      ```

   b. Change the file contents to reflect that this is your version of MON960. For example, change

      ```
      const char base_version[] = "MON960 n.n.n";
      ```

      to:

      ```
      const char base_version[] = "MY MON960";
      ```

   c. Save `Version.c`.

3. Build the new MON960

   Note: this example requires Microsoft's `nmake` make utility.

   By default the source for MON960 is located at:
   `c:\intel960\src\mon960\common`

   You may use the pre-built version of MON960 in `c:\intel960\roms`, or build a custom version. To create a custom version:

   a. Copy

      `c:\intel960\src\mon960\common\makefile.xxx`

      to

      `c:\intel960\src\mon960\common\makefile.`

      where `xxx` is one of the following make files

   - `makefile.ic` (ic960 interface, COFF format)

- `makefile.ie` (ic960 interface, ELF format)
- `makefile.gc` (gcc960 interface, COFF format)
- `makefile.ge` (gcc960 interface, ELF format)

b. Issue the commands:

```
nmake -s makefile
cyrd
```

This creates a file called `cyrd.fls`.

4. Writing the Flash.

   To write the Flash, use the mondb.exe utility located in the intel960\bin\ directory. If you are going to use the pre-built MON960 files, they are located in the `intel960\roms` directory. For example, if you used the default installation directory and are using the pre-built MON960 files for the 80960RD, enter:

   ```
   mondb -pci -ef -ne c:\intel960\roms\cyrd.fls
   ```

   The options in this command are:

`-pci`	use PCI bus for communication.
`-ne`	no execute.
`-ef`	erase Flash.
`cyrd.fls`	input Flash filename

   Note also that if you built a version of MON960 from the source code as described previously, the `cyrd.fls` file will be located in the `c:\intel960\src\mon960\common\` directory.

5. Set the board to boot from U9 socket.

   Locate the four-position DIP switch labeled SW1. Set SW1.3 ROMSWAP to the **OFF** position. This exchanges the addresses of the U9 and U10 ROMs. When the switch is **ON** the processor boots from the U10 ROM. When the switch is **OFF** the processor boots from the U9 ROM.

6. Reboot the base board by rebooting the host PC. There is no reset switch on the IQ80960RD66 evaluation board.

80960Rx Initialization Example

Complete the following steps:

1. Choose **RP Init Code**.
2. When the editor appears, open the Init Code menu and choose the initialization code that you wish to view. The options are **Main**, **Memory Controller**, **System Init**, and **Hardware Init**. The sections that follow describe these modules.

Module: INIT.S

This module contains the Initial Memory Image, including a PRCB, System Procedure Table, Fault Table, Interrupt Table. These data structures are in ROM during the initial boot. This module also contains the cold start address (start_ip) and the system initialization code. The initialization code does the following:

- code execution begins at start_ip label after boot
- calls pre_init, if required, to perform board self-test and enable RAM
  - — pre_init is set to init_mem in MONCYRP.LD file
  - — the init_mem function is located in the CYRP_ASM.S file
- copies the processor data structures to RAM
  - — initialize interrupt table and fill table with vectors
  - — initialize fault table with all entries to the fault entry point (Set_prcb will initialize the trace entry properly)
  - — initialize system procedure table
- initializes the monitor's data in RAM
  - — call _set_prcb to copy PRCB to RAM
- reinitialize processor with RAM based PRCB
- turn off interrupted state and changes to the monitor's stack
- branch to mon960_main in the MAIN.C file.

Module: MAIN.C

This module contains the mon960_main and init_regs functions.

mon960_main():

- call init_regs function in the MAIN.C file
- call init_hardware function in the CYRP_HW.C file
  - — performs initialization of the board and the Rx specific registers
- call init_monitor function in the MONITOR.C file
- call the monitor function in the MONITOR.C file
- Note: control does not return here after the call to monitor()

init_regs():

Initializes the "soft" copy of the register set used by the monitor

MODULE: CYRP_HW.C

This module contains the init_hardware function. This is the main initialization routine for the i960 Rx evaluation board.

init_hardware():

- call disable_dcache function in RP.S file
  - — disables data cache
- call init_eeprom function in FLASH.C file
  - — establish operational parameters for the various banks of Flash ROM
- call init_atu function in CYRP_HW.C file
  - — initialize the ATU and MU registers on the Rx.
- call init_bridge function in CYRP_HW.C file
  - — initialize the bridge registers on the Rx.
- call int_setup function in PCI_SERV.C file
- call init_pci_config_regs function in PCIDRVR.C
- call clear_retry function in CYRP_HW.C file
  - — clear the PCI config retry bit to allow acceptance of configuration cycles

MODULE: MONITOR.C

This module contains the init_monitor and monitor functions.

init_monitor():

- set up CPU version once at boot time

monitor():

- if first time, determine which com port to connect
- call hi_main or ui_main for HDIL or TERMINAL INTERFACE respectively
- note: control WILL NOT return here after the call to hi_main() or ui_main

Other i960 Processor Choices and the Remote Evaluation Facility

The i960 RISC processor family has a wide breadth of processors to match your design's price and performance needs. If you wish to evaluate other i960 processor family members, contact your local distributor and order different Cyclone CPU modules, or visit the Remote Evaluation Facility at `http://developer.intel.com/design/i960/testcntr`

NOTE. *The i960 Rx Processor is not available through the Remote Evaluation Facility.*

If you choose to order more CPU modules, you may rest assured that all i960 processor modules plug-n-play with your QUICK*val* kit. This configuration was specifically designed to protect your investment and offer a low cost migration path for future needs.

The i960 Hx CPU Example Programs

5

The i960 Hx microprocessor is the performance follow-on product to the i960 Cx microprocessor.

- The 80960Hx is pin and binary code-compatible with the 80960Cx Core Architecture. [1]
- It includes a 32-bit demultiplexed and pipelined burst bus interface.
- Integrated interrupt controller.
- The instruction cache is 16 Kbytes, the data cache is 8 Kbytes, and the data RAM is expanded to 2 Kbytes.
- The 80960HD is clock doubled, and the 80960HT is clock tripled.

The i960 Hx microprocessor is used in a wide variety of application areas:

Office Automation	Page-printer controllers, image scanners, X terminals, Local Area Network (LAN) controllers and communications bridges (ATM, FDDI), database engines, telecommunications and data communications equipment, and I/O processing for workstations/servers.
Industrial Robotics	Automated vision systems and factory process control.

1. Though not drop-in replaceable. Customers can design systems that accept either i960 Hx or Cx processors.

Medical Instrumentation	Real-time data collection and analyses, monitoring systems, and ultrasound imaging displays.
Avionics and Aerospace	Flight-control equipment, ground-to-air communication systems, and satellite navigation computers.

Additionally, you can optimize your system's performance with CTOOLS, which includes a profile-driven compiler that can automatically optimize your code based on its runtime behavior.

The following pages describe the example programs included with this kit. Each example highlights a feature of the architecture or CTOOLS and provides you with source code that can help shorten your software development cycle. Table 5-1 provides descriptions of the tutorials included in the i960 Hx QUICK*val* kit.

Table 5-1 QUICK*val* i960 Processor Sample Programs

Tutorial Description	Source Files
Hello World: Uses simple printf statement to verify system integrity.	`hello.c`: source file `system.c`: system file
Memory Test: Used for system verification of external memory. The programs perform byte, short, or word writes to external memory, and then they check the addresses written for correctness.	`memtst8.c:` 8 bit memory test `memtst16.c`: 16 bit memory test `memtst32.c`: 32 bit memory test `system.c`: system file
Data Cache: Uses the minimum edit distance algorithm to demonstrate the effectiveness of the on-chip data cache. This example also shows how to enable and disable the data cache and how to configure an area of memory for caching.	`dcache.c`: source file `system.c`: system file
Instruction Cache: Uses a simple loop to demonstrate how to enable and disable the instruction cache. It also highlights the performance advantage obtained when using the on-chip instruction cache.	`loop.c`: source file `system.c`: system file

continued ☞

Table 5-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
Register Cache: Demonstrates using the on-chip register cache in reducing the interrupt latency for high priority interrupts.	`reg_int.c`: source file `low_int.s`: interrupt handler for low priority `high_int.s`: interrupt handler for high priority `system.c`: system file
External Interrupts: Shows how to configure the Cyclone board timers to trigger hardware interrupts. This is also an example of using interrupt handlers and placing the handlers in the interrupt table.	`cyint.c`: source file `asm_fns.s`: interrupt handler for Sx `int_proc.s`: interrupt handler-all processors but Sx `t85c36.c`: eval board timer file `system.c`: system file
Internal Interrupts: Simple timer example showing how to overlay the memory-mapped registers with a structure to program the on-chip timers. This tutorial also shows how to set up interrupt routines using the timers.	`timrcntr.c`: source file `timers.c`: on-chip timer file `system.c`: system file
Fault Handling: Shows how to set up the fault handling procedures in the fault and system procedure tables.	`fault.c`: source file `flt_proc.c`: fault procedures `asm_flt.s`: assembly functions to help generate faults `system.c`: system file
C Local Optimizations: Shows how to use the C compiler with high levels of static optimization for improved runtime performance.	`chksum.c`, `system.c`: source files
C Global Optimizations: Shows how to use program-wide optimizations of the C compiler for increased performance.	`chksum.c`, `system.c`: source files
C++ Local Optimizations: Shows how to use the C++ compiler with high levels of static optimization for improved runtime performance.	`optimize.cpp`: source file

continued ☛

Table 5-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
C++ Global Optimizations: Shows how to use program-wide optimizations of the C++ compiler for increased performance.	`optimize.cpp`: source file
C++ Virtual Function Optimizations: Shows how a call to a virtual function can be replaced by a direct call to a member function, and, if possible, it may be inlined at the call site. This improves the runtime performance of the code.	`optimize.cpp`: source file
Profiling Lab: Teaches you how to use some of CTOOLS advanced profiling features.	`chksum.c`: source file
Self-Contained Profile: Shows how to create a self-contained profile that captures the program structure and associates it with the program counters from a raw profile. When the source program changes, the global decision making step interpolates or stretches the counters in the self-contained profile to fit the changed program.	`quick.c`: source file
Incremental Profiling: Shows how to profile a program in pieces and then re-combine them later, a useful methodology when the target execution environment is memory limited	`fault.c`, `flt_proc.c`, `asm_flt.s`, `system.c`: source files
C Cave: Uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.	`ttt.c`: source file
C++ Cave: Shows how to reduce target memory requirements. The text sections of compressed and uncompressed C++ executables are compared. This example also shows how to specify functions for compression.	`cavecpp.cpp`: source file

continued ☛

Table 5-1 QUICK*val* i960 Processor Sample Programs (continued)

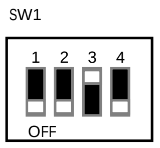
Tutorial Description	Source Files
Linker Directive Language: Provides a hyperlinked manual that describes the linker command options. This tutorial is found in the online help only, not in this manual.	
Linker Consumption: Shows the ability of the linker, gld960, to consume b.out-format, COFF, or ELF object files and libraries in any combination.	`cyint.c`, `int_proc.s`, `t85c36.c`, `system.c`: source files
i960 Processor Assembler Pseudo-Instruction Support: Shows how to use the new assembler pseudo-ops.	`pseudop.c`: source file
Debugging with gdb960: Uses the Go Fish card game to teach a few useful debugger commands.	`fish.c`: source file `system.c`: system file
ELF/DWARF Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to set a breakpoint on an in-line function.	`swap.c`: source file
C++ DWARF-2 Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to debug a C++ application.	`cppdwarf.cpp`: source file
Retargeting MON960: Provides steps for retargeting MON960. This tutorial is found in the online help only, not in this manual.	
Writing Flash: Demonstrates how to update the version of MON960 on your evaluation board.	
80960 Family Benchmark: Shows how to use this facility to compare your processor's performance with other i960 family members. This example uses a typical checksum routine to show how to add benchmarking routines into source code.	`chksum.c`, `system.c`: source files
Remote Evaluation Facility: Guides you through the use of this new benchmarking facility on the World-Wide Web.	

System Validation

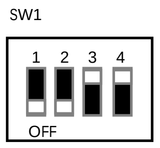
Hello World

The program `hello.c` is used to verify your software and hardware system integrity. The following steps provide instructions on how to compile, link, download, and execute this program.

1. Verify that your software and hardware have been installed according to the instructions in Chapter 2 through 3 and the frequency switch on your CPU module is set as shown. The switch settings below set the Hx CPU module frequency at 33 or 25 MHz respectively; since the HD processor is clock doubled, the processor runs internally at 66 or 50 MHz.



66 MHz module



50 MHz module

1. Power your Cyclone evaluation platform and i960 Hx CPU module
2. Double-click on the **Hx Jx Cx & Sx QUICK*val*** icon in the QUICK*val* program group.
3. Configure you hardware.
   - Select the **80960 Architecture** tab.
   - Select **Hx**.
   - Depending on the board you have installed, select either the EP80960BB or PCI80960DP tab.
   - Configure the software communication options to match those of your evaluation board.
   - Choose **OK**.
4. Choose **Hello World**.
5. Choose **Make** to compile, link, and download the program automatically.

6. Use the gdb960 debugger to execute hello. Type:
   `run`
7. The gdb960 debugger responds by displaying:

```
Hello...Welcome to the 80960HX QUICKval Kit!
SYSTEM CHECK COMPLETED!!
Now you may proceed with our Example Programs.
Program Exit: 01
(gdb960)
```

8. To exit the debugger, type: **`quit`**

CONGRATULATIONS! You have successfully installed your software and your hardware, compiled a program using gcc960, and downloaded and executed the program on your evaluation board using the gdb960 debugger.

If you received any error messages during this process, refer to "If Something Goes Wrong" on page 5-8.

Memory Test

The programs `memtst8.c`, `memtst16.c`, and `memtst32.c` are used to test the external memory on the Cyclone base board.

Depending on the test that is run, an 8, 16, or 32-bit test is run on an area of memory. The program writes F's and 0's to a memory location and reads the location to verify the integrity of what was written. All three programs are almost identical, with the exception of the casting of the variable *ADDR, which allows you to perform different test types.

NOTE. *Below,* `memtst*.c` *refers to either the byte, short, or word memory test example.*

1. Choose **Memory Test**.
2. Choose a memory test. The options are, **8-bit Memory Test**, **16-bit Memory Test**, or **32-bit Memory Test**.
3. Choose **Make** to compile, link, and download the program automatically.

4. Use the gdb960 debugger to execute memtst. Type:

   `run`

5. For the 8-bit test, `memtst8.c`, the gdb960 debugger responds by displaying:

```
This program will run a 8-bit test on the external memory.

Test to be implemented is byte test.
Starting address = a000dfb0
Ending address = a000ec30

Press enter to begin test with 0’s.
Number of errors that occurred is 0.

Begin test for f’s.

Press enter to continue.
Number of errors that occurred is 0.

All tests are complete.
Program exited with code 030.
(gdb960)
```

6. Exit the debugger. Type:

   `quit`

If Something Goes Wrong

The following section describes a few actions that may help resolve errors that may have occurred when invoking one of the tools. If you were unable to get the proper response from the gdb960 debugger after executing the above programs and the trouble-shooting hints described below do not help, contact the 80960 Technical Support Group by phone at 1-800-628-8686 or by E-mail at 960tools@intel.com.

MON960 Debug Monitor is Not Responding...

If the red FAIL LED (CR6) on the base board is lit, the monitor may not have booted up correctly. Press the reset button (S2). If the red FAIL LED remains lit, contact the 80960 Technical Support Group.

Invoking the gcc960 Compiler Resulted in Errors...

The environment must be set-up as described in Chapter 2. If you chose the default directories while installing CTOOLS, verify that the path names `C:\INTEL960\BIN` have been added to your PATH variable and that the following statement is in your `autoexec.bat` file. If you did not install these tools using the default directories, make the appropriate change.

```
SET G960BASE=C:\INTEL960
```

NOTE. *You did not use the default directories on installation, please make sure the G960BASE environment variable is assigned appropriately.*

NOTE. *Don't forget to re-boot your system once you have made any necessary changes to your* `autoexec.bat` *file.*

Invoking the gld960 Linker Resulted in Errors...

Verify that the directory that contains the `hello.c` and `memtst*.c` example programs also now has the object files, `hello.o` and `memtst*.o`. If `hello.o` and `memtst*.o` do not exist, then the gcc960 compiler command did not successfully create an object file. Re-compile `hello.c` and `memtst*.c` to see if an error occurred during the compilation.

If `hello.o` and `memtst*.o` do not exist, make note of the error message and contact the 80960 Technical Support Group.

Invoking the gdb960 Debugger Resulted in Errors...

NOTE. *If you are using the PCI-SDK evaluation platform, you may specify* `-pci` *for PCI download and PCI communication.*
For a list of all the gdb960 command line options, at a command prompt, enter: **`gdb960 -h | more`**

Serial communication error

A serial communication error causes the gdb960 debugger to respond by displaying:

```
HDIL error (10), communication failure
HDIL error (10), communication failure
You can’t do that when your target is ‘exec’
```

Verify that the serial port you are using is the one you specified in the gdb960 command line. Verify that your serial cable is properly connected to the board and to your PC.

Parallel communication error

A parallel communication error causes the gdb960 debugger to respond by displaying:

```
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type ‘show copying’ to see the conditions.
There is absolutely no warranty for GDB; type “show warranty” for
details.
gdb960.exe 6.0, Wed FEB 16 12:33:16 1998
GDB 5.10 (i486-intel-dos --target i960-intel-mon960), Copyright 1997
Free Software Foundation, Inc...(no debugging symbols found)...
Connected to com1 at 115200 bps.
(gdb960)
section 0, name .text, address 0xc0008000, size 0x50ec, flags 0x20
          writing section at 0xc0008000
```

Verify that the parallel port you are using is the one you specified in the gdb960 command line. Verify that your parallel cable is properly connected to the board and to your PC.

Data Cache Tutorial

The i960 Hx processors feature an 8-Kbyte, direct-mapped data cache that is write-through and write-allocate. These processors have a line size of four words. Each line in the cache has a valid bit; to reduce fetch latency on cache misses, each word within a line also has a valid bit.

The purpose of the `dcache.c` program is to show the performance advantage that can be obtained by the use of the data cache on the i960 Hx microprocessor.

This example uses the Minimum Edit Distance (MED) algorithm in order to show the effectiveness of using the data cache. The MED algorithm finds the minimum number of edit steps required to change one string into another.

This example is a real world example of using the data cache. This algorithm maintains a cost matrix to determine which change to the string being edited would incur the least cost. The cost matrix is a 2-D array [1..n][1..m], where n and m are the sizes of the two strings.

The algorithm really shows the speed of the data cache due to three reads for each write to the cost matrix. The algorithm reads from the cache to determine which step to take next, then writes its choice in the cost matrix. Since the writes to the data cache are write-through, there is no improvement for writes to the data cache. The Write-Through feature maintains coherency between the data cache and external memory.

The source code includes system files, `system.c` and `system.h`, that includes a macro and an assembly function that simplifies issuing data cache control instructions.

Also, the example shows how to define an area of memory to make data cacheable by using the Logical Memory Configuration (LMCON) registers. The address of the area to make cacheable is programmed into the Logical Memory Address Register (LMADR). The mask is programmed into the Logical Memory Mask Register (LMMR).

1. Choose **Cache Examples**.
2. Choose **Data Cache**.
3. Choose **Qv Code**.

4. Scroll through the `dcache.c` code to see the calls to the macro, `dcctl_contrl`.
5. Open and scroll through the `system.h` and `system.c` code to see the macro and assembly function, `dcctl_control` and `i960_dcctl`.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `dcache`. Type:

   `run`

   The debugger responds by displaying:

```
Minimum Edit Distance algorithm makes reads from the data cache.
This routine will determine how many steps are needed to convert:
StringA:  80960 QUICKval EvalKit
TO StringB:  i960(R) HxJxCxSx & Kx
Starting timed routine with data cache on ...
RESULT: 18 moves are required to convert string A to string B
Elapsed Time On = 0.002869 seconds
Elapsed Time for routine with data cache off ...
RESULT: 18 moves are required to convert string A to string B.
Elapsed Time Off = 0.003360 seconds
IMPROVEMENT: 14.6 percent
(gdb960)
```

8. Type: `quit`
9. Select **Results**.

NOTE. *Your actual run times may vary.*

Instruction Cache Tutorial

The i960 Hx processor comes equipped with 16 KB of four-way set-associative instruction cache. The instruction cache boosts your application's performance by reducing the number of instruction fetches from external memory. The cache provides fast execution of code and loops of code in the cache.

The `loop.c` program demonstrates the performance boost obtained by running a loop completely within versus outside of the instruction cache.

The source code includes system files, `system.c` and `system.h` that includes a macro and an assembly function that simplifies issuing instruction cache control instructions.

1. Choose **Cache Examples**.
2. Choose **Instruction Cache**.
3. Choose **Qv Code.**
4. Scroll through the `loop.c` code to see the calls to the macro, `icache_control`.
5. Open and scroll through the `system.h` and `system.c` code to see the macro and assembly function, `icache_control` and `i960_icctl`.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `loop`. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/loop
   Simple loop timed with instruction cache off ...
   Elapsed Time Off = 2.824 seconds
   Simple loop timed with instruction cache on ...
   Elapsed Time On = 0.509 seconds
   IMPROVEMENT : 82.0 percent
   Program exited with code 01
   (gdb960)
   ```
8. Type: **`quit`**
9. Select **Results**.

Register Cache

The i960 Hx processor provides fast storage of local registers for call and return operations by using an internal local register cache. Up to fifteen local register sets can be contained in the cache before sets must be saved in external memory. The default cache size is five register sets. The register set is all the registers (i.e., r0 through r15). The processor uses a 128-bit wide bus to store local register sets quickly to the register cache.

To decrease interrupt latency, software can reserve a number of frames in the local register cache solely for high priority interrupts (interrupted state and process priority greater than or equal to 28). When a frame is reserved for high-priority interrupts, the local registers of the code interrupted by a high-priority interrupt can be saved to the local register cache without causing a frame flush to memory.

This program demonstrates the use of the on-chip register cache in reducing the interrupt latency for high priority interrupts. First, high priority interrupts are timed using the register cache, then low priority interrupts are timed without the use of the register cache.

1. Choose **Cache Examples.**
2. Choose **Register Cache**.
3. Choose **Qv Code**.
4. Scroll through the `reg_int.c` code and find the `PRCB_Ptr -> reg_cache_config` assignment. That is where the Register Cache Configuration Word in the Processor Control Block gets written. It is assigned to allocate all 15 frames for high priority interrupts.
5. Open and scroll through the `high_int.s` and `low_int.s` files. The `high_int.s` file contains the interrupt handling procedure for the high priority interrupts, and the file `low_int.s` contains the interrupt handling procedure for the low priority interrupts.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `reg_int`. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\quickval/reg_int
   Triggering Interrupts ... Register Cache USED ...
   RESULT: Timeon is 0.000044 seconds
   Triggering Interrupts ... Register Cache NOT USED ...
   RESULT: Timeoff is 0.000058 seconds
   IMPROVEMENT: 24.1 percent
   Program exited with code 01.
   (gdb960)
   ```
8. Exit the debugger, type: `quit`
9. Select **Results**.

External Interrupts Tutorial

The purpose of this program, `cyint.c`, is to show the steps required when dealing with an interrupt triggered externally by the evaluation board timers. The `cyint.c` source code contains step-by-step instructions to save you time when you program interrupts for your application. `int_proc.s` is the interrupt handler, and `t85c36.c` contains the functions to program the evaluation board timers.

The example performs the following steps in the handling of a hardware interrupt.

- Modify the ICON register
- Modify the IMAP register
- Cache the interrupt vector and the interrupt handling procedure
- Lower the processor priority
- Modify the IMSK register
- Clear the IPND register
- Generate the hardware interrupt using the evaluation board timers

Complete these steps:

1. Choose **Interrupt Examps**.
2. Choose **External Interrupts**.
3. Choose **Qv Code**.
4. Scroll through the `cyint.c` source to see the code for setting up and handling a hardware interrupt triggered by the evaluation board timers.
5. Open and scroll through the `t85c36.c` and `t85c36.h` files to see the definitions and routines for programming the evaluation board timers. You can simplify the programming of the evaluation board timers by including this code in your own applications.
6. Choose **Make** to compile, link, and download the program automatically.

7. Use the gdb960 debugger to execute `cyint`. Type:

   `run`

   The debugger responds by displaying:

   ```
   interrupt count = 60
   interrupt count = 72
   interrupt count = 84
   interrupt count = 95
   interrupt count = 107
   interrupt count = 119
   interrupt count = 131
   interrupt count = 143
   interrupt count = 155
   Program exited with code 020.
   (gdb960)
   ```

8. Type: **`quit`**

NOTE. *Your actual interrupt counts may vary.*

Internal Interrupts Tutorial

A key feature of the i960 Hx processor are the dual, fully independent 32-bit timer units. Each is programmed by use of the timer registers. These registers are memory-mapped within the processor, addressable on 32-bit boundaries. The timers have a single shot mode and auto-reload capabilities for continuous operation. Each timer has an independent interrupt request to the processor's interrupt controller. Each timer can generate a fault when it detects unauthorized writes from user mode.

The `timrcntr.c` program demonstrates how to use the structures and routines found in `timers.c` to easily program either timer to cause periodic interrupts.

1. Choose **Interrupt Examps**.
2. Choose **Internal Interrupts**.
3. Choose **Qv Code**.
4. Scroll through the `timrcntr.c` code to see the code for setting up a timer to cause a hardware interrupt.
5. Open and scroll through `timers.c` and `timers.h` files to see the definitions and routines for programming the on-chip timers. You can simplify the programming of the timer by including this code in your own applications.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `timrcntr`. Type:

   `run`

   The debugger responds by displaying the current count of each timer every time timer0 causes an interrupt.
8. Type: **`quit`**

Fault Handling

These programs, `fault.c`, `flt_proc.c`, `asm_flt.s`, and `system.c`, show the steps taken in setting up the fault handling procedures in the fault and system procedure tables. The faults are then triggered one by one.

Table 5-2 i960 Hx Processor Fault Types and Subtypes

Fault Type		Fault Subtype		Fault Record
Number	**Name**	**Number or Bit Position**	**Name**	
0H	PARALLEL	NA		See your microprocessor user’s manual
1H	TRACE	Bit 1	INSTRUCTION	0001 0002H
		Bit 2	BRANCH	0001 0004H
		Bit 3	CALL	0001 0008H
		Bit 4	RETURN	0001 0010H
		Bit 5	PRERETURN	0001 0020H
		Bit 6	SUPERVISOR	0001 0040H
		Bit 7	MARK	0001 0080H
2H	OPERATION	1H	INVALID_OPCODE	0002 0001H
		2H	UNIMPLEMENTED	0002 0002H
		3H	UNALIGNED	0002 0003H
		4H	INVALID_OPERAND	0002 0004H
3H	ARITHMETIC	1H	INTEGER_OVERFLOW	0003 0001H
		2H	ZERO-DIVIDE	0003 0002H
4H	Reserved			
5H	CONSTRAINT	1H	RANGE	0005 0001H
6H	Reserved			
7H	PROTECTION	Bit 1	LENGTH	0007 0002H
		Bit 5	BAD_ACCESS	0007 0020H
9H	Reserved			
8H	MACHINE	2H	PARITY_ERROR	0008 0002H
AH	TYPE	1H	MISMATCH	000A 0001H
BH - FH	Reserved			
10H	OVERRIDE	NA	NA	NA

1. Choose **Fault Handling**.
2. Choose **QV Code**.
3. Scroll through the `fault.c` code to see a call to the function load_flt_proc(). This function loads the fault handling procedures into the fault and/or the system table.
4. Open and scroll through the `flt_proc.c` and `asm_flt.s` files. The `flt_proc.c` file contains the fault handling procedures, and the file `asm_flt.s` is used to help generate the faults.
5. Choose **Make** to compile, link, and download the program automatically.

NOTE. *When compiling, disregard the compiler warning:*

```
Warning: unaligned register
```

This is one of the faults that will be handled.

6. Use the gdb960 debugger to execute `fault`. Type:

   `run`

   The debugger responds by displaying the Fault Type and the Fault Subtype for each fault handled. The address of the faulting instruction is given (see Table 5-2).
7. Type: **`quit`**

Static, Global, and Profile-Driven Optimizations

Optimizing compilers provide you with a means of developing high performance code without detailed knowledge of the architecture. Engineers who understand the features of the i960 architecture developed gcc960 to provide optimizations that take full advantage of the i960 processor. In general, optimizing compilation takes more time and may require more memory for large functions. However, the benefit in runtime performance is well worth it.

There are several levels of optimization available. Typically, low levels of optimizations are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once your application is functioning properly, you can increase its runtime performance by using a higher level of optimization.

Release 5.0 and later of the development tools support the ELF object module format and DWARF version 2.0 debug information format. The new format enables more accurate mapping between source and object code at higher optimization levels and ease debugging of production code.

The C optimization example uses a program called `chksum.c`. The C++ examples use a program called `optimize.cpp`.

C No Optimization

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.
4. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 11.776468 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**

C Static Optimization

Use the following commands to compile the `chksum.c` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.

4. Choose **Make -O4** to compile with optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 0.988957 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ No Optimization

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.
5. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 4.93137 seconds.
   Program exited normally
   ```
7. Type: **`quit`**

C++ Static Optimization

Use the following commands to compile the `optimize.cpp` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.

3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.
5. Choose **Make -O4** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 3.93345 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

C Global Optimization

Use the following commands to compile the `chksum.c` with program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C Global Optimizations**.
4. Choose **Make +O5** to compile with optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 1.167572 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ Global Optimization

Use the following commands to compile the `optimize.cpp` program using the program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler**.
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Global Optimizations**.
5. Choose **Make+05** to compile with optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 3.90345 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation

A 92% improvement in C code performance is significant, but there is another level of optimization that is uniquely available through Intel's CTOOLS compilers: profile-driven optimization. This two-pass compilation procedure allows the compiler to make optimizations based on runtime behavior as well as the static information used by conventional 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. Standard (i.e., non-program-wide) optimizations are referred to as 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 before the end of the linking phase.

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.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Profiling Lab**.
4. Follow the **Profiling Tutorial** link in the online help.

Using profile-driven optimization, an increase in runtime performance of 1% is obtained. The average 80960 application can expect to gain 15 to 30% performance improvement through the use of this technology. This boost in performance is available to you without any further investment in hardware.

C++ Virtual Function Optimizations

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, however, 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.

Use the following commands to compile the `optimize.cpp` program.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.

4. Choose **C++ Virtual Opts**.
5. Choose **Make -NoVOpt** to compile without virtual function optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 3.90345 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Make -VOpt** to compile with virtual function optimizations, link, and download the program automatically.
9. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 3.54671 seconds.
   Program exited normally
   ```
10. Type: **`quit`**
11. Choose **Results**.

The virtual function optimizations yielded a 9% improvement.

Note the runtime performance at each optimization level as shown below.

Table 5-3 i960 Hx Processor Optimization Results

Optimization Level	C Execution Time	C++ Execution Time
no optimization (-O0)	11.776468 seconds	4.93137 seconds
maximum static (-O4)	0.988957 seconds	3.93345 seconds
global optimization	1.167572 seconds	3.90345 seconds
profile-driven	0.975972seconds	NA
Virtual Function Optimization	NA	3.54671 seconds

Building Self-contained Profiles with gmpf960

A *raw* profile contains program counters that record 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 has a very short useful life. When changes are made in the 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 used to optimize a program 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.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Self-Contained**.
4. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
5. Specify the program database directory.

   The PDB can be specified by setting the environment variable `G960PDB`.

   For example, if you chose the default directory during installation, enter:

   ```
   SET G960PDB=C:\quickval\prof_lab\lab_pdb
   ```

Or, specify the PDB at compiler invocation time with the `Z`*`dir`* option, as shown in the example below.

```
gcc960  -Zmypdb  foo.o
```

6. Compile for profile instrumentation.

   Insert profile instrumentation into *quick* so that when the linked program is executed, a profile can be collected. Type:

   ```
   gcc960 -Fcoff  -T{Link-dir} -A{arch} -fdb
   -gcdm,subst=:*+fprof -o quick quick.c
   ```

   The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-gcdm,subst=:*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only eligible modules not linked in from libraries.
`+fprof`	causes generation of profile instrumentation.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

7. Collect 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. Type:

   ```
   gdb960 -t mon960 -b 115200 -r com1 -D lpt1 quick
   ```

The options in this gdb960 compiler command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`quick`	the executable file

8. Use the gdb960 debugger to execute `quick`. Enter:
   `run`
9. Exit the debugger. Enter:
   `quit`
10. Enter the command:
    `gmpf960 -spf quick.pf default.pf`
    The options in this gmpf960 compiler command are:

`-spf`	causes a self-contained profile, `quick.pf`, to be produced as output.
`default.pf`	The input profile.

11. Recompile the `quick.c` source code using the profiling information obtained by the instrumentation. Type:
    `gcc960 -Fcoff -T`*`{Link-dir}`*`-A`*`{arch}`*`-fdb`
    `-gcdm,iprof=quick.pf -o quick quick.c`
    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=quick.pf`	This supplies a profile file `quick.pf` to the global decision making and optimization step.

`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

12. Change the control structure of `quick.c.`

    Edit `quick.c`. Find the procedure called QUICK. In this procedure, there is a control structure:

```
for(i = 2; i <= SORTELEMENTS; i+=1)
{
    (LOGIC)
}
```

    Change the control structure to:

```
i = 2;
while (i <= SORTELEMENTS)
{
    (LOGIC)
    i+=1;
}
```

13. Compile the new `quick.c` using the interpolated profile. Type:

```
gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
-gcdm,iprof=quick.pf -o quick quick.c
```

    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=quick.pf`	This supplies a profile file `quick.pf` to the global decision making and optimization step.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

Notice that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`.

NOTE. *The beauty of this example is that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`, not the results of running this example.*

Profiling a Program in Pieces

Suppose that the target execution environment is memory limited so that all your programs cannot be instrumented for profiling at the same time. You can use substitutions to make partially instrumented versions of the final executable, and then create self-contained profiles for each piece. Each executable created in this way has a limited set of instrumented modules.

After you've created the self-contained profiles, you can use gmpf960 to create a single merged self-contained profile. The final, merged self-contained profile is identical to a profile obtained by instrumenting the entire program at once.

In this example, you use the fault handling example programs to show incremental profiling.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations.**
3. Choose **Incremental**.
4. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

5. Specify the program database directory.

   You can specify the PDB by setting the environment variable `G960PDB`. For example, if you chose the default directory during installation, enter:

   ```
   SET G960PDB=C:\quickval\prof_lab\lab_pdb
   ```

   Or, specify the PDB at compiler invocation time with the `Zdir` option, as shown in the example below.

   ```
   gcc960  -Zmypdb  foo.o
   ```

6. Insert profile instrumentation into `fault` so that when the linked program is executed, a profile can be collected. The instrumented modules in this version of `fault` are from the files `fault.c` and `flt_proc.c`. Type:

   ```
   gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
   -gcdm,subst=:f*+fprof -o fault fault.c flt_proc.c
   asm_flt.s system.c
   ```

`-Fcoff`	creates a COFF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-fdb`	all modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-gcdm,subst=:f*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only the files that begin with *f*.
`+Fprof`	causes generation of profile instrumentation.
`-o fault`	names the executable file `fault`.
`fault.c`	the source files.
`flt_proc.c`	the fault procedures.

`asm_flt.s`	the assembly file to generate faults.
`system.c`	system file.

7. Collect the profile.

   When 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. Type:

   ```
   gdb960 -t mon960 -b 9600 -r com1 -D lpt1 fault
   ```

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`fault`	the executable file

8. Use the gdb960 debugger to execute `fault`. Enter:

   ```
   run
   ```

9. Exit the debugger. Enter:

   ```
   quit
   ```

10. Build the self-contained profiles with gmpf960.

    To create a self-contained profile, use the gmpf960 profile merger tool. gmpf960 is invoked with the raw profile as an input file. Enter:

    ```
    gmpf960 -spf prof1.pf default.pf
    ```

`-spf`	causes a self-contained profile, `prof1.pf`, to be produced as output.
`default.pf`	The input profile.

    The resultant self-contained profile, `prof1.pf`, has a limited set of instrumented modules.

11. Insert profile instrumentation into `fault` so that when the linked program is executed, a profile can be collected. The instrumented modules in this version of `fault` are from the file `system.c`. Type:

    ```
    gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
    -gcdm,subst=:s*+fprof -o fault fault.c flt_proc.c
    asm_flt.s system.c
    ```

`-Fcoff`	create a COFF format output file

`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD.
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,subst=:s*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only the files that begin with `s`.
`+fprof`	causes generation of profile instrumentation.
`-o fault`	names the executable file `fault`.
`fault.c`	the source files
`flt_proc.c`	the fault procedures
`asm_flt.s`	the assembly file to generate faults
`system.c`	system file

12. Collect the 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. Type:

    ```
    gdb960 -t mon960 -b 9600 -r com1 -D lpt1 fault
    ```

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`fault`	the executable file

13. Use the gdb960 debugger to execute `fault`. Enter:

    `run`

14. Exit the debugger. Enter:

    `quit`

15. Build the self-contained profiles with gmpf960.

    To create a self-contained profile, use the gmpf960 profile merger tool. gmpf960 is invoked with the raw profile as an input file. Enter:

    `gmpf960 -spf prof2.pf default.pf`

`-spf`	causes a self-contained profile, `prof2.pf`, to be produced as output.
`default.pf`	the input profile.

    The resultant self-contained profile, `prof2.pf`, has a limited set of instrumented modules.

16. Merge all the self-contained profiles into one.

    The final `prof.pf` profile is identical to a profile obtained by instrumenting the entire program at once. Type:

    `gmpf960 -spf prog.pf prof1.pf prof2.pf`

`-spf`	causes a self-contained profile, `prog.pf`, to be produced as output.
`prof1.pf`	an input self-contained profile.
`prof2.pf`	an input self-contained profile.

17. Recompile the fault handling source code using the profiling information obtained by the instrumentations. Type:

    ```
    gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
    -gcdm,iprof=prog.pf -o fault fault.c flt_proc.c
    asm_flt.s system.c
    ```

`-Fcoff`	create a COFF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-fdb`	all modules subject to program-wide optimization must be initially compiled with the `fdb` option.

`-Gcdm,iprof=prog.pf`	
	This supplies a profile file `prog.pf` to the global decision making and optimization step.
`-o fault`	`names the executable file fault.`
`fault.c`	the source file.
`flt_proc.c`	the fault procedures.
`asm_flt.s`	the assembly file to generate faults.
`system.c`	system file.

NOTE. *The beauty of this example is the methodology of incremental profiling, not the result of running the example.*

Compression Assisted Virtual Execution (CAVE)

This CTOOLS feature allows non-critical parts of an application's machine code to be stored in memory in compressed form resulting in reduced target memory requirements. The code is expanded into native machine code on demand for 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 of 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, 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, that 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.

Note that due to the overhead of decompressing code at runtime, only non-performance critical code should be secondary functions, such as error handling code or initialization code. You can use runtime profile information generated by gcov960 to aid in selecting the set of secondary functions.

This example uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.

For the sake of demonstration, we compress performance-critical code in the tic-tac-toe program. The purpose of this example is to show the reduced text section of the executable, not demonstrate run times.

C Example

1. Choose **Compiler**.
2. Choose **C Cave**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Use the gcc960 `mcave` option or `#pragma cave` to designate the specified functions as secondary. In the tic-tac-toe example, `ttt.c`, the following `#pragma` has been added:

   ```
   #pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)
   ```

   where `Initialize, Winner, Other, Play, Evaluate, Best_Move, Describe, Move,` and `Game` are all functions to be compressed.
5. Edit `ttt.c`. Make sure the `#pragma cave` program line is commented out:

   ```
   /*#pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)*/
   ```
6. Compile the tic-tac-toe program. Enter:

```
gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c
```

The options in this command are:

`-Fcoff`	create a COFF format output file

`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies mcyhx.gld.
`-o ttt`	names the executable file ttt
`ttt.c`	input file

7. Check the text section size of the uncompressed program. Enter:

   `gsize960 ttt`

   The option in this command is:

`ttt`	name of the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.

8. Edit `ttt.c`. Make sure the `#pragma cave` program line is uncommented:

   ```
   #pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)
   ```

9. Compile the tic-tac-toe program with the pragma program line. Enter:

   `gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c`

   The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies mcyhx.gld.
`-o ttt`	names the executable file ttt
`ttt.c`	input file

10. Check the text section size of the compressed program. Enter:

    `gsize960 ttt`

    The option in this command is:

`ttt`	executable file

The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 5-4 Uncompressed Text Sections

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	33,764	32,944	32,768	32,976	31,600

Table 5-5 After Function Compression

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	31,908	30,832	30,816	30,832	29,648
Cave Section	1,818	1,770	1,746	1,800	1,776
Total	33,726	32,602	32,562	32,632	31,424

Table 5-6 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
0.1%	1.0 %	0.6 %	1.0 %	0.6 %

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

C++ Compression Assisted Virtual Execution (CAVE)

1. Choose **Compiler**.
2. Choose **C++ Cave**.
3. Choose **Make**.The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.

4. Use the *gcc960* `mcave` option or `#pragma cave` designate the specified functions as secondary. In the C++ example, `cavecpp.cpp`, the following `#pragma` has been added:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

   where `initSetName`, `initSetDept`, `initSetGpa`, `initSetNumPubs`, `isOutstanding`, `printName`, and `InitializeRecords` are all functions to be compressed, i.e., all functions are secondary functions. All other functions of the program are primary functions.

   The primary set should contain performance-critical functions that are not to be affected by the CAVE mechanism; the secondary set is subject to compression. Secondary functions are compressed by the linker and reside in memory in compressed form.

   The C++ compiler behaves in essentially the same manner as the C compiler when the mcave or Gcave options are used - generating all functions in the compilation unit for which this option is in effect as secondary.

   A user typically designates a single function as secondary through the use of `pragma cave`. The following statement for example designates the function max as secondary.

   ```
   # pragma cave max
   ```

   However in C++ overloaded functions have the same name. Member functions of two different classes are also allowed to have the same name and these member functions can in turn have the same name as a function with file scope.

   When a user specifies a function as secondary through the use of `pragma cave`, the C++ compiler treats all functions with this name as secondary. To illustrate, consider the following example:

   ```
   # ifdef PRAGMA
   # pragma cave max
   # endif

   int max(int a, int b)
   {
   return a > b ? a : b;
   }
   ```

```
float max(float a, float b)
{
return a > b ? a : b;
}

class Tclass1 {
int a, b;
public:
int max();
};

int Tclass1::max()
{
return a > b ? a : b;
}

class Tclass2 {
float a, b;
public:
float max();
};


float Tclass2::max()
{
return a > b ? a : b;
}

Tclass1 t1;
Tclass2 t2;
```

The Compiler treats all the following functions as secondary.

```
int max(int, int);
float max(float, float);
int Tclass1::max();
float Tclass2::max();
```

5. Choose **Qv Code**. Edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is commented out:

```
//#pragma
cave(initSetName,initSetDept,initSetGpa,initSetNumPu
bs,isOutstanding,printName,InitializeRecords)
```

6. Compile the C++ program. Enter:

   `gcc960 -A{`*`arch`*`} -Felf -T{`*`Link-dir`*`} -stdlibcpp -o cavecpp cavecpp.cpp`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A{`*`arch`*`}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{`*`Link-dir`*`}`	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `cavecpp`
`cavecpp.cpp`	input file

7. Check the text section size of the uncompressed program. Enter:

   ```
   gsize960 cavecpp
   ```

   The option in this command is:

`cavecpp`	specifies the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.

8. Choose **Qv Code** and edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is uncommented:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

9. Compile the C++ program with the pragma program line. Enter:

   `gcc960 -A{`*`arch`*`} -Felf -T{`*`Link-dir`*`} -stdlibcpp`
   `-o cavecpp cavecpp.cpp`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, -AHD specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.

`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `ttt`
`cavecpp.cpp`	specifies the input file

10. Check the text section size of the compressed program. Enter:

    `gsize960 cavecpp`

    The option in this command is:

`cavecpp`	executable file

The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 5-7 Uncompressed Text Sections

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	89,788	84,196	83,512	84,196	81,764

Table 5-8 After Function Compression

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	87,612	81,892	81,512	81,892	79,796
Cave Section	1,920	1,546	1,514	1,546	1,512
Total	89,532	83,438	83,026	83,438	81,308

Table 5-9 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
1%	1%	1%	1%	1%

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

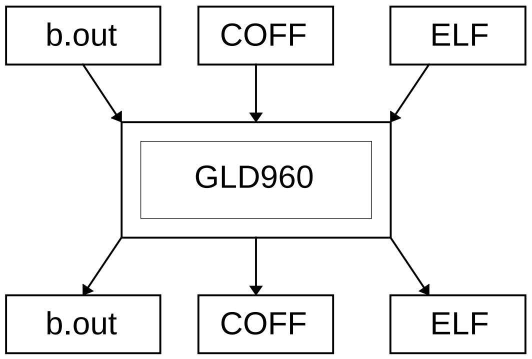
Linker Consumption

You can link b.out-format, COFF or ELF object files and libraries in any combination. To determine a file format, the linker examines the first two bytes of the file. An unrecognized value indicates a linker-directive file. This feature is useful when using third-party archives with CTOOLS runtime libraries and your application code. The runtime libraries are shipped in ELF format *only* (effective with the 5.0 version of the tools). Each can potentially have a different OMF, and the linkage still completes.

NOTE. *As of version 5.0 of the tools, all runtime libraries are shipped in ELF format only.*

If the linker generates a different output format than the input, the linker does not copy debug information from the input file to the output file. Because of this, you should use only one OMF.

The symbol tables of each OMF are abbreviated when crossing OMF boundaries. For example, when you include a b.out OMF file in a linkage where the output file OMF is COFF format, none of the debug information from the b.out file is copied into the output COFF file.



1. Choose **Linker and Utilities**.
2. Choose **Linker Consumption**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Compile the first file in COFF format. Enter:

   `gcc960 -Fcoff -A`*`{arch}`*` -c t85c36.c`

   The options in this command are:

`-Fcoff`	creates a COFF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-c`	compile, but do not link.
`t85c36.c`	input file.

5. Compile the second file in ELF format. Enter:

   `gcc960 -Felf -A`*`{arch}`*` -c system.c`

   The options in this command are:

`-Felf`	creates an ELF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-c`	compiles, but does not link.
`system.c`	input file.

6. Compile the third file in b.out format. Enter:

   ```
   gcc960 -Fbout -A{arch} -c -r cyint.c int_proc.s
   ```

   The options in this command are:

`-Fbout`	creates a b.out format output file.
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-c`	compiles, but does not link.
`-r`	allows unresolved references.
`cyint.c`	the source file.
`int_proc.s`	the interrupt handler.

7. Generate an absolute file in ELF format by linking files in b.out-format, ELF format, and COFF format. The absolute file could have also been in b.out-format or COFF format. Enter:

   ```
   gld960 -Felf -T{Link-dir} -A{arch} -o elf t85c36.o
   system.o cyint.o int_proc.o
   ```

   The options in this command are:

`-Felf`	specifies the absolute file as ELF format.
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tcyhx` specifies `cyhx.gld`.
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-o elf`	names the executable file `elf`.
`cyint.o`	file in b.out-format.
`int_proc.o`	file in b.out-format.
`t85c36.o`	file in COFF format.
`system.o`	file in ELF format.

NOTE. *The beauty of this example is the functionality of the linker, not the result of running the example.*

Assembler Pseudo-instruction Tutorial

This tutorial demonstrates the use of pseudo-instructions that have been added to the CTOOLS assembler to ease migration between processors. The tutorial that follows demonstrates how to enable and disable the instruction cache for the i960 Cx, Hx, Jx, and Rx microprocessors using microprocessor specific instructions. The tutorial then demonstrates how easy it is to enable and disable the instruction cache using only one pair of pseudo-instructions.

What Are Pseudo-instructions?

A number of pseudo-instructions (pseudo-ops) have been added to the CTOOLS assembler to ease the migration between processors. These pseudo-ops provide an architecture-independent method for performing some of the more common low-level processing operations. Using these pseudo-ops should reduce the number of changes required when moving assembly code from one i960 processor to another.

When you use any of the new i960 pseudo-instructions, you are required to re-assemble your source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best processor instructions to replace the pseudo-instructions based on the processor targeted.

pseudop.c: Editing the File for the Cx Microprocessor

1. If you are using the Hx Jx Cx & Sx QUICK*val* software, choose **Linker** and **Utilities**. If using the Rx QUICK*val* software, this step is not necessary.
2. Choose **Pseudo-op Tutorial**
3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.

4. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`. Both procedures contain no code initially. `cache_off()` looks like:

   ```
   cache_off()
   {

   }
   ```

5. Add the code necessary to disable the instruction cache for the Cx microprocessor. Between the brackets of the `cache_off()` procedure, add the following line exactly:

   ```
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   ```

   The `cache_off()` procedure should look like this:

   ```
   cache_off()
   {
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   }
   ```

   This procedure, `cache_off()`, uses the instruction cache control processor instruction `sysctl`. This instruction is valid in the i960 Cx processor for managing and controlling the instruction cache. `sysctl` is used above to disable the instruction cache. Also, the `CONFIGURE_ICACHE` and `DISABLE_ICACHE` constants are found in the `system.h` file that is included in the `pseudop.c` file.

6. Likewise, edit the `cache_on()` procedure adding the following line exactly:

   ```
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (ENABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   ```

The cache_on() procedure should look like this:

```
cache_on()
{
__asm__ __volatile__("sysctl %0,%1,%2" ::"d"
(((CONFIGURE_ICACHE)<<8)|
(ENABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
}
```

Similarly, the `cache_on()` procedure for the Cx microprocessor uses the instruction cache control processor instruction `sysctl`. `sysctl` is used directly above to enable the instruction cache.

7. Save the `pseudop.c` file.

Running pseudop.c for the Cx Microprocessor

1. Compile and run the `pseudop.c` program to show that it works as desired.

NOTE. *If you do not have an i960 Cx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

2. In the Command Prompt window, enter the following commands:

```
gcc960 -AC{F/A} -Fcoff -Tmcycx -o pseudop pseudop.c
```

The options in this command are:

`-AC{F/A}`	sets the target architecture for the compiler. For this example, choose the Cx architecture, `-ACF` or `-ACA`
`-Fcoff`	sets the object file type as coff.
`-Tmcycx`	sets the linker directive file for the Cx architecture.
`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

If you have a Cx microprocessor and want to run the program, enter:

`gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

3. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works with instructions chosen specifically for the Cx architecture. Of course, this is what is expected. This program becomes more interesting when you start using pseudo-instructions.*

4. At the (gdb960) prompt, enter: **`quit`**

pseudop.c: Migrating the File to the Jx/Hx/Rx Microprocessor

Since the i960 Jx, Hx, and Rx microprocessors use the same processor instruction to enable and disable the instruction cache, this migration supports all three processors.

In order to use the program, `pseudop.c`, modified in the first part of this tutorial to support the Jx, Hx, or Rx microprocessor, it must first be migrated to those processors since they do not use the `sysctl` instruction to enable and disable the instruction cache.

1. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`.

   `cache_off()` contains the Cx specific code and looks like:

   ```
   cache_off()
   {
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   }
   ```

2. Change the code to disable the instruction cache for the i960 Jx/Hx/Rx microprocessors. Between the brackets of the `cache_off()` procedure, delete the previously added line and insert the following line exactly:

   ```
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   ```

   The `cache_off()` procedure should now look like this:

   ```
   cache_off()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   }
   ```

   This procedure, `cache_off()`, uses the instruction cache control processor instruction `icctl`. This instruction is valid in the 80960 Jx/Hx/Rx processors for managing and controlling the instruction cache. `icctl` is used above to disable the instruction cache. Also, the `ICACHE_OFF` constant is found in the `system.h` file that is included in the `pseudop.c` file.

3. Likewise, edit the `cache_on()` procedure deleting the previously added line and inserting the following line exactly:

   ```
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_ON),"d"(0),"d"(0));
   ```

   The `cache_on()` procedure should now look like this:

   ```
   cache_on()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_ON),"d"(0),"d"(0));
   }
   ```

   Similarly, the `cache_on()` procedure for the Jx/Hx/Rx microprocessors use the instruction cache control processor instruction `icctl`. `icctl` is used directly above to enable the instruction cache.
4. Save the `pseudop.c` file.

Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors

1. Compile and run the pseudop.c program to show that it works as desired.

NOTE. *If you do not have an i960 Jx, Hx, or Rx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

2. In the Command Prompt window, enter the following commands:

   For the Jx Microprocessor:

   `gcc960 -AJ{`*`F|D|A`*`} -Fcoff -Tmcyjx -o pseudop pseudop.c`

   The options in this command are:

`-AJ{`*`AF|D|T`*`}`	sets the target architecture for the compiler. For this example, to choose the Jx architecture, `-AJA`, `-AJF`, `-AJD`, or `-AJT`
`-Fcoff`	sets the object file type as coff.
`-Tmcyjx`	sets the linker directive file for the Jx architecture.

`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

For the Hx Microprocessor:

`gcc960 -AH{D|A} -Fcoff -Tmcyhx -o pseudop pseudop.c`

The options in this command are:

`-AH{`*`D`*`|`*`A`*`}`	sets the target architecture for the compiler. For this example, to choose the Hx architecture, `-AHD` or `-AHA`
`-Fcoff`	sets the object file type as coff.
`-Tmcyhx`	sets the linker directive file for the Hx architecture.
`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

For Rx Microprocessor:

`gcc960 -AR{`*`P`*`|`*`D`*`} -Fcoff -Tmcyrx -o pseudop pseudop.c`

The options in this command are:

`-AR{`*`P`*`|`*`D`*`}`	sets the target architecture for the compiler. For this example, to choose the Rx architecture: `-ARP` or `-ARD`
`-Fcoff`	sets the object file type as coff.
`-Tmcyrx`	sets the linker directive file for the Rx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

3. If you have a Jx, Hx, or Rx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works with instructions chosen specifically for the architecture in question. Of course, this is what is expected. This program becomes more interesting when* you *start using pseudo-instructions.*

5. At the (gdb960) prompt, enter: **`quit`**

pseudop.c: Adding Pseudo-Ops to the Program

As can be seen, it is neither easy nor fun migrating code from one processor to another, especially when your code is many thousands of lines long. Fortunately, pseudo-instructions have been added to the CTOOLS assembler to ease migration between processors.

1. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. You are ready now to rewrite this program using pseudo-instructions. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`.

   `cache_off()` contains the i960 Jx/Hx/Rx microprocessor specific code:

   ```
   cache_off()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   }
   ```

2. Change the code to disable the instruction cache for ALL processors. Between the brackets of the `cache_off()` procedure, delete the previously added line and insert the following line exactly:

   ```
       __asm__ __volatile__("ic_disable r5");
   ```

   The `cache_off()` procedure should now look like this:

   ```
   cache_off()
   {
       /* local register r5 is used to hold the status
   returned */
       __asm__ __volatile__("ic_disable r5");
   }
   ```

   This procedure, `cache_off()`, uses the pseudo-instruction `ic_disable`. When this program, `pseudop.c`, is compiled for a specific 80960 processor by using a `-A` architecture flag, the best instructions for that architecture are chosen to replace the `ic_disable` pseudo-op. Thus, pseudo-ops ease migration between processors. Also, notice only one argument to the pseudo-op is necessary. The `icctl` instruction requires three arguments. Programming with pseudo-ops can be simpler. Pseudo-instructions are also available to perform the other instruction cache management and controlling functions, such as cache invalidation.

3. Likewise, edit the `cache_on()` procedure deleting the previously added line and inserting the following line exactly:

   ```
   __asm__ __volatile__("ic_enable r5");
   ```

   The `cache_on()` procedure should now look like this:

   ```
   cache_on()
   {
         /* local register r5 is used to hold the status
   returned */
       __asm__ __volatile__("ic_enable r5");
   }
   ```

   Similarly, `cache_on()` uses a pseudo-instruction: `ic_enable`. When this program, `pseudop.c`, is compiled for a specific 80960 processor, the best instruction for that architecture is chosen to replace the `ic_enable` pseudo-op.
4. Save the `pseudop.c` file.

Running pseudop.c with Pseudo-instruction

1. Compile and run the `pseudop.c` program to show that the pseudo-instructions work as desired. To prove that the best instruction is chosen for the architecture, compile the code for the Cx microprocessor and then the Jx, Hx, or Rx microprocessor.
2. In the Command Prompt window, enter the following command:

   ```
   gcc960 -AC{F/A} -Fcoff -Tmcycx -o pseudop pseudop.c
   ```

   The options in this command are:

`-AC{F/A}`	sets the target architecture for the compiler. For this example, choose the Cx architecture, `-ACF` or `-ACA.`
`-Fcoff`	sets the object file type as coff.
`-Tmcycx`	sets the linker directive file for the Cx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

   When compiled, warnings may be generated. The warnings are generated just to point out this simple fact: when you use any of the new i960 pseudo-instructions, you are required to re-assemble your

source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best instructions to replace the pseudo-instructions based on the processor targeted.

3. If you have a Cx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works as expected with pseudo-instructions.*

The result of this example is similar to using instructions specifically chosen for the Cx architecture. So, using pseudo-instructions can maintain the logic of your code, while easing migration to future i960 microprocessors.

5. At the (gdb960) prompt, enter: `quit`

NOTE. *If you do not have an i960 Jx, Hx, or Rx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors

Now you are ready to compile the code for the Jx, Hx, or Rx microprocessor to demonstrate similar results on a different processor.

1. In the Command Prompt window, enter the following commands:

   For the Jx Microprocessor:

   `gcc960 -AJ{`*`F`*`|`*`D`*`|`*`A`*`} -Fcoff -Tmcyjx -o pseudop pseudop.c`

   The options in this command are:

`-AJ{`*`A`*`|`*`F`*`|`*`D`*`|`*`T`*`}`	sets the target architecture for the compiler. For this example, to choose the Jx architecture, `-AJA`, `-AJF`, `-AJD`, or `-AJT`
`-Fcoff`	sets the object file type as coff.
`-Tmcyjx`	sets the linker directive file for the Jx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

   For the Hx Microprocessor:

   `gcc960 -AH{`*`D`*`|`*`A`*`} -Fcoff -Tmcyhx -o pseudop pseudop.c`

   The options in this command are:

`-AH{`*`D`*`|`*`A`*`}`	sets the target architecture for the compiler. For this example, to choose the Hx architecture, `-AHD` or `-AHA`
`-Fcoff`	sets the object file type as coff.

`-Tmcyhx`	sets the linker directive file for the Hx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

For Rx Microprocessor:

`gcc960 -AR{`*`P|D`*`} -Fcoff -Tmcyrx -o pseudop pseudop.c`

The options in this command are:

`-AR{`*`P|D`*`}`	sets the target architecture for the compiler. For this example, to choose the Rx architecture, `-ARP` or `-ARD`
`-Fcoff`	sets the object file type as coff.
`-Tmcyrx`	sets the linker directive file for the Rx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

2. If you have a Jx, Hx, or Rx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.

`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

3. At the (gdb960) prompt, enter: **`run`**

   The result of this example is the same as using instructions specifically chosen for the Jx, Hx, or Rx architecture. So, using pseudo-instructions does not change the logic of the program. It only eases future migration of your code to future i960 microprocessors.

4. At the (gdb960) prompt, enter: **`quit`**

   When compiled, warnings may be generated. The warnings are generated just to point out this simple fact: When you use any of the new i960 pseudo-instructions, you are required to re-assemble your source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best instructions to replace the pseudo-instructions based on the processor targeted.

CONGRATULATIONS! You can now start using pseudo-instructions in your code to ease migration of your code to future i960 processors.

Debugging with gdb960

A software debugger is a useful tool that allows you to learn more about the behavior of an application program while it is running on a target or simulator. gdb960 is a source-level debugger that allows you to interact with your application program running on a target system through the debug monitor, MON960. MON960 is resident on the Cyclone CPU module.

This example uses the card game, Go Fish, and is designed to teach you a few debugger commands so that you can further examine the example programs provided with this kit or your own programs. In the card game, Go Fish, you and the computer each get several cards. You take turns guessing which cards are in each other's hands. When you guess correctly, you acquire that card. If you don't guess correctly, you need to "Go Fish" and draw another card from the pack. When you get four-of-a-kind, you

remove those cards from your hand. The objective of the game is to have the most sets of four-of-a-kind when either you or the computer has no cards remaining in your hands.

NOTE. *This example uses the command line interface to gdb960. The program also features a Graphical User Interface in both Windows and UNIX. See The gdb960 User's Manual for more information.*

1. Choose **Debugger**.
2. Choose **gdb960 Tutorial**.
3. Choose **Make** to compile, link, and download the program automatically.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

NOTE. *DEBUGGING SHORTCUTS*
Abbreviations for gdb960 commands are accepted as long as they are unambiguous.
*To **run,** enter: **r***
*To **break,** enter: **br***
*To **list,** enter: **l***
*To **continue,** enter: **c***
*To **print,** enter: **p***
*To **clear,** enter: **cl***
*To **quit,** enter: **qu***
*For **help,** enter: **he***

4. **Do Not Type Run!** First, use the gdb960 debugger to set a breakpoint at function `main()`. Type:

   `break main`

   The debugger responds by displaying:

```
Breakpoint 1 set at 0xa0008570: file fish.c, line 209.
```

5. Set a second breakpoint at line 275. Type:

   `break 275`

   The debugger responds by displaying:

```
Breakpoint 2 set at 0xa0008bc4: file fish.c, line 275.
```

6. To execute the program from the beginning, type:

   `run`

   The debugger responds by displaying:

```
Starting program: C:\QUICKVAL/fish
Breakpoint 1, main() at fish.c, 209.
209    srand();
```

7. To display the code at the breakpoint, type:

   `list`

   The debugger displays lines 204-213 of the `fish.c` source. To see the next ten lines, type `list` again.

8. To continue executing the program from this location, type:

   `continue`

   The debugger responds by displaying:

```
Continue.
Would you like instructions[n]?
```

9. Reply by typing `y` for yes or <Enter> or `n` for no.

```
your hand is: A A 6 6 8 8 9
Breakpoint 2, game() at fish.c:275.
275    if(!move(yourhand,myhand,g=guess(),0))break;
```

10. In the source code in step 9, there are two variable arrays, `myhand` and `yourhand`. `Myhand` is the computer's hand and `yourhand` is yours. To look at the card in the computer's hand, type:

    `print myhand`

    The debugger responds by displaying:

```
$1=“000\000\000\001\000\002\000\001\000\000\001\002\000”
```

`myhand[0]` does not represent a card.

`myhand[1]` represents the number of Aces.

`myhand[2]` represents the number of 2s, and so on.

The same order of cards is represented in the array, `yourhand`.

If a King is drawn by either player, `myhand[13]` or `yourhand[13]` will appear when you print the array.

11. Using the ability to see the computer's hand, you are able to beat the computer every time. Clear the first breakpoint at the function `main()` and continue playing the game, looking at the computer's hand any time you need to. To clear the breakpoint at `main()`, type:

    ```
    clear main
    ```

    The debugger responds by displaying:

    ```
    Deleted breakpoint 1
    ```

12. To continue executing the program, type:

    ```
    continue
    ```

13. If you need further assistance beating the computer, contact the 80960 Technical Support Group for more hints.
14. Type: **`quit`**

Debugging Optimized Code

CTOOLS can use the ELF object module format and DWARF Version 2 debug information format as described in the *80960 Embedded Application Binary Interface (ABI) Specification* (order number 631999). The new formats enable more accurate mapping between source and object code at higher optimization levels and ease production code debugging.

This example shows that at the highest level of module-local optimization, it is possible to set a breakpoint on an inline function using ELF/DWARF, while with COFF this is not possible.

1. Choose **Debugger**.
2. Choose **C ELF/DWARF Format**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

4. Compile *swap.c* with no module-local optimizations (no inlining). This shows that the procedure *swap* is not inlined. Enter:

   ```
   gcc960 -Felf -T{Link-dir} -A{arch} -O0 -S swap.c
   ```

   The options in this command are:

`-Felf`	creates an ELF format output file
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD

`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-O0`	no module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

5. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, see the call to the procedure swap:

   ```
   callj _swap
   ```

   This is an out-of-line call to the procedure swap. The function swap has not been inlined.
6. Now, compile `swap.c` with the highest level of module-local optimizations. This inlines the procedure `swap`.

   ```
   gcc960 -Felf -T{Link-dir} -A{arch} -O4 -S swap.c
   ```

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-O4`	highest level of module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

7. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, note the call to the procedure `swap` does not exist:

   ```
   callj _swap  /* Does Not Exist*/
   ```

   The procedure swap has been inlined.
8. Recompile using the `-O4` optimization level, the ELF/DWARF format, and add debugging information.

```
gcc960 -Felf -T{Link-dir} -A{arch} -O4 -g -o swap swap.c
```

The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD

`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcyhx` specifies mcyhx.gld.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o swap`	names the executable file swap
`swap.c`	input file

9. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

```
gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap
```

The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`swap`	the executable file

10. **Do Not Type Run!**

First, set a breakpoint on the procedure *swap*. Enter:

```
break swap
```

The debugger responds by displaying:

```
breakpoint 1 @0xa00080f0:file swap.c, line 43
breakpoint 2 @0xa0008148:file swap.c, line 54
```

Breakpoint 1 is the out-of-line reference to the procedure `swap`. Breakpoint 2 is the inline reference to the procedure `swap`.

`Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is still possible to set a breakpoint on the inline function. Breakpoint 2 stops program execution.

11. To execute the program, enter:

```
run
```

The debugger responds by displaying:

```
Breakpoint 2, main() @ swap.c: 54
54 printf(ìThe smallest number is %d\nî,a);
```

12. To continue the program, enter:

    `c`

    When the program has finished, enter:

    `quit`

13. Compile using the `-O4` optimization level, the COFF format, and add debugging information.

```
gcc960 -Fcoff -T{Link-dir} -A{arch} -g -O4 -o swap swap.c
```

The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o swap`	names the executable file `swap`
`swap.c`	input file

14. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

```
gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap
```

The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`swap`	the executable file

15. **Do Not Type Run!!**

    First, set a breakpoint on the procedure `swap`. Enter:

    `break swap`

    The debugger responds by displaying:

```
breakpoint 1 @0xa00080f0
```

Breakpoint 1 is the out-of-line reference to the procedure `swap`. Notice that no inline breakpoint has been set. This breakpoint does not stop execution of the program.

`Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is not possible to set a breakpoint on the inline function. Program execution does not stop.

16. To execute the program, enter:

    `run`

    The debugger responds by displaying the smallest number from the swap. There is no break in program execution.

17. When the program has finished, enter:

    `quit`

    You have now seen that with the ELF/DWARF format, it is now possible to debug your production code, even after high levels of program optimization.

Debugging Optimized C++ Code Tutorial

The C++ compiler generates debug information using the DWARF format when the `-g` option is specified with the `-Felf` option. This debug information format is richer than that of other supported OMFs, and allows more reliable debugging under optimization.

This tutorial demonstrates that at the highest level of module-local optimization, debugging a C++ application is still possible due to the DWARF debug format.

In this example, you compile a C++ program using the -O0 optimization compiler option, which disables all optimizations, including those that may interfere with debugging. The same C++ program is then compiled using the highest-level of module-local optimization, -O4.

There are several levels of program optimization available with the CTOOLS development tool suite. Typically, low levels of optimization are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once the application is functioning properly, the application's performance may be increased by using a higher level of optimization. The static optimization options are:

`O0`	Turn optimization off
`O1`	Basic optimization
`O2`	strength-reduction, instruction scheduling for pipelining, etc...
`O3`	`O2` plus `fconstprop, finline-functions`, etc...
`O4`	`O3` plus `fsplit-mem, fmarry-mem, fcoalesce`

Level O4 is the highest level of static optimization. Please refer to the *i960 Processor Compiler User's Guide* for more information on ELF/DWARF and compiler optimizations.

In this tutorial, you compile and debug a C++ program, `cppdwarf.cpp`, that contains many of the advanced features of the C++ language, including:

- Classes
- Public, protected, and private variable accessibility
- Virtual functions
- Scope operators
- Overloaded functions
- Class inheritance

Using ELF/DWARF, both levels of optimization, `-O0` and `-O4`, retain the C++ program structure so that the above features may be investigated.

1. Choose **Debugger**.
2. Choose **C++ ELF/DWARF Format**.
3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Compile the program using the `-O0` optimization level. In the Command Prompt window, enter the following command:

```
gcc960 -Felf -A{arch} -T{Link-dir} -stdlibcpp -O0 -g
-o cppdwarf cppdwarf.cpp
```

The options in this command are:

`-Felf`	creates an ELF format output file.
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD.
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcyhx` specifies mcyhx.gld.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-O0`	specifies the lowest level of module-local optimizations.
`-g`	includes debug information in object file.
`-o cppdwarf`	specifies the executable file `cppdwarf`.
`cppdwarf.cpp`	specifies the input file `cppdwarf.cpp`.

5. Run the program using the debugger, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -D {`*`parallel port`*`} -pci cppdwarf`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, ... com99.

`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are lpt1 and lpt2.
`-pci`	sets the code download option for the PCI bus (optional). When no serial port is specified, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`cppdwarf`	specifies the executable file `cppdwarf`.

6. **Do Not Enter Run!**

   Now you are ready to examine some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes.

   At the (gdb960) prompt, enter:

   `ptype person`

   The following data type information concerning the class `person` appears:

Example 5-1 person Class

```
    type = class person {
      protected:
        char name[40];
        char dept[40];
      public:
        void setName ();
        void setName (char *);
        void setDept ();
        void setDept (char *);
        void printName ();
        virtual int isOutstanding ();
        virtual char * getDept ();
}
```

Please note the following concerning the above output:

- The entire class information for person is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.

Another C++ class in the program is `professor`, which inherits from the person class. Again, you use the gdb960 command `ptype` to display a description of the `professor` class.

7. At the (gdb960) prompt, enter:

   `ptype professor`

   The following data type information concerning the class professor appears:

Example 5-2 professor Class

```
 type = class professor : public person {
   private:
     int numPubs;
   public:
     void setNumPubs ();
     void setNumPubs (int);
     virtual int isOutstanding ();
}
```

Please note the following concerning the above output:

- The entire class information for `professor` is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.
- `type = class professor : public person` indicates that the `professor` class inherits from the `person` class.

8. You are ready to set some breakpoints.
   a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

      `break professor::setNumPubs`

      The following information concerning breakpoints is displayed:

      ```
      [0] cancel
      [1] all
      [2] professor::setNumPubs(int) at
      cppdwarf.cpp:125
      [3] professor::setNumPubs(void) at
      cppdwarf.cpp:118
      ```

      Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.
   b. Set a breakpoint on all `professor::setNumPubs` functions. At the `>` prompt, enter: **`1`**

      The following information about breakpoints is displayed:

      ```
      Breakpoint 1 at 0xa00083d0: file cppdwarf.cpp,
      line 125.
      Breakpoint 2 at 0xa0008358: file cppdwarf.cpp,
      line 118.
      ```
   c. Set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

      `break professor::isOutstanding`

      The following information concerning breakpoints is displayed:

      ```
      Breakpoint 3 at 0xa0009080: file cppdwarf.cpp,
      line 110.
      ```
9. You are now ready to start the program. At the (gdb960) prompt, enter:

   `run`

   Notice that the program stops at all three of the breakpoints.
10. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.

11. At the (gdb960) prompt, enter: **`quit`**

    The results of the debug session were as expected because no optimizations had been performed on the source code during compilation. You can now recompile the `cppdwarf.cpp` program using the highest-level of module-local optimization and repeat the previous debug session.

12. Compile the program using the `-O4` optimization level. In the Command Prompt window, enter the following command:

    ```
    gcc960 -Felf -A{arch}-T{Link-dir} -stdlibcpp -O4 -g
    -o cppdwarf cppdwarf.cpp
    ```

    The options in this command are:

`-Felf`	create an ELF format output file
`-A{`*`arch`*`}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{`*`Link-dir`*`}`	specifies the linker directive file. For example, `-Tmcyhx` specifies `mcyhx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o cppdwarf`	specifies the executable file `cppdwarf`
`cppdwarf.cpp`	input file

13. Run the program using the debugger, enter:

    ```
    gdb960 -t mon960 -b {baudrate} -r {comport} -D
    {parallel port} -pci cppdwarf
    ```

    The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.

`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, ... com99.
`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are: lpt1 and lpt2.
`-pci`	sets the code download option for the PCI bus (optional). When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used.)
`cppdwarf`	specifies the executable file.

14. **Do Not Enter Run!**

    You are now ready to investigate some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes. At the (gdb960) prompt, enter:

    `ptype person`

    Please note, the output matches that of Example 5-1, "person Class". Optimizations did not affect the `person` class output. It is the same as the first debug session.

15. Another C++ class in the program is `professor`, which inherits from the person class. Once again, you use the gdb960 command `ptype` to display a description of the `professor` class. At the (gdb960) prompt, enter:

    `ptype professor`

    Again please note, the output matches that of Example 5-2, "professor Class". Optimizations did not affect the `professor` class output. It is the same as the first debug session.

16. You are now ready to set some breakpoints.
    a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

       `break professor::setNumPubs`

       The following information concerning breakpoints is displayed:

       ```
       [0] cancel
       [1] all
       [2] professor::setNumPubs(int) at
       cppdwarf.cpp:125
       [3] professor::setNumPubs(void) at
       cppdwarf.cpp:118
       ```

       Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 only sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.
    b. Set a breakpoint on all `professor::setNumPubs` functions, so At the `>` prompt, enter: **`1`**.

       The following information about breakpoints is displayed:

       ```
       Breakpoint 1 at 0xa00082e4: file cppdwarf.cpp,
       line 125.
       Breakpoint 2 at 0xa0008294: file cppdwarf.cpp,
       line 118.
       ```
    c. Finally, set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

       `break professor::isOutstanding`

       The following information concerning breakpoints is displayed:

       ```
       Breakpoint 3 at 0xa0008960: file cppdwarf.cpp,
       line 111.
       ```

17. You are now ready to start the program. At the (gdb960) prompt, enter:

    `run`

    Notice that the program does not stop at all three of the breakpoints. As can be seen, the DWARF debug information format is very rich, 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.
18. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.
19. At the (gdb960) prompt, enter: **`quit`**

CONGRATULATIONS! You may now know how to use ELF/DWARF to debug your optimized C++ code.

Writing Flash

NOTE. *In order to write to flash on your Cyclone base board, you need a 12 volt power supply. Also, these instructions are used with the CTOOLS 6.0 and MON960 3.2.3 toolsets.*

This example teaches you the following:

- Writing to flash on the Cyclone base board.
- Booting off of the flash in socket U27 of the Cyclone base board, as opposed to the flash on the CPU Module.
- Setting the Cyclone base board to 12 volts.
- Using *mondb.exe* as a simple utility to download and execute an application program on the target board running MON960.
- Using *mondb.exe* to write flash.
- Building a new monitor for a particular i960 microprocessor family member.
- Retargeting MON960 for other boards.

Complete this step:

1. Choose **MON960**.
2. Choose **Writing Flash.**
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Identify the Flash on the Cyclone base board.

   A blank Flash chip ships on each Cyclone base board in socket U22. To write MON960 to Flash, you must move the blank Flash from socket U22 to socket U27.
5. Set the Cyclone base board voltage to 12 volts.

   Locate the four-position DIP switch labeled S1. Flip S1.1 to the *ON* position. This enables VPP to the Cyclone base board Flash.
6. Power up the Cyclone eval base board.

   Locate the four-pin connector that interfaces to a secondary power supply labeled J6. Three of the connector pins connect to +5 VDC, +12 VDC and ground. (On the PCI-SDK Platform, +12 VDC and +5 VDC power is supplied through the edge connector.)
7. Edit `Version.c`.
   a. Change directories to where the `version.c` file resides. The default installation directory for CTOOLS is:

      `c:\intel960\src\mon960\common`

      If you cannot find the mon960 directory, You need to install MON960 as directed in the *MON960 Debug Monitor User's Manual*.

      Version.c contains the following information:

```
const char mon_version_byte =  nn;   /* version n.n = nn */
const char base_version[] = "MON960 n.n.n";
const char build_date[] = __DATE__;
```

   b. Change the file contents to reflect that this is your version of MON960. For example, change

      `const char base_version[] = "MON960 n.n.n";`

      to:

      `const char base_version[] = "MY MON960";`

   c. Save `Version.c`.

8. Build the new MON960 from source (optional).

   By default the source for MON960 is located at:
   `c:\intel960\src\mon960\common`

   You may use the pre-built version of MON960 there, or build a custom verion. To create a custom version:

   a. Copy `makefile.xxx` to
      `c:\intel960\src\mon960\common\makefile`.

      where xxx is one of the following make files:

`makefile.ic`	(ic960 interface, COFF format)
`makefile.ie`	(ic960 interface, ELF format)
`makefile.gc`	(gcc960 interface, COFF format)
`makefile.ge`	(gcc960 interface, ELF format)

   b. Issue the commands:

      ```
      nmake -s makefile
      cyhx
      ```

      This creates a file called `cyhx.fls`.

9. Write the Flash.

   To write the Flash, use the mondb.exe utility located in the `intel960\bin\` directory. If you are going to use the pre-built MON960 files, they are located in the `intel960\roms` directory. For example, if you used the default installation directory and are using the pre-built MON960 files for the 80960Hx, enter:

   ```
   mondb -ser com1 -par lpt1 -ef -ne
   c:\intel960\roms\cyhx.fls
   ```

   The options in this command are:

`-ser com1`	use serial port 1
`-par lpt1`	use parallel port 1
`-ne`	no execute
`-ef`	erase Flash
`cyhx.fls`	input Flash filename

   Note also that if you built a version of MON960 from the source code as described previously, the `cyhx.fls` file will be located in the `c:\intel960\src\mon960\common\` directory.

10. Set Board Voltage Back To +5 VDC.

    Locate the four-position DIP switch labeled S1. Set S1.1 to the *OFF* position. This disables VPP to Cyclone EP base board Flash and protects the Flash. Note that the PCI80960DP and i960 `Hx` evaluation platforms do not boot when VPP is enabled and MON960 is running from the evaluation board Flash.

11. Set board to boot from U27 socket.

    Locate the four-position DIP switch labeled S1. Set S1.3 ROMSWAP to the *ON* position. This exchanges the addresses of the CPU Module ROM and the base board ROMs. When the switch is *OFF* the processor boots from the CPU Module ROM; when the switch is *ON* the processor boots from the base board ROMs.

12. Reset Base Board.

    Locate the reset pushbutton labeled S2. Use this button to manually reset the Cyclone base board and boot from the base board ROMs.

NOTE. *If you have trouble with this example, refer to Chapter 3 for troubleshooting tips.*

How to Add Benchmarking Routines to Your Code

Benchmarking is a common way to evaluate an architecture for its performance. CTOOLS comes with two routines for benchmarking code. These routines are called `bentime()` and `init_bentime()`. `init_bentime()` is called once to program the on-board Counter/Timer to periodically interrupt the processor. The `bentime()` routine returns the time in microseconds based on the count from the interrupt handler, `timer_isr`, and the current count read from the timer. By placing a call to `bentime()` at the start and end of the code you are timing, the elapsed time can be calculated by the difference between the second call to `bentime()` and the first.

1. Choose **Benchmarking**.
2. Choose **Qv Code**.

3. Scroll through the `chksum.c` code for comments that refer to "Benchmarking Routine". You can add similar lines to the code that you want to time.
4. Choose Make to compile, link, and download the program automatically.
5. Execute the `chksum` program. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 1.566630 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**

Other i960 Processor Choices and the Remote Evaluation Facility

The i960 RISC processor family has a wide breadth of processors to match your design's price and performance needs. If you wish to evaluate other i960 processor family members, contact your local distributor and order different Cyclone CPU modules, or visit the Remote Evaluation Facility at `http://developer.intel.com/design/i960/testcntr`

NOTE. *The i960 Rx Processor is not available through the Remote Evaluation Facility.*

If you choose to order more CPU modules, you may rest assured that all i960 processor modules plug-n-play with your QUICK*val* kit. This configuration was specifically designed to protect your investment and offer a low cost migration path for future needs.

The i960 Jx CPU Example Programs

6

The i960 Jx processor family, nicknamed the Cobra series, is comprised of five products offering a variety of features and performance levels. Using Intel's advanced design and process technologies, the Cobra series makes a variety of choices from operation voltage, cache size and speed doubling available to you.

Additionally, you can optimize your system's performance with CTOOLS, which includes a profile-driven compiler that can automatically optimize your code based on its runtime behavior.

The following pages describe the example programs included with this kit. Each example highlights a feature of the architecture or CTOOLS and provides you with source code that can help shorten your software development cycle. Table 6-1 provides descriptions of the tutorials included in the i960 Jx QUICK*val* kit.

Table 6-1 QUICK*val* i960 Processor Sample Programs

Tutorial Description	Source Files
Hello World: Uses simple printf statement to verify system integrity.	`hello.c`: source file `system.c`: system file
Memory Test: Used for system verification of external memory. The programs perform byte, short, or word writes to external memory, and then they check the addresses written for correctness.	`memtst8.c:` 8 bit memory test `memtst16.c`: 16 bit memory test `memtst32.c`: 32 bit memory test `system.c`: system file

continued ☞

Table 6-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
Data Cache: Uses the minimum edit distance algorithm to demonstrate the effectiveness of the on-chip data cache. This example also shows how to enable and disable the data cache and how to configure an area of memory for caching.	`dcache.c`: source file `system.c`: system file
Instruction Cache: Uses a simple loop to demonstrate how to enable and disable the instruction cache. It also highlights the performance advantage obtained when using the on-chip instruction cache.	`loop.c`: source file `system.c`: system file
Register Cache: Demonstrates using the on-chip register cache in reducing the interrupt latency for high priority interrupts.	`reg_int.c`: source file `low_int.s`: interrupt handler for low priority `high_int.s`: interrupt handler for high priority `system.c`: system file
External Interrupts: Shows how to configure the Cyclone board timers to trigger hardware interrupts. This is also an example of using interrupt handlers and placing the handlers in the interrupt table.	`cyint.c`: source file `asm_fns.s`: interrupt handler for Sx `int_proc.s`: interrupt handler-all processors but Sx `t85c36.c`: eval board timer file `system.c`: system file
Internal Interrupts: Simple timer example showing how to overlay the memory-mapped registers with a structure to program the on-chip timers. This tutorial also shows how to set up interrupt routines using the timers.	`timrcntr.c`: source file `timers.c`: on-chip timer file `system.c`: system file
Halt Mode: This program shows how to make the processor enter halt mode, a power saving state that reduces energy consumption and heat dissipation as it waits to continue code execution. The example uses the on-chip timers to trigger interrupts and “wake up” the processor.	`halt.c`: source file `incremen.s`: interrupt handler `system.c`: system file

continued ☛

Table 6-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
Fault Handling: Shows how to set up the fault handling procedures in the fault and system procedure tables.	`fault.c`: source file `flt_proc.c`: fault procedures `asm_flt.s`: assembly functions to help generate faults `system.c`: system file
C Local Optimizations: Shows how to use the C compiler with high levels of static optimization for improved runtime performance.	`chksum.c`, `system.c`: source files
C Global Optimizations: Shows how to use program-wide optimizations of the C compiler for increased performance.	`chksum.c`, `system.c`: source files
C++ Local Optimizations: Shows how to use the C++ compiler with high levels of static optimization for improved runtime performance.	`optimize.cpp`: source file
C++ Global Optimizations: Shows how to use program-wide optimizations of the C++ compiler for increased performance.	`optimize.cpp`: source file
C++ Virtual Function Optimizations: Shows how a call to a virtual function can be replaced by a direct call to a member function, and, if possible, it may be inlined at the call site. This improves the runtime performance of the code.	`optimize.cpp`: source file
Profiling Lab: Teaches you how to use some of CTOOLS advanced profiling features.	`chksum.c`: source file
Self-Contained Profile: Shows how to create a self-contained profile that captures the program structure and associates it with the program counters from a raw profile. When the source program changes, the global decision making step interpolates or stretches the counters in the self-contained profile to fit the changed program.	`quick.c`: source file

continued ☛

Table 6-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
Incremental Profiling: Shows how to profile a program in pieces and then re-combine them later, a useful methodology when the target execution environment is memory limited	`fault.c`, `flt_proc.c`, `asm_flt.s`, `system.c`: source files
C Cave: Uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.	`ttt.c`: source file
C++ Cave: Shows how to reduce target memory requirements. The text sections of compressed and uncompressed C++ executables are compared. This example also shows how to specify functions for compression.	`cavecpp.cpp`: source file
Linker Directive Language: Provides a hyperlinked manual that describes the linker command options. This tutorial is found in the online help only, not in this manual.	
Linker Consumption: Shows the ability of the linker, gld960, to consume b.out-format, COFF, or ELF object files and libraries in any combination.	`cyint.c`, `int_proc.s`, `t85c36.c`, `system.c`: source files
xlate960 Assembly Language Converter: Shows how to use xlate960 to convert assembly language code written for one i960 processor family member to that of another.	`xlt.s`: source file
i960 Processor Assembler Pseudo-Instruction Support: Shows how to use the new assembler pseudo-ops.	`pseudop.c`: source file
Debugging with gdb960: Uses the Go Fish card game to teach a few useful debugger commands.	`fish.c`: source file `system.c`: system file

continued ☛

Table 6-1 QUICK*val* i960 Processor Sample Programs (continued)

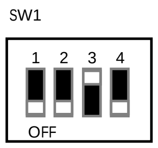
Tutorial Description	Source Files
ELF/DWARF Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to set a breakpoint on an in-line function.	`swap.c`: source file
C++ DWARF-2 Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to debug a C++ application.	`cppdwarf.cpp`: source file
Retargeting MON960: Provides steps for retargeting MON960. This tutorial is found in the online help only, not in this manual.	
Writing Flash: Demonstrates how to update the version of MON960 on your evaluation board.	
80960 Family Benchmark: Shows how to use this facility to compare your processor's performance with other i960 family members. This example uses a typical checksum routine to show how to add benchmarking routines into source code.	`chksum.c`, `system.c`: source files
Remote Evaluation Facility: Guides you through the use of this new benchmarking facility on the World-Wide Web.	

System Validation

Hello World

The program `hello.c` is used to verify your software and hardware system integrity. The following steps provide instructions on how to compile, link, download, and execute this program.

1. Verify that your software and hardware have been installed according to the instructions in Chapter 2 through 3 and the frequency switch on your CPU module is set as shown. The switch settings below set the 80960Jx CPU module frequency at 33 MHz.



1. Power your Cyclone evaluation platform and i960 Jx CPU module.
2. Double-click on the **Hx Jx Cx & Sx QUICK*val*** icon in the QUICK*val* program group.
3. Configure you hardware.
   - Select the **80960 Architecture** tab
   - Select **Jx**.
   - Depending on the board you have installed, select either the EP80960BB or PCI80960DP tab.
   - Configure the software communication options to match those of your evaluation board.
   - Choose **OK**.
4. Choose **Hello World**.
5. Choose **Make** to compile, link, and download the program automatically.
6. Use the gdb960 debugger to execute hello. Type:
   `run`
7. The gdb960 debugger responds by displaying:

```
Hello...Welcome to the 80960JX QUICKval Kit!
SYSTEM CHECK COMPLETED!!
Now you may proceed with our Example Programs.
Program Exit: 01
(gdb960)
```

8. To exit the debugger, type: **`quit`**

CONGRATULATIONS! You have successfully installed your software and your hardware, compiled a program using gcc960, and downloaded and executed the program on your evaluation board using the gdb960 debugger.

If you received any error messages during this process, refer to "If Something Goes Wrong" on page 6-8.

Memory Test

The programs `memtst8.c`, `memtst16.c`, and `memtst32.c` are used to test the external memory on the Cyclone base board.

Depending on the test that is run, an 8, 16, or 32-bit test is run on an area of memory. The program writes F's and 0's to a memory location and reads the location to verify the integrity of what was written. All three programs are almost identical, with the exception of the casting of the variable *ADDR, which allows you to perform different test types.

NOTE. *Below,* `memtst*.c` *refers to either the byte, short, or word memory test example.*

1. Choose **Memory Test**.
2. Choose a memory test. The options are, **8-bit Memory Test**, **16-bit Memory Test**, or **32-bit Memory Test**.
3. Choose **Make** to compile, link, and download the program automatically.
4. Use the gdb960 debugger to execute memtst. Type:

   ```
   run
   ```

5. For the 8-bit test, `memtst8.c`, the gdb960 debugger responds by displaying:

```
This program will run a 8-bit test on the external memory.

Test to be implemented is byte test.
Starting address = a000dfb0
Ending address = a000ec30

Press enter to begin test with 0’s.
Number of errors that occurred is 0.

Begin test for f’s.

Press enter to continue.
Number of errors that occurred is 0.

All tests are complete.
Program exited with code 030.
(gdb960)
```

6. Exit the debugger. Type:

   `quit`

If Something Goes Wrong

The following section describes a few actions that may help resolve errors that may have occurred when invoking one of the tools. If you were unable to get the proper response from the gdb960 debugger after executing the above programs and the trouble-shooting hints described below do not help, contact the 80960 Technical Support Group by phone at 1-800-628-8686 or by E-mail at 960tools@intel.com.

MON960 Debug Monitor is Not Responding...

If the red FAIL LED (CR6) on the base board is lit, the monitor may not have booted up correctly. Press the reset button (S2). If the red FAIL LED remains lit, contact the 80960 Technical Support Group.

Invoking the gcc960 Compiler Resulted in Errors...

The environment must be set-up as described in Chapter 2. If you chose the default directories while installing CTOOLS, verify that the path names `C:\INTEL960\BIN` have been added to your PATH variable and that the following statement is in your `autoexec.bat` file. If you did not install these tools using the default directories, make the appropriate change.

```
SET G960BASE=C:\INTEL960
```

NOTE. *You did not use the default directories on installation, please make sure the G960BASE environment variable is assigned appropriately.*

NOTE. *Don't forget to re-boot your system once you have made any necessary changes to your* `autoexec.bat` *file.*

Invoking the gld960 Linker Resulted in Errors...

Verify that the directory that contains the `hello.c` and `memtst*.c` example programs also now has the object files, `hello.o` and `memtst*.o`. If `hello.o` and `memtst*.o` do not exist, then the gcc960 compiler command did not successfully create an object file. Re-compile `hello.c` and `memtst*.c` to see if an error occurred during the compilation.

If `hello.o` and `memtst*.o` do not exist, make note of the error message and contact the 80960 Technical Support Group.

Invoking the gdb960 debugger resulted in errors...

NOTE. *If you are using the PCI-SDK evaluation platform, you may specify* `-pci` *for PCI download and PCI communication.*
For a list of all the gdb960 command line options, at a command prompt, enter: **`gdb960 -h | more`**

Serial communication error

A serial communication error causes the gdb960 debugger to respond by displaying:

```
HDIL error (10), communication failure
HDIL error (10), communication failure
You can’t do that when your target is ‘exec’
```

Verify that the serial port you are using is the one you specified in the gdb960 command line. Verify that your serial cable is properly connected to the board and to your PC.

Parallel communication error

A parallel communication error causes the gdb960 debugger to respond by displaying:

```
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type ‘show copying’ to see the conditions.
There is absolutely no warranty for GDB; type “show warranty” for
details.
gdb960.exe 6.0, Wed FEB 16 12:33:16 1998
GDB 5.10 (i486-intel-dos --target i960-intel-mon960), Copyright 1997
Free Software Foundation, Inc...(no debugging symbols found)...
Connected to com1 at 115200 bps.
(gdb960)
section 0, name .text, address 0xc0008000, size 0x50ec, flags 0x20
          writing section at 0xc0008000
```

Verify that the parallel port you are using is the one you specified in the gdb960 command line. Verify that your parallel cable is properly connected to the board and to your PC.

Data Cache Tutorial

The i960 Jx processors feature a 2-Kbyte, direct-mapped data cache that is write-through and write-allocate. These processors have a line size of four words. Each line in the cache has a valid bit; to reduce fetch latency on cache misses, each word within a line also has a valid bit.

The purpose of the `dcache.c` program is to show the performance advantage that can be obtained by the use of the data cache on the i960 Jx microprocessor.

This example uses the Minimum Edit Distance (MED) algorithm in order to show the effectiveness of using the data cache. The MED algorithm finds the minimum number of edit steps required to change one string into another.

This example is a real world example of using the data cache. This algorithm maintains a cost matrix to determine which change to the string being edited would incur the least cost. The cost matrix is a 2-D array [1..n][1..m], where n and m are the sizes of the two strings.

The algorithm really shows the speed of the data cache due to three reads for each write to the cost matrix. The algorithm reads from the cache to determine which step to take next, then writes its choice in the cost matrix. Since the writes to the data cache are write-through, there is no improvement for writes to the data cache. The Write-Through feature maintains coherency between the data cache and external memory.

The source code includes system files, `system.c` and `system.h`, that includes a macro and an assembly function that simplifies issuing data cache control instructions.

Also, the example shows how to define an area of memory to make data cacheable by using the Logical Memory Configuration (LMCON) registers. The address of the area to make cacheable is programmed into the Logical Memory Address Register (LMADR). The mask is programmed into the Logical Memory Mask Register (LMMR).

1. Choose **Cache Examples**.
2. Choose **Data Cache**.
3. Choose **Qv Code**.

4. Scroll through the `dcache.c` code to see the calls to the macro, `dcctl_contrl`.
5. Open and scroll through the `system.h` and `system.c` code to see the macro and assembly function, `dcctl_control` and `i960_dcctl`.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `dcache`. Type:

   `run`

   The debugger responds by displaying:

```
Minimum Edit Distance algorithm makes reads from the data cache.
This routine will determine how many steps are needed to convert:
StringA:  80960 QUICKval EvalKit
TO StringB:  i960(R) HxJxCxSx & Kx
Starting timed routine with data cache on ...
RESULT: 18 moves are required to convert string A to string B
Elapsed Time On = 0.004048 seconds
Elapsed Time for routine with data cache off ...
RESULT: 18 moves are required to convert string A to string B.
Elapsed Time Off = 0.004581 seconds
IMPROVEMENT: 11.6 percent
(gdb960)
```

8. Type: `quit`
9. Select **Results**.

NOTE. *Your actual run times may vary.*

Instruction Cache Tutorial

The i960 Jx processor comes equipped with 4 KB of two-way set-associative instruction cache. The instruction cache boosts your application's performance by reducing the number of instruction fetches from external memory. The cache provides fast execution of code and loops of code in the cache.

The `loop.c` program demonstrates the performance boost obtained by running a loop completely within versus outside of the instruction cache.

The source code includes system files, `system.c` and `system.h` that includes a macro and an assembly function that simplifies issuing instruction cache control instructions.

1. Choose **Cache Examples**.
2. Choose **Instruction Cache**.
3. Choose **Qv Code.**
4. Scroll through the `loop.c` code to see the calls to the macro, `icache_control`.
5. Open and scroll through the `system.h` and `system.c` code to see the macro and assembly function, `icache_control` and `i960_icctl`.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `loop`. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/loop
   Simple loop timed with instruction cache off ...
   Elapsed Time Off = 2.259 seconds
   Simple loop timed with instruction cache on ...
   Elapsed Time On = 0.798 seconds
   IMPROVEMENT : 64.7 percent
   Program exited with code 01
   (gdb960)
   ```
8. Type: **`quit`**
9. Select **Results**.

Register Cache

The i960 Jx processor provides fast storage of local registers for call and return operations by using an internal local register cache. Up to eight local register sets can be contained in the cache before sets must be saved in external memory. The default cache size is five register sets. The register set is all the registers (i.e., r0 through r15). The processor uses a 128-bit wide bus to store local register sets quickly to the register cache.

To decrease interrupt latency, software can reserve a number of frames in the local register cache solely for high priority interrupts (interrupted state and process priority greater than or equal to 28). When a frame is reserved for high-priority interrupts, the local registers of the code interrupted by a high-priority interrupt can be saved to the local register cache without causing a frame flush to memory.

This program demonstrates the use of the on-chip register cache in reducing the interrupt latency for high priority interrupts. First, high priority interrupts are timed using the register cache, then low priority interrupts are timed without the use of the register cache.

1. Choose **Cache Examples.**
2. Choose **Register Cache**.
3. Choose **Qv Code**.
4. Scroll through the `reg_int.c` code and find the `PRCB_Ptr -> reg_cache_config` assignment. That is where the Register Cache Configuration Word in the Processor Control Block gets written. It is assigned to allocate all 8 frames for high priority interrupts.
5. Open and scroll through the `high_int.s` and `low_int.s` files. The `high_int.s` file contains the interrupt handling procedure for the high priority interrupts, and the file `low_int.s` contains the interrupt handling procedure for the low priority interrupts.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `reg_int`. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\quickval/reg_int
   Triggering Interrupts ... Register Cache USED ...
   RESULT: Timeon is 0.000113 seconds
   Triggering Interrupts ... Register Cache NOT USED ...
   RESULT: Timeoff is 0.000115 seconds
   IMPROVEMENT: 1.7 percent
   Program exited with code 01.
   (gdb960)
   ```
8. Exit the debugger, type: `quit`
9. Select **Results**.

External Interrupts Tutorial

The purpose of this program, `cyint.c`, is to show the steps required when dealing with an interrupt triggered externally by the evaluation board timers. The `cyint.c` source code contains step-by-step instructions to save you time when you program interrupts for your application. `int_proc.s` is the interrupt handler, and `t85c36.c` contains the functions to program the evaluation board timers.

The example performs the following steps in the handling of a hardware interrupt.

- Modify the ICON register
- Modify the IMAP register
- Cache the interrupt vector and the interrupt handling procedure
- Lower the processor priority
- Modify the IMSK register
- Clear the IPND register
- Generate the hardware interrupt using the evaluation board timers

Complete these steps:

1. Choose **Interrupt Examps**.
2. Choose **External Interrupts**.
3. Choose **Qv Code**.
4. Scroll through the `cyint.c` source to see the code for setting up and handling a hardware interrupt triggered by the evaluation board timers.
5. Open and scroll through the `t85c36.c` and `t85c36.h` files to see the definitions and routines for programming the evaluation board timers. You can simplify the programming of the evaluation board timers by including this code in your own applications.
6. Choose **Make** to compile, link, and download the program automatically.

7. Use the gdb960 debugger to execute `cyint`. Type:

   `run`

   The debugger responds by displaying:

```
   interrupt count = 83
   interrupt count = 96
   interrupt count = 108
   interrupt count = 121
   interrupt count = 133
   interrupt count = 145
   interrupt count = 157
   interrupt count = 170
   interrupt count = 182
   Program exited with code 020.
(gdb960)
```

8. Type: **`quit`**

NOTE. *Your actual interrupt counts may vary.*

Internal Interrupts Tutorial

A key feature of the i960 Jx processor are the dual, fully independent 32-bit timer units. Each is programmed by use of the timer registers. These registers are memory-mapped within the processor, addressable on 32-bit boundaries. The timers have a single shot mode and auto-reload capabilities for continuous operation. Each timer has an independent interrupt request to the processor's interrupt controller. Each timer can generate a fault when it detects unauthorized writes from user mode.

The `timrcntr.c` program demonstrates how to use the structures and routines found in `timers.c` to easily program either timer to cause periodic interrupts.

1. Choose **Interrupt Examps**.
2. Choose **Internal Interrupts**.
3. Choose **Qv Code**.

4. Scroll through the `timrcntr.c` code to see the code for setting up a timer to cause a hardware interrupt.
5. Open and scroll through `timers.c` and `timers.h` files to see the definitions and routines for programming the on-chip timers. You can simplify the programming of the timer by including this code in your own applications.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `timrcntr`. Type:

   `run`

   The debugger responds by displaying the current count of each timer every time timer0 causes an interrupt.
8. Type: **`quit`**

Halt Mode

Another key enhancement of the i960 Jx processor — not available on any other i960 processor family members — is the `halt` CPU instruction.

Entry into HALT mode by the `halt` instruction causes the following actions to occur:

- Interrupts are enabled or disabled based on the value of the `src1` argument supplied in the `halt` instruction.
- The processor ensures that all previous load and store operations have completed before continuing. If the bus queues are not empty, the processor asserts the BSTAT pin and waits for the bus queues to empty.
- The processor attempts to reduce power consumption to more efficiently wait for the exit from HALT mode.

The i960 Jx processor's power needs drop by approximately an order of magnitude while in HALT mode. Code execution stops but the processor maintains its internal state and can still respond to certain internal and external events.

The internal timers, when enabled, continue to decrement each cycle during HALT mode and can even force the processor out of HALT mode if either timer generates an interrupt of sufficient priority.

The processor responds normally to external events such as interrupt requests, hardware RESET, and HOLD requests.

1. Choose **Halt Mode**.
2. Choose **QV Code**.
3. Scroll through the `halt.c` code to see a call to the function Halt_Mode (ENABLE).
4. Open and scroll through `system.c` and `system.h` files to see the definitions and routines for programming HALT Mode. You can simplify the programming of the HALT Mode by including this code into your own applications.
5. Choose **Make** to compile, link, and download the program automatically.
6. Use the gdb960 debugger to execute `halt`. Type:

   `run`

   The debugger responds by displaying:

   ```
   HALT MODE
   POWER SAVINGS
   I WILL NOW ENTER HALT MODE !!!
   I will not execute any instructions, and I will save
   power !!!
   I will wait for an interrupt in 268,435,455 bus
   cycles to wake me up !!!

   Look at the CR5 LED on the Cyclone board to see when
   I continue running!!!
   Program exited with code 01.
   (gdb960)
   ```

The `halt.c` program is controlled by a while loop. At the head of the loop, a message concerning HALT Mode is printed. The 32 bit on-chip timers are then configured to count down and trigger an interrupt. Lastly, the HALT Mode instruction is executed. When the timer counts down, it triggers an interrupt which "wakes" up the processor. The interrupt handler increments a variable which controls the while loop. The loop will be executed 3 times.

It is important to notice that the CR5 LED (Run LED) goes out while the processor is in HALT Mode, and when the interrupt is triggered and the processor “wakes” up, the CR5 LED lights because it starts executing code.

7. Exit the debugger. Type: **`quit`**

Fault Handling

These programs, `fault.c`, `flt_proc.c`, `asm_flt.s`, and `system.c`, show the steps taken in setting up the fault handling procedures in the fault and system procedure tables. The faults are then triggered one by one.

Table 6-2 i960 Jx Processor Fault Types and Subtypes

Fault Type		Fault Subtype		Fault Record
Number	**Name**	**Number or Bit Position**	**Name**	
0H	OVERRIDE	NA	NA	See your microprocessor user's manual
0H	PARALLEL	NA		See your microprocessor user's manual
1H	TRACE	Bit 1	INSTRUCTION	0001 0002H
		Bit 2	BRANCH	0001 0004H
		Bit 3	CALL	0001 0008H
		Bit 4	RETURN	0001 0010H
		Bit 5	PRERETURN	0001 0020H
		Bit 6	SUPERVISOR	0001 0040H
		Bit 7	MARK	0001 0080H
2H	OPERATION	1H	INVALID_OPCODE	0002 0001H
		2H	UNIMPLEMENTED	0002 0002H
		3H	UNALIGNED	0002 0003H
		4H	INVALID_OPERAND	0002 0004H
3H	ARITHMETIC	1H	INTEGER_OVERFLOW	0003 0001H
		2H	ZERO-DIVIDE	0003 0002H
4H	Reserved			
5H	CONSTRAINT	1H	RANGE	0005 0001H
6H	Reserved			
7H	PROTECTION	Bit 1	LENGTH	0007 0002H
9H	Reserved			
AH	TYPE	1H	MISMATCH	000A 0001H
BH - FH	Reserved			

1. Choose **Fault Handling**.
2. Choose **QV Code**.
3. Scroll through the `fault.c` code to see a call to the function load_flt_proc(). This function loads the fault handling procedures into the fault and/or the system table.
4. Open and scroll through the `flt_proc.c` and `asm_flt.s` files. The `flt_proc.c` file contains the fault handling procedures, and the file `asm_flt.s` is used to help generate the faults.
5. Choose **Make** to compile, link, and download the program automatically.

NOTE. *When compiling, disregard the compiler warning:*

```
Warning: unaligned register
```

This is one of the faults that will be handled.

6. Use the gdb960 debugger to execute `fault`. Type:

   `run`

   The debugger responds by displaying the Fault Type and the Fault Subtype for each fault handled. The address of the faulting instruction is given (see Table 6-2).
7. Type: **`quit`**

Static, Global, and Profile-Driven Optimizations

Optimizing compilers provide you with a means of developing high performance code without detailed knowledge of the architecture. Engineers who understand the features of the i960 architecture developed gcc960 to provide optimizations that take full advantage of the i960 processor. In general, optimizing compilation takes more time and may require more memory for large functions. However, the benefit in runtime performance is well worth it.

There are several levels of optimization available. Typically, low levels of optimizations are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once your application is functioning properly, you can increase its runtime performance by using a higher level of optimization.

Release 5.0 and later of the development tools support the ELF object module format and DWARF version 2.0 debug information format. The new format enables more accurate mapping between source and object code at higher optimization levels and ease debugging of production code.

The C optimization example uses a program called `chksum.c`. The C++ examples use a program called `optimize.cpp`

C No Optimization

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.
4. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 16.343843 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**

C Static Optimization

Use the following commands to compile the `chksum.c` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.

4. Choose **Make -O4** to compile with optimizations, link, and download the program automatically.
5. To execute the program, type:

   ```
   run
   ```

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 3.908083 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ No Optimization

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.
5. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   ```
   run
   ```

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 10.9502 seconds.
   Program exited normally
   ```
7. Type: **`quit`**

C++ Static Optimization

Use the following commands to compile the `optimize.cpp` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.

3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.
5. Choose **Make -O4** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 8.27324 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

C Global Optimization

Use the following commands to compile the `chksum.c` with program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C Global Optimizations**.
4. Choose **Make +O5** to compile with optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 1.871137 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ Global Optimization

Use the following commands to compile the `optimize.cpp` program using the program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Global Optimizations**.
5. Choose **Make+05** to compile with optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 8.206485 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation

A 89% improvement in C code performance is significant, but there is another level of optimization that is uniquely available through Intel's CTOOLS compilers: profile-driven optimization. This two-pass compilation procedure allows the compiler to make optimizations based on runtime behavior as well as the static information used by conventional 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. Standard (i.e., non-program-wide) optimizations are referred to as 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 before the end of the linking phase.

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.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Profiling Lab**.
4. Follow the **Profiling Tutorial** link in the online help.

Using profile-driven optimization, an increase in runtime performance of 52% is obtained. The average 80960 application can expect to gain 15 to 30% performance improvement through the use of this technology. This boost in performance is available to you without any further investment in hardware.

C++ Virtual Function Optimizations

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, however, 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.

Use the following commands to compile the `optimize.cpp` program.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.

3. Choose **C++ Optimizations**.
4. Choose **C++ Virtual Opts**.
5. Choose **Make -NoVOpt** to compile without virtual function optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 8.20649 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Make -VOpt** to compile with virtual function optimizations, link, and download the program automatically.
9. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 7.20885 seconds.
   Program exited normally
   ```
10. Type: **`quit`**
11. Choose **Results**.

The virtual function optimizations yielded a 12% improvement.

Note the runtime performance at each optimization level as shown below.

Table 6-3 i960 Processor Optimization Results

Optimization Level	C Execution Time	C++ Execution Time
no optimization (-O0)	16.343843 seconds	10.9502 seconds
maximum static (-O4)	3.908083 seconds	8.27324 seconds
global optimization	1.871137 seconds	8.206485 seconds
profile-driven	1.871134seconds	NA
Virtual Function Optimization	NA	7.20885 seconds

Building Self-contained Profiles with gmpf960

A *raw* profile contains program counters that record 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 has a very short useful life. When changes are made in the 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 used to optimize a program 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.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Self-Contained**.
4. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

5. Specify the program database directory.

   The PDB can be specified by setting the environment variable `G960PDB`.

   For example, if you chose the default directory during installation, enter:

   ```
   SET G960PDB=C:\quickval\prof_lab\lab_pdb
   ```

   Or, specify the PDB at compiler invocation time with the `Z`*`dir`* option, as shown in the example below.

   ```
   gcc960  -Zmypdb  foo.o
   ```

6. Compile for profile instrumentation.

   Insert profile instrumentation into *quick* so that when the linked program is executed, a profile can be collected. Type:

   ```
   gcc960 -Fcoff  -T{Link-dir} -A{arch} -fdb
   -gcdm,subst=:*+fprof -o quick quick.c
   ```

   The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-gcdm,subst=:*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only eligible modules not linked in from libraries.
`+fprof`	causes generation of profile instrumentation.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

7. Collect 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. Type:

   ```
   gdb960 -t mon960 -b 115200 -r com1 -D lpt1 quick
   ```

   The options in this gdb960 compiler command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`quick`	the executable file

8. Use the gdb960 debugger to execute `quick`. Enter:

   `run`

9. Exit the debugger. Enter:

   `quit`

10. Enter the command:

    `gmpf960 -spf quick.pf default.pf`

    The options in this gmpf960 compiler command are:

`-spf`	causes a self-contained profile, `quick.pf`, to be produced as output.
`default.pf`	The input profile.

11. Recompile the `quick.c` source code using the profiling information obtained by the instrumentation. Type:

    ```
    gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
    -gcdm,iprof=quick.pf -o quick quick.c
    ```

    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.

`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=quick.pf`	This supplies a profile file `quick.pf` to the global decision making and optimization step.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

12. Change the control structure of `quick.c.`

    Edit `quick.c`. Find the procedure called QUICK. In this procedure, there is a control structure:

```
for(i = 2; i <= SORTELEMENTS; i+=1)
{
    (LOGIC)
}
```

Change the control structure to:

```
i = 2;
while (i <= SORTELEMENTS)
{
    (LOGIC)
    i+=1;
}
```

13. Compile the new `quick.c` using the interpolated profile. Type:

    `gcc960 -Fcoff -T`*`{Link-dir}`* `-A`*`{arch}`* `-fdb`
    `-gcdm,iprof=quick.pf -o quick quick.c`

    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.

`-Gcdm,iprof=quick.pf`	
	This supplies a profile file `quick.pf` to the global decision making and optimization step.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

Notice that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`.

NOTE. *The beauty of this example is that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`, not the results of running this example.*

Profiling a Program in Pieces

Suppose that the target execution environment is memory limited so that all your programs cannot be instrumented for profiling at the same time. You can use substitutions to make partially instrumented versions of the final executable, and then create self-contained profiles for each piece. Each executable created in this way has a limited set of instrumented modules.

After you've created the self-contained profiles, you can use gmpf960 to create a single merged self-contained profile. The final, merged self-contained profile is identical to a profile obtained by instrumenting the entire program at once.

In this example, you use the fault handling example programs to show incremental profiling.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations.**
3. Choose **Incremental**.
4. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

5. Specify the program database directory.

   You can specify the PDB by setting the environment variable `G960PDB`. For example, if you chose the default directory during installation, enter:

   ```
   SET G960PDB=C:\quickval\prof_lab\lab_pdb
   ```

   Or, specify the PDB at compiler invocation time with the `Zdir` option, as shown in the example below.

   ```
   gcc960  -Zmypdb  foo.o
   ```

6. Insert profile instrumentation into `fault` so that when the linked program is executed, a profile can be collected. The instrumented modules in this version of `fault` are from the files `fault.c` and `flt_proc.c`. Type:

   ```
   gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
   -gcdm,subst=:f*+fprof -o fault fault.c flt_proc.c
   asm_flt.s system.c
   ```

`-Fcoff`	creates a COFF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-fdb`	all modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-gcdm,subst=:f*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only the files that begin with *f*.
`+Fprof`	causes generation of profile instrumentation.
`-o fault`	names the executable file `fault`.
`fault.c`	the source files.
`flt_proc.c`	the fault procedures.

`asm_flt.s`	the assembly file to generate faults.
`system.c`	system file.

7. Collect the profile.

   When 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. Type:

   ```
   gdb960 -t mon960 -b 9600 -r com1 -D lpt1 fault
   ```

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`fault`	the executable file

8. Use the gdb960 debugger to execute `fault`. Enter:

   ```
   run
   ```

9. Exit the debugger. Enter:

   ```
   quit
   ```

10. Build the self-contained profiles with gmpf960.

    To create a self-contained profile, use the gmpf960 profile merger tool. gmpf960 is invoked with the raw profile as an input file. Enter:

    ```
    gmpf960 -spf prof1.pf default.pf
    ```

`-spf`	causes a self-contained profile, `prof1.pf`, to be produced as output.
`default.pf`	The input profile.

    The resultant self-contained profile, `prof1.pf`, has a limited set of instrumented modules.

11. Insert profile instrumentation into `fault` so that when the linked program is executed, a profile can be collected. The instrumented modules in this version of `fault` are from the file `system.c`. Type:

    `gcc960 -Fcoff -T`*`{Link-dir}`* `-A`*`{arch}`* `-fdb -gcdm,subst=:s*+fprof -o fault fault.c flt_proc.c asm_flt.s system.c`

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,subst=:s*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only the files that begin with `s`.
`+fprof`	causes generation of profile instrumentation.
`-o fault`	names the executable file `fault`.
`fault.c`	the source files
`flt_proc.c`	the fault procedures
`asm_flt.s`	the assembly file to generate faults
`system.c`	system file

12. Collect the 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. Type:

    `gdb960 -t mon960 -b 9600 -r com1 -D lpt1 fault`

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`fault`	the executable file

13. Use the gdb960 debugger to execute `fault`. Enter:

    `run`

14. Exit the debugger. Enter:

    `quit`

15. Build the self-contained profiles with gmpf960.

    To create a self-contained profile, use the gmpf960 profile merger tool. gmpf960 is invoked with the raw profile as an input file. Enter:

    `gmpf960 -spf prof2.pf default.pf`

`-spf`	causes a self-contained profile, `prof2.pf`, to be produced as output.
`default.pf`	the input profile.

    The resultant self-contained profile, `prof2.pf`, has a limited set of instrumented modules.

16. Merge all the self-contained profiles into one.

    The final `prof.pf` profile is identical to a profile obtained by instrumenting the entire program at once. Type:

    `gmpf960 -spf prog.pf prof1.pf prof2.pf`

`-spf`	causes a self-contained profile, `prog.pf`, to be produced as output.
`prof1.pf`	an input self-contained profile.
`prof2.pf`	an input self-contained profile.

17. Recompile the fault handling source code using the profiling information obtained by the instrumentations. Type:

```
gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
-gcdm,iprof=prog.pf -o fault fault.c flt_proc.c
asm_flt.s system.c
```

`-Fcoff`	create a COFF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-fdb`	all modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=prog.pf`	This supplies a profile file `prog.pf` to the global decision making and optimization step.
`-o fault`	`names the executable file fault.`
`fault.c`	the source file.
`flt_proc.c`	the fault procedures.
`asm_flt.s`	the assembly file to generate faults.
`system.c`	system file.

NOTE. *The beauty of this example is the methodology of incremental profiling, not the result of running the example.*

Compression Assisted Virtual Execution (CAVE)

This CTOOLS feature allows non-critical parts of an application's machine code to be stored in memory in compressed form resulting in reduced target memory requirements. The code is expanded into native machine code on demand for 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 of 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, 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, that 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.

Note that due to the overhead of decompressing code at runtime, only non-performance critical code should be secondary functions, such as error handling code or initialization code. You can use runtime profile information generated by gcov960 to aid in selecting the set of secondary functions.

This example uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.

For the sake of demonstration, we compress performance-critical code in the tic-tac-toe program. The purpose of this example is to show the reduced text section of the executable, not demonstrate run times.

C Example

1. Choose **Compiler**.
2. Choose **C Cave**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

4. Use the gcc960 `mcave` option or `#pragma cave` to designate the specified functions as secondary. In the tic-tac-toe example, `ttt.c`, the following `#pragma` has been added:

   ```
   #pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)
   ```

   where `Initialize, Winner, Other, Play, Evaluate, Best_Move, Describe, Move,` and `Game` are all functions to be compressed.
5. Edit `ttt.c`. Make sure the `#pragma cave` program line is commented out:

   ```
   /*#pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)*/
   ```
6. Compile the tic-tac-toe program. Enter:

```
gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c
```

   The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies mcyjx.gld.
`-o ttt`	names the executable file ttt
`ttt.c`	input file

7. Check the text section size of the uncompressed program. Enter:

   ```
   gsize960 ttt
   ```

   The option in this command is:

`ttt`	name of the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.
8. Edit `ttt.c`. Make sure the `#pragma cave` program line is uncommented:

   ```
   #pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)
   ```

9. Compile the tic-tac-toe program with the pragma program line. Enter:
   `gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c`
   The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies mcy`jx`.gld.
`-o ttt`	names the executable file ttt
`ttt.c`	input file

10. Check the text section size of the compressed program. Enter:
    `gsize960 ttt`
    The option in this command is:

`ttt`	executable file

The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 6-4 Uncompressed Text Sections

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	33,764	32,944	32,768	32,976	31,600

Table 6-5 After Function Compression

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	31,908	30,832	30,816	30,832	29,648
Cave Section	1,818	1,770	1,746	1,800	1,776
Total	33,726	32,602	32,562	32,632	31,424

Table 6-6 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
0.1%	1.0 %	0.6 %	1.0 %	0.6 %

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

C++ Compression Assisted Virtual Execution (CAVE)

1. Choose **Compiler**.
2. Choose **C++ Cave**.
3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Use the *gcc960* `mcave` option or `#pragma cave` designate the specified functions as secondary. In the C++ example, `cavecpp.cpp`, the following `#pragma` has been added:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

   where `initSetName`, `initSetDept`, `initSetGpa`, `initSetNumPubs`, `isOutstanding`, `printName`, and `InitializeRecords` are all functions to be compressed, i.e., all functions are secondary functions. All other functions of the program are primary functions.

   The primary set should contain performance-critical functions that are not to be affected by the CAVE mechanism; the secondary set is subject to compression. Secondary functions are compressed by the linker and reside in memory in compressed form.

   The C++ compiler behaves in essentially the same manner as the C compiler when the mcave or Gcave options are used - generating all functions in the compilation unit for which this option is in effect as secondary.

A user typically designates a single function as secondary through the use of `pragma cave`. The following statement for example designates the function max as secondary.

```
# pragma cave max
```

However in C++ overloaded functions have the same name. Member functions of two different classes are also allowed to have the same name and these member functions can in turn have the same name as a function with file scope.

When a user specifies a function as secondary through the use of `pragma cave`, the C++ compiler treats all functions with this name as secondary. To illustrate, consider the following example:

```
# ifdef PRAGMA
# pragma cave max
# endif

int max(int a, int b)
{
return a > b ? a : b;
}

float max(float a, float b)
{
return a > b ? a : b;
}

class Tclass1 {
int a, b;
public:
int max();
};

int Tclass1::max()
{
return a > b ? a : b;
}
```

```
class Tclass2 {
float a, b;
public:
float max();
};

float Tclass2::max()
{
return a > b ? a : b;
}

Tclass1 t1;
Tclass2 t2;
```

The Compiler treats all the following functions as secondary.

```
int max(int, int);
float max(float, float);
int Tclass1::max();
float Tclass2::max();
```

5. Choose **Qv Cod**e. Edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is commented out:

   ```
   //#pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

6. Compile the C++ program. Enter:

   `gcc960 -A{`*`arch`*`} -Felf -T{`*`Link-dir`*`} -stdlibcpp -o cavecpp cavecpp.cpp`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A{`*`arch`*`}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{`*`Link-dir`*`}`	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `cavecpp`
`cavecpp.cpp`	input file

7. Check the text section size of the uncompressed program. Enter:

   ```
   gsize960 cavecpp
   ```

   The option in this command is:

`cavecpp`	specifies the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.

8. Choose **Qv Code** and edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is uncommented:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

9. Compile the C++ program with the pragma program line. Enter:

   ```
   gcc960 -A{arch} -Felf -T{Link-dir} -stdlibcpp
   -o cavecpp cavecpp.cpp
   ```

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, -AHD specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `ttt`
`cavecpp.cpp`	specifies the input file

10. Check the text section size of the compressed program. Enter:

    ```
    gsize960 cavecpp
    ```

    The option in this command is:

`cavecpp`	executable file

The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 6-7 Uncompressed Text Sections

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	89,788	84,196	83,512	84,196	81,764

Table 6-8 After Function Compression

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	87,612	81,892	81,512	81,892	79,796
Cave Section	1,920	1,546	1,514	1,546	1,512
Total	89,532	83,438	83,026	83,438	81,308

Table 6-9 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
1%	1%	1%	1%	1%

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

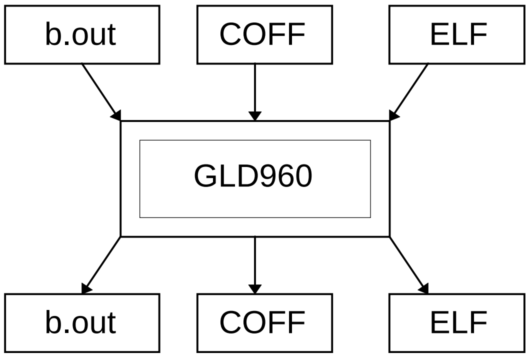
Linker Consumption

You can link b.out-format, COFF or ELF object files and libraries in any combination. To determine a file format, the linker examines the first two bytes of the file. An unrecognized value indicates a linker-directive file. This feature is useful when using third-party archives with CTOOLS runtime libraries and your application code. The runtime libraries are shipped in ELF format *only* (effective with the 5.0 version of the tools). Each can potentially have a different OMF, and the linkage still completes.

NOTE. *As of version 5.0 of the tools, all runtime libraries are shipped in ELF format only.*

If the linker generates a different output format than the input, the linker does not copy debug information from the input file to the output file. Because of this, you should use only one OMF.

The symbol tables of each OMF are abbreviated when crossing OMF boundaries. For example, when you include a b.out OMF file in a linkage where the output file OMF is COFF format, none of the debug information from the b.out file is copied into the output COFF file.



1. Choose **Linker and Utilities**.
2. Choose **Linker Consumption**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

4. Compile the first file in COFF format. Enter:

   `gcc960 -Fcoff -A`*`{arch}`*` -c t85c36.c`

   The options in this command are:

`-Fcoff`	creates a COFF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-c`	compile, but do not link.
`t85c36.c`	input file.

5. Compile the second file in ELF format. Enter:

   `gcc960 -Felf -A`*`{arch}`*` -c system.c`

   The options in this command are:

`-Felf`	creates an ELF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-c`	compiles, but does not link.
`system.c`	input file.

6. Compile the third file in b.out format. Enter:

   `gcc960 -Fbout -A`*`{arch}`*` -c -r cyint.c int_proc.s`

   The options in this command are:

`-Fbout`	creates a b.out format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-c`	compiles, but does not link.
`-r`	allows unresolved references.
`cyint.c`	the source file.
`int_proc.s`	the interrupt handler.

7. Generate an absolute file in ELF format by linking files in b.out-format, ELF format, and COFF format. The absolute file could have also been in b.out-format or COFF format. Enter:

```
gld960 -Felf -T{Link-dir} -A{arch} -o elf t85c36.o
system.o cyint.o int_proc.o
```

The options in this command are:

`-Felf`	specifies the absolute file as ELF format.
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tcyjx` specifies `cyjx.gld`.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-o elf`	names the executable file `elf`.
`cyint.o`	file in b.out-format.
`int_proc.o`	file in b.out-format.
`t85c36.o`	file in COFF format.
`system.o`	file in ELF format.

NOTE. *The beauty of this example is the functionality of the linker, not the result of running the example.*

XLATE960 Tutorial

This tutorial shows how to use the xlate960 utility provided with CTOOLS release 6.0. xlate960 is the 80960 translation utility that generates i960 Rx-compatible code sequences to replace instructions and addressing modes that are only available on other i960 processors.

1. If you are using the **Hx Jx Cx & Sx QUICK*val*** software, Choose **Linker** and **Utilities**. If using the **Rx QUICK*val*** software, this step is not necessary.
2. Choose **xlate960 Tutorial**.
3. Choose **Qv Code**.

The assembly file, `xlt.s`, is loaded into the editor shown on your screen. This program is a contrived example that really does not do any useful work. It was written to help demonstrate how to migrate assembly code to the Rx Strategy. This program supports i960 processor functionality that is not available when using the Rx Strategy. `xlt.s` has two complex addressing modes:

- indexed
- ip-relative

and three classes of instructions

- arithmetic (scanbit)
- triple word /quad word instructions (quad word move)
- integer/overflow behavior (addi)

that demonstrate behavior not supported under the Rx Strategy.

If `xlt.s` were compiled with the `-AJF` architecture option, there would be no compilation errors. However, if `xlt.s` were compiled with the `-ARD` or `-ARP` architecture options, compilation errors would stop the build. The offending instructions and addressing modes would have to be translated to Rx Strategy compatible instructions and addressing modes. xlate960 can do this automatically for you, with only a little user interaction.

Looking at the xlt.s File

To understand what the `xlt.s` file is doing, please review the `xlt.s` file in an editor. The lines that violate the Rx Strategy are detailed below:

Line 83: `bx 24(ip)`

IP-relative addressing is not available when specifying an i960 Rx processor-based target.

The xlate960 utility replaces the above `bx 24(ip)` operation with the following instruction sequence that duplicates the functionality of the `bx 24(ip)` operation:

```
#xlate-beginbx 24(ip)
#xlate-err“Fill in register for E0”
#xlate-warn“Verify use of local labels ‘8’ and ‘9’”
#xlate-err“Verify that register g14 can be clobbered”
bal .+4
8:  lda        24+9f-8b(g14),E0; 9:
bx  (E0)
#xlate-end
```

The line beginning with `#xlate-begin` marks the start of the code added by the xlate960 utility to replace the `bx 24(ip)` instruction, and the line beginning with `#xlate-end` marks the end of the code. All translation errors are marked with a comment of the form `#xlate-err`. More subtle translation incompatibilities are flagged with a `#xlate-warn` comment.

Above, three non-comment lines were added to replace the `bx 24(ip)` instruction. However, based on the suggestions of the comments, these lines may require manual editing. Manual translation is demonstrated later in the tutorial.

Line 126: `st r9,_VariableArray[r11*8]`

Indexed addressing modes are not available when specifying an i960 Rx processor-based target. The xlate960 utility replaces the above `st r9,_VariableArray[r11*8]` operation with the following instruction sequence that duplicates the functionality of the `st r9,_VariableArray[r11*8]` operation:

```
#xlate-beginst r9,_VariableArray[r11*8]
#xlate-err“Fill in register for E1”
shlo3,rll,E1
st  r9,_VariableArray(E1)
#xlate-end
```

Two instructions were inserted by the xlate960 utility to replace the `st r9,_VariableArray[r11*8]` operation. Also, as before, it may be necessary to edit these two instructions to complete code migration.

Line 160: `scanbit r9,r8`

The `scanbit` instruction is not guaranteed to set the condition code with the Rx Strategy.

Line 208: `addi r10,r11,r8`

The `addi` instruction is not supported with the i960 Rx architectures.

Line 239: `movq r8,g8`

The `movq` instruction is not supported with the i960 Rx architectures. The instruction sequence inserted by xlate960 to replace the `movq` instruction does not test for unaligned or overlapping registers. It is left to the programmer to ensure that the registers used do not overlap and that the registers are aligned. The programmer can do this by making sure the code is compatible with existing i960 processors before running the code through xlate960. The programmer should not experience unaligned or overlapping registers if the code has been assembled for another processor prior to running it through xlate960.

Using xlate960

To prove that `xlt.s` compiles unaltered as code designed for earlier i960 processors, complete the following steps:

1. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
2. Enter the following command in the Command Prompt window provided:

   ```
   gcc960 -AJF -Fcoff -Tmcyjx -o xlt xlt.s
   ```

   The options in this command are:

`-AJF`	sets the target architecture for the compiler.
`-Fcoff`	sets the object file type as COFF.

`-Tmcyjx`	uses the linker directive file for the Jx architecture.
`-o xlt`	sets the object file name as `xlt` (optional).
`xlt.s`	specifies the input source file.

3. To run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci xlt`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`xlt`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out:

   - the value of register r11 before and after the ip-relative branch.
   - the value of the displacement in the index with displacement addressing mode.
   - the condition code before and after the `scanbit` instruction.
   - the condition code before and after the `add` instruction.
   - the result of performing the `movq` instruction.

NOTE. *The significance of this example is not in the results of the running program, but in the code translation performed by xlate960 in the next few steps.*

5. At the (gdb960) prompt, enter: **`quit`**

To prove that `xlt.s` does **not** compile unaltered using the Rx Strategy, complete the following steps:

6. Enter the following command in the Command Prompt window provided:

   `gcc960 -AR{`*`P`*`|`*`D`*`} -Fcoff -Tmcyrx -o xlt xlt.s`

`-AR{`*`P`*`|`*`D`*`}`	sets the target architecture for the compiler. Since you are compiling for the Rx Strategy, use the available i960 Rx architecture options `-ARP` or `-ARD`.
`-Fcoff`	sets the object file type as COFF.
`-Tmcyrx`	uses the linker directive file for the i960 Rx architecture.
`-o xlt`	sets the object file name as `xlt` (optional).
`xlt.s`	specifies the input source file.

   There are errors during the compilation. The errors are:

```
xlt.s:83: Register is not in target architecture:
"(ip)".
xlt.s:126: indexed addressing mode not available
xlt.s:208: Opcode is not in target architecture:
"addi".
xlt.s:239: Opcode is not in target architecture:
"movq".
```

   These errors must be resolved before the program compiles using the `-ARD` or `-ARP` architecture flags.

xlate960 generates Rx-compatible code sequences to replace those instructions and addressing modes that appear in the JF processor causing errors above.

7. Enter the following command in the Command Prompt window provided:

   `xlate960 xlt.s`

   The previous command converts instructions in `xlt.s` to Rx-compliant instructions, placing the output into the file `xlt.xlt`.

   The output in the Command Prompt window is:

   ```
   C:\INTEL960\BIN\XLATE960.EXE:  Output file
   'xlt.xlt' requires further manual translation.
   ```

   This message above means you must edit the output file `xlt.xlt` to finish the translation to i960 Rx-compliant code.

You Are now ready to Edit the `xlt.xlt` file.

8. Open `xlt.xlt` in an editor.

   The output file produced by xlate960 is identical to the input file except for the instances where translation occurred. Each instruction that was translated is replaced with a sequence of the following format in the output file:

   ```
   #xlate-beginoriginal instruction
   <translation errors or warnings, marked by xlate-err
   or xlate-warn>
   <translation routine>
   #xlate-end
   ```

9. Find the translation points in `xlt.xlt` by searching the file for `#xlate-begin` flags.

   There are five translation points in the file.

   At the first translation point beginning on line 84 of `xlt.xlt`, note two `#xlate-err` translation errors and the `#xlate-warn` translation warning. The first translation error is:

   ```
       #xlate-err  "Fill in register for E0"
   ```

   The following two instructions are found on lines 89 and 90 of the translation routine:

   ```
   lda 24+9f-8b(g14),E0; 9:
   bx  (E0)
   ```

   Fill in a register that can be used for the place holder E0 that does not affect the program logic (i.e., choose a register that is not being used). In our example, it is all right to use register r13. So, edit the code and change E0 to r13:

   ```
   lda 24+9f-8b(g14),r13; 9:
   bx  (r13)
   ```

   The next translation error is:

   ```
   #xlate-err"Verify that register g14 can be clobbered"
   ```

   The translation routine uses register `g14` on line 89. Since `g14` can be overwritten, it does not need to be changed. The translation warning reported is:

   ```
   #xlate-warn"Verify use of local labels '8' and '9' "
   ```

   The translation routine uses the local labels '8' and '9'. Since they do not conflict with other local labels used in the program, no change is needed.

   Lastly, the original program, `xlt.s`, made a branch ahead by 24 plus the contents of the ip-register. The translation routine discredits the displacement number due to added instructions, and it is now necessary to change the displacement to 28.

10. So, edit the translation routine and change 24 to 28 to maintain the correct logic:

    ```
    lda 28+9f-8b(g14),r13; 9:
    ```

11. Find the next translation point; it is the following:

    ```
    #xlate-beginst r9,_VariableArray[r11*8]
    ```

    The translation error reported is:

    ```
    #xlate-err"Fill in register for E1"
    ```

    Like previously, all that is necessary is to use a register for the placeholder E1 that is not used and that does not affect the logic of the program. This time, register r15 is all right.

12. Edit the code on lines 138 and 139 from:

    ```
    shlo3,r11,E1
    st  r9,_VariableArray(E1)
    ```

    to the following:

    ```
    shlo3,r11,r15
    st  r9,_VariableArray(r15)
    ```

13. In order for the program to print the correct displacement after the translation, the code needs a little more editing. On line 128 of the `xlt.xlt` file, the following code segment begins:

    ```
    lda LC9,g0
    mov r15,g1
    callj_printf
    mov g0,g4
    ```

    Move this code segment to line 139 of the file. The segment thus occupies lines 139 through 142. Make sure to delete the code segment from lines 128 through 131.

14. Translation point three concerning the scanbit instruction had no translation warnings or errors.

15. View translation point four; it starts with the following:

    ```
    #xlate-beginaddi r10,r11,r8
    ```

    The translation warning for this translation routine is:

    ```
    #xlate-warn"Loss of faulting behavior"
    ```

    and the translation routine is:

    ```
    addor10,r11,r8
    ```

    xlate960 uses the xlate-warn comment lines to indicate instances where the translated code has subtle differences from the original code. Here, the `addo` instruction differs from the `addi` instruction because it does not fault when an overflow is generated. If overflow behavior is important to the program's operation, you would need to rewrite the code to manually check for an overflow condition.

16. Finally, view translation point five; it starts with the following:

    ```
    #xlate-beginmovq r8,g8
    ```

    The translation warning for this translation routine is:

    ```
    #xlate-warn"Does not test for unaligned or
    overlapping registers"
    ```

    and the translation routine is:

    ```
    mov r8,g8
    mov r9,g9
    mov r10,g10
    mov r11,g11
    ```

    Because our original code was 80960-compatible, the movq instruction was aligned and did not access overlapping registers. However, the translator draws our attention to the fact that invalid code would be generated when either of these conditions were present. Since neither are, you can ignore this warning.

17. The program has been manually translated. Close the `xlt.xlt` file.

Running the New Rx-compatible Source Code

1. Copy the xlt.xlt file to another file. At the command prompt, enter:

   `copy xlt.xlt xltconv.s`

2. To compile the Rx-compatible code, enter:

   `gcc960 -AR{`*`P`*`|`*`D`*`} -Fcoff -Tmcyrx -o xlt xltconv.s`

   The options in this command are:

`-AR{`*`P`*`|`*`D`*`}`	sets the target architecture for the compiler. Since you are compiling for the Rx Strategy, use the available i960 Rx architecture options `-ARP` or `-ARD`.
`-Fcoff`	sets the object file type as COFF.
`-Tmcyrx`	uses the linker directive file for the i960 Rx architecture.
`-o xlt`	sets the object file name as `xlt` (optional).
`xltconv.s`	specifies the input source file.

3. To run the program, enter:
   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci xlt`
   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`xlt`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**
   The program prints out:
   - the value of register r11 before and after the ip-relative branch.
   - the value of the displacement in the index with displacement addressing mode.
   - the condition code before and after the `scanbit` instruction.
   - the condition code before and after the add instruction.
   - the result of performing the `movq` instruction.
5. At the (gdb960) prompt, enter: **`quit`**

CONGRATULATIONS! You have translated source code written for earlier i960 processors. The source code is now Rx-compatible!

Assembler Pseudo-instruction Tutorial

This tutorial demonstrates the use of pseudo-instructions that have been added to the CTOOLS assembler to ease migration between processors. The tutorial that follows demonstrates how to enable and disable the instruction cache for the i960 Cx, Hx, Jx, and Rx microprocessors using microprocessor specific instructions. The tutorial then demonstrates how easy it is to enable and disable the instruction cache using only one pair of pseudo-instructions.

What Are Pseudo-instructions?

A number of pseudo-instructions (pseudo-ops) have been added to the CTOOLS assembler to ease the migration between processors. These pseudo-ops provide an architecture-independent method for performing some of the more common low-level processing operations. Using these pseudo-ops should reduce the number of changes required when moving assembly code from one i960 processor to another.

When you use any of the new i960 pseudo-instructions, you are required to re-assemble your source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best processor instructions to replace the pseudo-instructions based on the processor targeted.

pseudop.c: Editing the File for the Cx Microprocessor

1. If you are using the Hx Jx Cx & Sx QUICK*val* software, choose **Linker** and **Utilities**. If using the Rx QUICK*val* software, this step is not necessary.
2. Choose **Pseudo-op Tutorial**
3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.

4. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`. Both procedures contain no code initially. `cache_off()` looks like:

   ```
   cache_off()
   {

   }
   ```

5. Add the code necessary to disable the instruction cache for the Cx microprocessor. Between the brackets of the `cache_off()` procedure, add the following line exactly:

   ```
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   ```

   The `cache_off()` procedure should look like this:

   ```
   cache_off()
   {
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   }
   ```

   This procedure, `cache_off()`, uses the instruction cache control processor instruction `sysctl`. This instruction is valid in the i960 Cx processor for managing and controlling the instruction cache. `sysctl` is used above to disable the instruction cache. Also, the `CONFIGURE_ICACHE` and `DISABLE_ICACHE` constants are found in the `system.h` file that is included in the `pseudop.c` file.

6. Likewise, edit the `cache_on()` procedure adding the following line exactly:

   ```
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (ENABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   ```

   The cache_on() procedure should look like this:

   ```
   cache_on()
   {
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (ENABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   }
   ```

   Similarly, the `cache_on()` procedure for the Cx microprocessor uses the instruction cache control processor instruction `sysctl`. `sysctl` is used directly above to enable the instruction cache.

7. Save the `pseudop.c` file.

Running pseudop.c for the Cx Microprocessor

1. Compile and run the `pseudop.c` program to show that it works as desired.

NOTE. *If you do not have an i960 Cx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

2. In the Command Prompt window, enter the following commands:

   `gcc960 -AC{`*`F/A`*`} -Fcoff -Tmcycx -o pseudop pseudop.c`

   The options in this command are:

`-AC{`*`F/A`*`}`	sets the target architecture for the compiler. For this example, choose the Cx architecture, `-ACF` or `-ACA`
`-Fcoff`	sets the object file type as coff.

`-Tmcycx`	sets the linker directive file for the Cx architecture.
`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

If you have a Cx microprocessor and want to run the program, enter:

`gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

3. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works with instructions chosen specifically for the Cx architecture. Of course, this is what is expected. This program becomes more interesting when you start using pseudo-instructions.*

4. At the (gdb960) prompt, enter: **`quit`**

pseudop.c: Migrating the File to the Jx/Hx/Rx Microprocessor

Since the i960 Jx, Hx, and Rx microprocessors use the same processor instruction to enable and disable the instruction cache, this migration supports all three processors.

In order to use the program, `pseudop.c`, modified in the first part of this tutorial to support the Jx, Hx, or Rx microprocessor, it must first be migrated to those processors since they do not use the `sysctl` instruction to enable and disable the instruction cache.

1. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`.

   `cache_off()` contains the Cx specific code and looks like:

   ```
   cache_off()
   {
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   }
   ```

2. Change the code to disable the instruction cache for the i960 Jx/Hx/Rx microprocessors. Between the brackets of the `cache_off()` procedure, delete the previously added line and insert the following line exactly:

   ```
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   ```

   The `cache_off()` procedure should now look like this:

   ```
   cache_off()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   }
   ```

   This procedure, `cache_off()`, uses the instruction cache control processor instruction `icctl`. This instruction is valid in the 80960 Jx/Hx/Rx processors for managing and controlling the instruction cache. `icctl` is used above to disable the instruction cache. Also, the `ICACHE_OFF` constant is found in the `system.h` file that is included in the `pseudop.c` file.
3. Likewise, edit the `cache_on()` procedure deleting the previously added line and inserting the following line exactly:

   ```
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_ON),"d"(0),"d"(0));
   ```

   The `cache_on()` procedure should now look like this:

   ```
   cache_on()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_ON),"d"(0),"d"(0));
   }
   ```

   Similarly, the `cache_on()` procedure for the Jx/Hx/Rx microprocessors use the instruction cache control processor instruction `icctl`. `icctl` is used directly above to enable the instruction cache.
4. Save the `pseudop.c` file.

Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors

1. Compile and run the `pseudop.c` program to show that it works as desired.

NOTE. *If you do not have an i960 Jx, Hx, or Rx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

2. In the Command Prompt window, enter the following commands:

   For the Jx Microprocessor:

   ```
   gcc960 -AJ{F|D|A} -Fcoff -Tmcyjx -o pseudop pseudop.c
   ```

   The options in this command are:

`-AJ{AF|D|T}`	sets the target architecture for the compiler. For this example, to choose the Jx architecture, `-AJA`, `-AJF`, `-AJD`, or `-AJT`
`-Fcoff`	sets the object file type as coff.
`-Tmcyjx`	sets the linker directive file for the Jx architecture.
`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

   For the Hx Microprocessor:

   ```
   gcc960 -AH{D|A} -Fcoff -Tmcyhx -o pseudop pseudop.c
   ```

   The options in this command are:

`-AH{D|A}`	sets the target architecture for the compiler. For this example, to choose the Hx architecture, `-AHD` or `-AHA`
`-Fcoff`	sets the object file type as coff.
`-Tmcyhx`	sets the linker directive file for the Hx architecture.

`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

For Rx Microprocessor:

`gcc960 -AR{`*`P|D`*`} -Fcoff -Tmcyrx -o pseudop pseudop.c`

The options in this command are:

`-AR{`*`P|D`*`}`	sets the target architecture for the compiler. For this example, to choose the Rx architecture: `-ARP` or `-ARD`
`-Fcoff`	sets the object file type as coff.
`-Tmcyrx`	sets the linker directive file for the Rx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

3. If you have a Jx, Hx, or Rx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.

`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works with instructions chosen specifically for the architecture in question. Of course, this is what is expected. This program becomes more interesting when* you *start using pseudo-instructions.*

5. At the (gdb960) prompt, enter: **`quit`**

pseudop.c: Adding Pseudo-Ops to the Program

As can be seen, it is neither easy nor fun migrating code from one processor to another, especially when your code is many thousands of lines long. Fortunately, pseudo-instructions have been added to the CTOOLS assembler to ease migration between processors.

1. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. You are ready now to rewrite this program using pseudo-instructions. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`.

   `cache_off()` contains the i960 Jx/Hx/Rx microprocessor specific code:

```
cache_off()
{
__asm__ __volatile__("icctl
%0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
}
```

2. Change the code to disable the instruction cache for ALL processors. Between the brackets of the `cache_off()` procedure, delete the previously added line and insert the following line exactly:

   ```
       __asm__ __volatile__("ic_disable r5");
   ```

   The `cache_off()` procedure should now look like this:

   ```
   cache_off()
   {
       /* local register r5 is used to hold the status
   returned */
       __asm__ __volatile__("ic_disable r5");
   }
   ```

   This procedure, `cache_off()`, uses the pseudo-instruction `ic_disable`. When this program, `pseudop.c`, is compiled for a specific 80960 processor by using a `-A` architecture flag, the best instructions for that architecture are chosen to replace the `ic_disable` pseudo-op. Thus, pseudo-ops ease migration between processors. Also, notice only one argument to the pseudo-op is necessary. The `icctl` instruction requires three arguments. Programming with pseudo-ops can be simpler. Pseudo-instructions are also available to perform the other instruction cache management and controlling functions, such as cache invalidation.

3. Likewise, edit the `cache_on()` procedure deleting the previously added line and inserting the following line exactly:

   ```
   __asm__ __volatile__("ic_enable r5");
   ```

   The `cache_on()` procedure should now look like this:

   ```
   cache_on()
   {
         /* local register r5 is used to hold the status
   returned */
       __asm__ __volatile__("ic_enable r5");
   }
   ```

   Similarly, `cache_on()` uses a pseudo-instruction: `ic_enable`. When this program, `pseudop.c`, is compiled for a specific 80960 processor, the best instruction for that architecture is chosen to replace the `ic_enable` pseudo-op.

4. Save the `pseudop.c` file.

Running pseudop.c with Pseudo-instruction

1. Compile and run the `pseudop.c` program to show that the pseudo-instructions work as desired. To prove that the best instruction is chosen for the architecture, compile the code for the Cx microprocessor and then the Jx, Hx, or Rx microprocessor.
2. In the Command Prompt window, enter the following command:

   `gcc960 -AC{`*`F|A`*`} -Fcoff -Tmcycx -o pseudop pseudop.c`

   The options in this command are:

`-AC{`*`F|A`*`}`	sets the target architecture for the compiler. For this example, choose the Cx architecture, `-ACF` or `-ACA.`
`-Fcoff`	sets the object file type as coff.
`-Tmcycx`	sets the linker directive file for the Cx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

   When compiled, warnings may be generated. The warnings are generated just to point out this simple fact: when you use any of the new i960 pseudo-instructions, you are required to re-assemble your source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best instructions to replace the pseudo-instructions based on the processor targeted.
3. If you have a Cx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.

`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works as expected with pseudo-instructions.*

The result of this example is similar to using instructions specifically chosen for the Cx architecture. So, using pseudo-instructions can maintain the logic of your code, while easing migration to future i960 microprocessors.

5. At the (gdb960) prompt, enter: `quit`

NOTE. *If you do not have an i960 Jx, Hx, or Rx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors

Now you are ready to compile the code for the Jx, Hx, or Rx microprocessor to demonstrate similar results on a different processor.

1. In the Command Prompt window, enter the following commands:

   For the Jx Microprocessor:

   `gcc960 -AJ{`*`F|D|A`*`} -Fcoff -Tmcyjx -o pseudop pseudop.c`

   The options in this command are:

`-AJ{`*`A|F|D|T`*`}`	sets the target architecture for the compiler. For this example, to choose the Jx architecture, `-AJA`, `-AJF`, `-AJD`, or `-AJT`
`-Fcoff`	sets the object file type as coff.
`-Tmcyjx`	sets the linker directive file for the Jx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

   For the Hx Microprocessor:

   `gcc960 -AH{`*`D|A`*`} -Fcoff -Tmcyhx -o pseudop pseudop.c`

   The options in this command are:

`-AH{`*`D|A`*`}`	sets the target architecture for the compiler. For this example, to choose the Hx architecture, `-AHD` or `-AHA`
`-Fcoff`	sets the object file type as coff.
`-Tmcyhx`	sets the linker directive file for the Hx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

   For Rx Microprocessor:

   `gcc960 -AR{`*`P|D`*`} -Fcoff -Tmcyrx -o pseudop pseudop.c`

   The options in this command are:

`-AR{`*`P|D`*`}`	sets the target architecture for the compiler. For this example, to choose the Rx architecture, `-ARP` or `-ARD`
`-Fcoff`	sets the object file type as coff.

`-Tmcyrx`	sets the linker directive file for the Rx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

2. If you have a Jx, Hx, or Rx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

3. At the (gdb960) prompt, enter: **`run`**

   The result of this example is the same as using instructions specifically chosen for the Jx, Hx, or Rx architecture. So, using pseudo-instructions does not change the logic of the program. It only eases future migration of your code to future i960 microprocessors.

4. At the (gdb960) prompt, enter: `quit`

   When compiled, warnings may be generated. The warnings are generated just to point out this simple fact: When you use any of the new i960 pseudo-instructions, you are required to re-assemble your source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best instructions to replace the pseudo-instructions based on the processor targeted.

CONGRATULATIONS! You can now start using pseudo-instructions in your code to ease migration of your code to future i960 processors.

Debugging with gdb960

A software debugger is a useful tool that allows you to learn more about the behavior of an application program while it is running on a target or simulator. gdb960 is a source-level debugger that allows you to interact with your application program running on a target system through the debug monitor, MON960. MON960 is resident on the Cyclone CPU module.

This example uses the card game, Go Fish, and is designed to teach you a few debugger commands so that you can further examine the example programs provided with this kit or your own programs. In the card game, Go Fish, you and the computer each get several cards. You take turns guessing which cards are in each other's hands. When you guess correctly, you acquire that card. If you don't guess correctly, you need to "Go Fish" and draw another card from the pack. When you get four-of-a-kind, you remove those cards from your hand. The objective of the game is to have the most sets of four-of-a-kind when either you or the computer has no cards remaining in your hands.

NOTE. *This example uses the command line interface to gdb960. The program also features a Graphical User Interface in both Windows and UNIX. See The gdb960 User's Manual for more information.*

1. Choose **Debugger**.
2. Choose **gdb960 Tutorial**.

3. Choose **Make** to compile, link, and download the program automatically.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

NOTE. *DEBUGGING SHORTCUTS*
Abbreviations for gdb960 commands are accepted as long as they are unambiguous.
*To **run,** enter: **r***
*To **break,** enter: **br***
*To **list,** enter: **l***
*To **continue,** enter: **c***
*To **print,** enter: **p***
*To **clear,** enter: **cl***
*To **quit,** enter: **qu***
*For **help,** enter: **he***

4. **Do Not Type Run!** First, use the gdb960 debugger to set a breakpoint at function `main()`. Type:

   `break main`

   The debugger responds by displaying:

```
Breakpoint 1 set at 0xa0008570: file fish.c, line 209.
```

5. Set a second breakpoint at line 275. Type:

   `break 275`

   The debugger responds by displaying:

```
Breakpoint 2 set at 0xa0008bc4: file fish.c, line 275.
```

6. To execute the program from the beginning, type:

   `run`

   The debugger responds by displaying:

```
Starting program: C:\QUICKVAL/fish
Breakpoint 1, main() at fish.c, 209.
209    srand();
```

7. To display the code at the breakpoint, type:

   `list`

   The debugger displays lines 204-213 of the `fish.c` source. To see the next ten lines, type `list` again.

8. To continue executing the program from this location, type:

   `continue`

   The debugger responds by displaying:

   ```
   Continue.
   Would you like instructions[n]?
   ```

9. Reply by typing `y` for yes or <Enter> or `n` for no.

   ```
   your hand is: A A 6 6 8 8 9
   Breakpoint 2, game() at fish.c:275.
   275    if(!move(yourhand,myhand,g=guess(),0))break;
   ```

10. In the source code in step 9, there are two variable arrays, `myhand` and `yourhand`. `Myhand` is the computer's hand and `yourhand` is yours. To look at the card in the computer's hand, type:

    `print myhand`

    The debugger responds by displaying:

```
$1="000\000\000\001\000\002\000\001\000\000\001\002\000"
```

    `myhand[0]` does not represent a card.

    `myhand[1]` represents the number of Aces.

    `myhand[2]` represents the number of 2s, and so on.

    The same order of cards is represented in the array, `yourhand`.

    If a King is drawn by either player, `myhand[13]` or `yourhand[13]` will appear when you print the array.

11. Using the ability to see the computer's hand, you are able to beat the computer every time. Clear the first breakpoint at the function `main()` and continue playing the game, looking at the computer's hand any time you need to. To clear the breakpoint at `main()`, type:

    `clear main`

    The debugger responds by displaying:

    ```
    Deleted breakpoint 1
    ```

12. To continue executing the program, type:

    `continue`

13. If you need further assistance beating the computer, contact the 80960 Technical Support Group for more hints.
14. Type: **`quit`**

Debugging Optimized Code

CTOOLS can use the ELF object module format and DWARF Version 2 debug information format as described in the *80960 Embedded Application Binary Interface (ABI) Specification* (order number 631999). The new formats enable more accurate mapping between source and object code at higher optimization levels and ease production code debugging.

This example shows that at the highest level of module-local optimization, it is possible to set a breakpoint on an inline function using ELF/DWARF, while with COFF this is not possible.

1. Choose **Debugger**.
2. Choose **C ELF/DWARF Format**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Compile *swap.c* with no module-local optimizations (no inlining). This shows that the procedure *swap* is not inlined. Enter:

   ```
   gcc960 -Felf -T{Link-dir} -A{arch} -O0 -S swap.c
   ```

   The options in this command are:

`-Felf`	creates an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-O0`	no module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

5. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, see the call to the procedure swap:

   ```
   callj _swap
   ```

   This is an out-of-line call to the procedure swap. The function swap has not been inlined.

6. Now, compile `swap.c` with the highest level of module-local optimizations. This inlines the procedure `swap`.

   `gcc960 -Felf -T`*`{Link-dir}`*` -A`*`{arch}`*` -O4 -S swap.c`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-O4`	highest level of module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

7. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, note the call to the procedure `swap` does not exist:

   ```
   callj _swap  /* Does Not Exist*/
   ```

   The procedure swap has been inlined.

8. Recompile using the `-O4` optimization level, the ELF/DWARF format, and add debugging information.

`gcc960 -Felf -T`*`{Link-dir}`*` -A`*`{arch}`*` -O4 -g -o swap swap.c`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcyjx` specifies mcyjx.gld.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file

`-o swap`	names the executable file swap
`swap.c`	input file

9. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

```
gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap
```

The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`swap`	the executable file

10. DO NOT TYPE RUN!

First, set a breakpoint on the procedure *swap.* Enter:

```
break swap
```

The debugger responds by displaying:

```
breakpoint 1 @0xa00080f0:file swap.c, line 43
breakpoint 2 @0xa0008148:file swap.c, line 54
```

Breakpoint 1 is the out-of-line reference to the procedure `swap`. Breakpoint 2 is the inline reference to the procedure `swap`.

`Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is still possible to set a breakpoint on the inline function. Breakpoint 2 stops program execution.

11. To execute the program, enter:

```
run
```

The debugger responds by displaying:

```
Breakpoint 2, main() @ swap.c: 54
54 printf(ìThe smallest number is %d\nî,a);
```

12. To continue the program, enter:

```
c
```

When the program has finished, enter:

```
quit
```

13. Compile using the `-O4` optimization level, the COFF format, and add debugging information.

```
gcc960 -Fcoff -T{Link-dir} -A{arch} -g -O4 -o swap swap.c
```

The options in this command are:

`-Fcoff`	create a COFF format output file
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o swap`	names the executable file `swap`
`swap.c`	input file

14. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

```
gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap
```

The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`swap`	the executable file

15. DO NOT TYPE RUN!!

First, set a breakpoint on the procedure `swap`. Enter:

```
break swap
```

The debugger responds by displaying:

```
breakpoint 1 @0xa00080f0
```

Breakpoint 1 is the out-of-line reference to the procedure `swap`. Notice that no inline breakpoint has been set. This breakpoint does not stop execution of the program.

`Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is not possible to set a breakpoint on the inline function. Program execution does not stop.

16. To execute the program, enter:

    ```
    run
    ```

    The debugger responds by displaying the smallest number from the swap. There is no break in program execution.

17. When the program has finished, enter:

    ```
    quit
    ```

    You have now seen that with the ELF/DWARF format, it is now possible to debug your production code, even after high levels of program optimization.

Debugging Optimized C++ Code Tutorial

The C++ compiler generates debug information using the DWARF format when the `-g` option is specified with the `-Felf` option. This debug information format is richer than that of other supported OMFs, and allows more reliable debugging under optimization.

This tutorial demonstrates that at the highest level of module-local optimization, debugging a C++ application is still possible due to the DWARF debug format.

In this example, you compile a C++ program using the `-O0` optimization compiler option, which disables all optimizations, including those that may interfere with debugging. The same C++ program is then compiled using the highest-level of module-local optimization, `-O4`.

There are several levels of program optimization available with the CTOOLS development tool suite. Typically, low levels of optimization are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once the application is functioning properly, the application's performance may be increased by using a higher level of optimization. The static optimization options are:

`O0`	Turn optimization off
`O1`	Basic optimization
`O2`	strength-reduction, instruction scheduling for pipelining, etc...

`O3`	`O2` plus `fconstprop, finline-functions`, etc...
`O4`	`O3` plus `fsplit-mem, fmarry-mem, fcoalesce`

Level O4 is the highest level of static optimization. Please refer to the *i960 Processor Compiler User's Guide* for more information on ELF/DWARF and compiler optimizations.

In this tutorial, you compile and debug a C++ program, `cppdwarf.cpp`, that contains many of the advanced features of the C++ language, including:

- Classes
- Public, protected, and private variable accessibility
- Virtual functions
- Scope operators
- Overloaded functions
- Class inheritance

Using ELF/DWARF, both levels of optimization, `-O0` and `-O4`, retain the C++ program structure so that the above features may be investigated.

1. Choose **Debugger**.
2. Choose **C++ ELF/DWARF Format**.
3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Compile the program using the `-O0` optimization level. In the Command Prompt window, enter the following command:

   ```
   gcc960 -Felf -A{arch} -T{Link-dir} -stdlibcpp -O0 -g
   -o cppdwarf cppdwarf.cpp
   ```

   The options in this command are:

`-Felf`	creates an ELF format output file.
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD.
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcyjx` specifies mcyjx.gld.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.

`-O0`	specifies the lowest level of module-local optimizations.
`-g`	includes debug information in object file.
`-o cppdwarf`	specifies the executable file `cppdwarf`.
`cppdwarf.cpp`	specifies the input file `cppdwarf.cpp`.

5. Run the program using the debugger, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -D {`*`parallel port`*`} -pci cppdwarf`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, ... com99.
`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are lpt1 and lpt2.

`-pci`	sets the code download option for the PCI bus (optional). When no serial port is specified, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`cppdwarf`	specifies the executable file `cppdwarf`.

6. **Do Not Enter Run!**

   Now you are ready to examine some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes.

   At the (gdb960) prompt, enter:

   `ptype person`

   The following data type information concerning the class `person` appears:

Example 6-1 person Class

```
  type = class person {
    protected:
      char name[40];
      char dept[40];
    public:
      void setName ();
      void setName (char *);
      void setDept ();
      void setDept (char *);
      void printName ();
      virtual int isOutstanding ();
      virtual char * getDept ();
}
```

Please note the following concerning the above output:

- The entire class information for person is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.

Another C++ class in the program is `professor`, which inherits from the person class. Again, you use the gdb960 command `ptype` to display a description of the `professor` class.

7. At the (gdb960) prompt, enter:

   `ptype professor`

   The following data type information concerning the class `professor` appears:

Example 6-2 professor Class

```
type = class professor : public person {
  private:
    int numPubs;
  public:
    void setNumPubs ();
    void setNumPubs (int);
    virtual int isOutstanding ();
}
```

Please note the following concerning the above output:

- The entire class information for `professor` is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.
- `type = class professor : public person` indicates that the `professor` class inherits from the `person` class.

8. You are ready to set some breakpoints.
   a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

      `break professor::setNumPubs`

      The following information concerning breakpoints is displayed:

      ```
      [0] cancel
      [1] all
      [2] professor::setNumPubs(int) at
      cppdwarf.cpp:125
      [3] professor::setNumPubs(void) at
      cppdwarf.cpp:118
      ```

      Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.
   b. Set a breakpoint on all `professor::setNumPubs` functions. At the `>` prompt, enter: **`1`**

      The following information about breakpoints is displayed:

      ```
      Breakpoint 1 at 0xa00083d0: file cppdwarf.cpp,
      line 125.
      Breakpoint 2 at 0xa0008358: file cppdwarf.cpp,
      line 118.
      ```
   c. Set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

      `break professor::isOutstanding`

      The following information concerning breakpoints is displayed:

      ```
      Breakpoint 3 at 0xa0009080: file cppdwarf.cpp,
      line 110.
      ```
9. You are now ready to start the program. At the (gdb960) prompt, enter:

   `run`

   Notice that the program stops at all three of the breakpoints.
10. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.

11. At the (gdb960) prompt, enter: **`quit`**

    The results of the debug session were as expected because no optimizations had been performed on the source code during compilation. You can now recompile the `cppdwarf.cpp` program using the highest-level of module-local optimization and repeat the previous debug session.

12. Compile the program using the `-O4` optimization level. In the Command Prompt window, enter the following command:

    `gcc960 -Felf -A{`*`arch`*`}-T{`*`Link-dir`*`} -stdlibcpp -O4 -g -o cppdwarf cppdwarf.cpp`

    The options in this command are:

`-Felf`	create an ELF format output file
`-A{`*`arch`*`}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{`*`Link-dir`*`}`	specifies the linker directive file. For example, `-Tmcyjx` specifies `mcyjx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o cppdwarf`	specifies the executable file `cppdwarf`
`cppdwarf.cpp`	input file

13. Run the program using the debugger, enter:

    `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -D {`*`parallel port`*`} -pci cppdwarf`

    The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.

`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, ... com99.
`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are: lpt1 and lpt2.
`-pci`	sets the code download option for the PCI bus (optional). When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used.)
`cppdwarf`	specifies the executable file.

14. **Do Not Enter Run!**

    You are now ready to investigate some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes. At the (gdb960) prompt, enter:

    `ptype person`

    Please note, the output matches that of Example 6-1, "person Class". Optimizations did not affect the `person` class output. It is the same as the first debug session.

15. Another C++ class in the program is `professor`, which inherits from the person class. Once again, you use the gdb960 command `ptype` to display a description of the `professor` class. At the (gdb960) prompt, enter:

    `ptype professor`

    Again please note, the output matches that of Example 6-2, "professor Class". Optimizations did not affect the `professor` class output. It is the same as the first debug session.

16. You are now ready to set some breakpoints.

    a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

       `break professor::setNumPubs`

       The following information concerning breakpoints is displayed:

       ```
       [0] cancel
       [1] all
       [2] professor::setNumPubs(int) at
       cppdwarf.cpp:125
       [3] professor::setNumPubs(void) at
       cppdwarf.cpp:118
       ```

       Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 only sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.

    b. Set a breakpoint on all `professor::setNumPubs` functions, so At the `>` prompt, enter: **`1`**.

       The following information about breakpoints is displayed:

       ```
       Breakpoint 1 at 0xa00082e4: file cppdwarf.cpp,
       line 125.
       Breakpoint 2 at 0xa0008294: file cppdwarf.cpp,
       line 118.
       ```

    c. Finally, set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

       `break professor::isOutstanding`

       The following information concerning breakpoints is displayed:

       ```
       Breakpoint 3 at 0xa0008960: file cppdwarf.cpp,
       line 111.
       ```

17. You are now ready to start the program. At the (gdb960) prompt, enter:

    `run`

    Notice that the program does not stop at all three of the breakpoints. As can be seen, the DWARF debug information format is very rich, 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.
18. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.
19. At the (gdb960) prompt, enter: **`quit`**

CONGRATULATIONS! You may now know how to use ELF/DWARF to debug your optimized C++ code.

Writing Flash

NOTE. *In order to write to flash on your Cyclone base board, you need a 12 volt power supply. Also, these instructions are used with the CTOOLS 6.0 and MON960 3.2.3 toolsets.*

This example teaches you the following:

- Writing to flash on the Cyclone base board.
- Booting off of the flash in socket U27 of the Cyclone base board, as opposed to the flash on the CPU Module.
- Setting the Cyclone base board to 12 volts.
- Using *mondb.exe* as a simple utility to download and execute an application program on the target board running MON960.
- Using *mondb.exe* to write flash.
- Building a new monitor for a particular i960 microprocessor family member.
- Retargeting MON960 for other boards.

Complete this step:

1. Choose **MON960**.
2. Choose **Writing Flash.**
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Identify the Flash on the Cyclone base board.

   A blank Flash chip ships on each Cyclone base board in socket U22. To write MON960 to Flash, you must move the blank Flash from socket U22 to socket U27.
5. Set the Cyclone base board voltage to 12 volts.

   Locate the four-position DIP switch labeled S1. Flip S1.1 to the *ON* position. This enables VPP to the Cyclone base board Flash.
6. Power up the Cyclone eval base board

   Locate the four-pin connector that interfaces to a secondary power supply labeled J6. Three of the connector pins connect to +5 VDC, +12 VDC and ground. (On the PCI-SDK Platform, +12 VDC and +5 VDC power is supplied through the edge connector.)
7. Edit `Version.c`.

   a. Change directories to where the `version.c` file resides. The default installation directory for CTOOLS is:

      `c:\intel960\src\mon960\common`

      If you cannot find the mon960 directory, You need to install MON960 as directed in the *MON960 Debug Monitor User's Manual*.

      Version.c contains the following information:

```
const char mon_version_byte =  nn;   /* version n.n = nn */
const char base_version[] = "MON960 n.n.n";
const char build_date[] = __DATE__;
```

   b. Change the file contents to reflect that this is your version of MON960. For example, change

```
const char base_version[] = "MON960 n.n.n";
```

      to:

```
const char base_version[] = "MY MON960";
```

   c. Save `Version.c`.

8. Build the new MON960 from source (optional)

   By default the source for MON960 is located at:
   `c:\intel960\src\mon960\common`

   You may use the pre-built version of MON960 there, or build a custom verion. To create a custom version:

   a. Copy `makefile.xxx` to
      `c:\intel960\src\mon960\common\makefile`.

      where xxx is one of the following make files:

      `makefile.ic` (ic960 interface, COFF format)

      `makefile.ie` (ic960 interface, ELF format)

      `makefile.gc` (gcc960 interface, COFF format)

      `makefile.ge` (gcc960 interface, ELF format)

   b. Issue the commands:

      ```
      nmake -s makefile
      cyjx
      ```

      This creates a file called `cyjx.fls`.

9. Write the Flash

   To write the Flash, use the mondb.exe utility located in the `intel960\bin\` directory. If you are going to use the pre-built MON960 files, they are located in the `intel960\roms` directory. For example, if you used the default installation directory and are using the pre-built MON960 files for the 80960Jx, enter:

   ```
   mondb -ser com1 -par lpt1 -ef -ne
   c:\intel960\roms\cyjx.fls
   ```

   The options in this command are:

`-ser com1`	use serial port 1
`-par lpt1`	use parallel port 1
`-ne`	no execute
`-ef`	erase Flash
`cyjx.fls`	input Flash filename

   Note also that if you built a version of MON960 from the source code as described previously, the `cyjx.fls` file will be located in the `c:\intel960\src\mon960\common\` directory.

10. Set Board Voltage Back To +5 VDC

    Locate the four-position DIP switch labeled S1. Set S1.1 to the *OFF* position. This disables VPP to Cyclone EP base board Flash and protects the Flash. Note that the PCI80960DP and i960 `Jx` evaluation platforms do not boot when VPP is enabled and MON960 is running from the evaluation board Flash.

11. Set board to boot from U27 socket

    Locate the four-position DIP switch labeled S1. Set S1.3 ROMSWAP to the *ON* position. This exchanges the addresses of the CPU Module ROM and the base board ROMs. When the switch is *OFF* the processor boots from the CPU Module ROM; when the switch is *ON* the processor boots from the base board ROMs.

12. Reset Base Board

    Locate the reset pushbutton labeled S2. Use this button to manually reset the Cyclone base board and boot from the base board ROMs.

NOTE. *If you have trouble with this example, refer to Chapter 3 for troubleshooting tips.*

How to Add Benchmarking Routines to Your Code

Benchmarking is a common way to evaluate an architecture for its performance. CTOOLS comes with two routines for benchmarking code. These routines are called `bentime()` and `init_bentime()`. `init_bentime()` is called once to program the on-board Counter/Timer to periodically interrupt the processor. The `bentime()` routine returns the time in microseconds based on the count from the interrupt handler, `timer_isr`, and the current count read from the timer. By placing a call to `bentime()` at the start and end of the code you are timing, the elapsed time can be calculated by the difference between the second call to `bentime()` and the first.

1. Choose **Benchmarking**.
2. Choose **Qv Code**.

3. Scroll through the `chksum.c` code for comments that refer to "Benchmarking Routine". You can add similar lines to the code that you want to time.
4. Choose **Make** to compile, link, and download the program automatically.
5. Execute the `chksum` program. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 6.233021 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**

Other i960 Processor Choices and the Remote Evaluation Facility

The i960 RISC processor family has a wide breadth of processors to match your design's price and performance needs. If you wish to evaluate other i960 processor family members, contact your local distributor and order different Cyclone CPU modules, or visit the Remote Evaluation Facility at `http://developer.intel.com/design/i960/testcntr`

NOTE. *The i960 Rx Processor is not available through the Remote Evaluation Facility.*

If you choose to order more CPU modules, you may rest assured that all i960 processor modules plug-n-play with your QUICK*val* kit. This configuration was specifically designed to protect your investment and offer a low cost migration path for future needs.

The i960 Cx CPU Example Programs 7

The i960 CA and CF superscalar microprocessors represent Intel's commitment to provide a spectrum of reliable, cost-effective, high-performance processors that satisfy the requirements of today's innovative microprocessor-based products. The i960 Cx processors are designed for applications which require greater performance on a single chip than is usually found in an entire embedded system. The sheer speed of the i960 Cx processors enriches traditional embedded applications and makes many new functions possible at a reduced cost. These embedded processors are versatile; they are found in diverse products such as laser printers, X-terminals, bridges, routers, PC add-in cards and server motherboards.

Additionally, you can optimize your system's performance with CTOOLS, which includes a profile-driven compiler that can automatically optimize your code based on its runtime behavior.

The following pages describe the example programs included with this kit. Each example highlights a feature of the architecture or CTOOLS and provides you with source code that can help shorten your software development cycle. Table 7-1 provides descriptions of the tutorials included in the i960 Cx QUICK*val* kit.

Table 7-1 QUICK*val* i960 Processor Sample Programs

Tutorial Description	Source Files
Hello World: Uses simple printf statement to verify system integrity.	`hello.c`: source file `system.c`: system file
Memory Test: Used for system verification of external memory. The programs perform byte, short, or word writes to external memory, and then they check the addresses written for correctness.	`memtst8.c:` 8 bit memory test `memtst16.c`: 16 bit memory test `memtst32.c`: 32 bit memory test `system.c`: system file
Data Cache: Uses the minimum edit distance algorithm to demonstrate the effectiveness of the on-chip data cache. This example also shows how to enable and disable the data cache and how to configure an area of memory for caching.	`dcache.c`: source file `system.c`: system file
Instruction Cache: Uses a simple loop to demonstrate how to enable and disable the instruction cache. It also highlights the performance advantage obtained when using the on-chip instruction cache.	`loop.c`: source file `system.c`: system file
External Interrupts: Shows how to configure the Cyclone board timers to trigger hardware interrupts. This is also an example of using interrupt handlers and placing the handlers in the interrupt table.	`cyint.c`: source file `asm_fns.s`: interrupt handler for Sx `int_proc.s`: interrupt handler-all processors but Sx `t85c36.c`: eval board timer file `system.c`: system file
Fault Handling: Shows how to set up the fault handling procedures in the fault and system procedure tables.	`fault.c`: source file `flt_proc.c`: fault procedures `asm_flt.s`: assembly functions to help generate faults `system.c`: system file

continued ☛

Table 7-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
DMA Controller (i960 Cx): Provides an example of programming the DMA controller of the 80960 CX microprocessor. This example is setup for block mode chaining transfer.	`dma.c`: source file `int_rout.c`: DMA interrupt handling routines `dma.s`: configures DMA channel 0 and provide chained linked buffers. `system.c`: system file
C Local Optimizations: Shows how to use the C compiler with high levels of static optimization for improved runtime performance.	`chksum.c`, `system.c`: source files
C Global Optimizations: Shows how to use program-wide optimizations of the C compiler for increased performance.	`chksum.c`, `system.c`: source files
C++ Local Optimizations: Shows how to use the C++ compiler with high levels of static optimization for improved runtime performance.	`optimize.cpp`: source file
C++ Global Optimizations: Shows how to use program-wide optimizations of the C++ compiler for increased performance.	`optimize.cpp`: source file
C++ Virtual Function Optimizations: Shows how a call to a virtual function can be replaced by a direct call to a member function, and, if possible, it may be inlined at the call site. This improves the runtime performance of the code.	`optimize.cpp`: source file
Profiling Lab: Teaches you how to use some of CTOOLS advanced profiling features.	`chksum.c`: source file
Self-Contained Profile: Shows how to create a self-contained profile that captures the program structure and associates it with the program counters from a raw profile. When the source program changes, the global decision making step interpolates or stretches the counters in the self-contained profile to fit the changed program.	`quick.c`: source file

continued ☛

Table 7-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
Incremental Profiling: Shows how to profile a program in pieces and then re-combine them later, a useful methodology when the target execution environment is memory limited	`fault.c`, `flt_proc.c`, `asm_flt.s`, `system.c`: source files
C Cave: Uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.	`ttt.c`: source file
C++ Cave: Shows how to reduce target memory requirements. The text sections of compressed and uncompressed C++ executables are compared. This example also shows how to specify functions for compression.	`cavecpp.cpp`: source file
Linker Directive Language: Provides a hyperlinked manual that describes the linker command options. This tutorial is found in the online help only, not in this manual.	
Linker Consumption: Shows the ability of the linker, gld960, to consume b.out-format, COFF, or ELF object files and libraries in any combination.	`cyint.c`, `int_proc.s`, `t85c36.c`, `system.c`: source files
i960 Processor Assembler Pseudo-Instruction Support: Shows how to use the new assembler pseudo-ops.	`pseudop.c`: source file
Debugging with gdb960: Uses the Go Fish card game to teach a few useful debugger commands.	`fish.c`: source file `system.c`: system file
ELF/DWARF Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to set a breakpoint on an in-line function.	`swap.c`: source file

continued ☛

Table 7-1 QUICK*val* i960 Processor Sample Programs (continued)

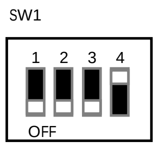
Tutorial Description	Source Files
C++ DWARF-2 Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to debug a C++ application.	`cppdwarf.cpp`: source file
Retargeting MON960: Provides steps for retargeting MON960. This tutorial is found in the online help only, not in this manual.	
Writing Flash: Demonstrates how to update the version of MON960 on your evaluation board.	
80960 Family Benchmark: Shows how to use this facility to compare your processor's performance with other i960 family members. This example uses a typical checksum routine to show how to add benchmarking routines into source code.	`chksum.c`, `system.c`: source files
Remote Evaluation Facility: Guides you through the use of this new benchmarking facility on the World-Wide Web.	

System Validation

Hello World

The program `hello.c` is used to verify your software and hardware system integrity. The following steps provide instructions on how to compile, link, download, and execute this program.

1. Verify that your software and hardware have been installed according to the instructions in Chapter 2 through 3 and the frequency switch on your CPU module is set as shown. The switch settings below set the 80960Cx CPU module frequency at 40 MHz.



2. Power your Cyclone evaluation platform and i960 Cx CPU module
3. Double-click on the **Hx Jx Cx & Sx QUICK*val*** icon in the QUICK*val* program group.
4. Configure you hardware.
   - Select the **80960 Architecture** tab
   - Select **Cx**.
   - Depending on the board you have installed, select either the EP80960BB or PCI80960DP tab.
   - Configure the software communication options to match those of your evaluation board.
   - Choose **OK**
5. Choose **Hello World**.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute hello. Type:
   `run`
8. The gdb960 debugger responds by displaying:

```
Hello...Welcome to the 80960CX QUICKval Kit!
SYSTEM CHECK COMPLETED!!
Now you may proceed with our Example Programs.
Program Exit: 01
(gdb960)
```

9. To exit the debugger, type: **`quit`**

CONGRATULATIONS! You have successfully installed your software and your hardware, compiled a program using gcc960, and downloaded and executed the program on your evaluation board using the gdb960 debugger.

If you received any error messages during this process, refer to "If Something Goes Wrong" on page 7-8.

Memory Test

The programs `memtst8.c`, `memtst16.c`, and `memtst32.c` are used to test the external memory on the Cyclone base board.

Depending on the test that is run, an 8, 16, or 32-bit test is run on an area of memory. The program writes F's and 0's to a memory location and reads the location to verify the integrity of what was written. All three programs are almost identical, with the exception of the casting of the variable *ADDR, which allows you to perform different test types.

NOTE. *Below,* `memtst*.c` *refers to either the byte, short, or word memory test example.*

1. Choose **Memory Test**.
2. Choose a memory test. The options are, **8-bit Memory Test**, **16-bit Memory Test**, or **32-bit Memory Test**.
3. Choose **Make** to compile, link, and download the program automatically.
4. Use the gdb960 debugger to execute memtst. Type:
   `run`
5. For the 8-bit test, `memtst8.c`, the gdb960 debugger responds by displaying:

```
This program will run a 8-bit test on the external memory.

Test to be implemented is byte test.
Starting address = a000dfb0
Ending address = a000ec30

Press enter to begin test with 0’s.
Number of errors that occurred is 0.

Begin test for f’s.
```

```
Press enter to continue.
Number of errors that occurred is 0.

All tests are complete.
Program exited with code 030.
(gdb960)
```

6. Exit the debugger. Type:

   `quit`

If Something Goes Wrong

The following section describes a few actions that may help resolve errors that may have occurred when invoking one of the tools. If you were unable to get the proper response from the gdb960 debugger after executing the above programs and the trouble-shooting hints described below do not help, contact the 80960 Technical Support Group by phone at 1-800-628-8686 or by E-mail at 960tools@intel.com.

MON960 Debug Monitor is Not Responding...

If the red FAIL LED (CR6) on the base board is lit, the monitor may not have booted up correctly. Press the reset button (S2). If the red FAIL LED remains lit, contact the 80960 Technical Support Group.

Invoking the gcc960 Compiler Resulted in Errors...

The environment must be set-up as described in Chapter 2. If you chose the default directories while installing CTOOLS, verify that the path names `C:\INTEL960\BIN` have been added to your PATH variable and that the following statement is in your `autoexec.bat` file. If you did not install these tools using the default directories, make the appropriate change.

```
SET G960BASE=C:\INTEL960
```

NOTE. *You did not use the default directories on installation, please make sure the G960BASE environment variable is assigned appropriately.*

NOTE. *Don't forget to re-boot your system once you have made any necessary changes to your* `autoexec.bat` *file.*

Invoking the gld960 Linker Resulted in Errors...

Verify that the directory that contains the `hello.c` and `memtst*.c` example programs also now has the object files, `hello.o` and `memtst*.o`. If `hello.o` and `memtst*.o` do not exist, then the gcc960 compiler command did not successfully create an object file. Re-compile `hello.c` and `memtst*.c` to see if an error occurred during the compilation.

If `hello.o` and `memtst*.o` do not exist, make note of the error message and contact the 80960 Technical Support Group.

Invoking the gdb960 Debugger Resulted in Errors...

NOTE. *If you are using the PCI-SDK evaluation platform, you may specify* `-pci` *for PCI download and PCI communication.*
For a list of all the gdb960 command line options, at a command prompt, enter: **`gdb960 -h | more`**

Serial communication error

A serial communication error causes the gdb960 debugger to respond by displaying:

```
HDIL error (10), communication failure
HDIL error (10), communication failure
You can't do that when your target is 'exec'
```

Verify that the serial port you are using is the one you specified in the gdb960 command line. Verify that your serial cable is properly connected to the board and to your PC.

Parallel communication error

A parallel communication error causes the gdb960 debugger to respond by displaying:

```
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type ‘show copying’ to see the conditions.
There is absolutely no warranty for GDB; type “show warranty” for
details.
gdb960.exe 6.0, Wed FEB 16 12:33:16 1998
GDB 5.10 (i486-intel-dos --target i960-intel-mon960), Copyright 1997
Free Software Foundation, Inc...(no debugging symbols found)...
Connected to com1 at 115200 bps.
(gdb960)
section 0, name .text, address 0xc0008000, size 0x50ec, flags 0x20
          writing section at 0xc0008000
```

Verify that the parallel port you are using is the one you specified in the gdb960 command line. Verify that your parallel cable is properly connected to the board and to your PC.

Data Cache Tutorial (80960CF Only)

The i960CF processor has a 1-Kbyte direct mapped data cache which enhances performance by reducing the number of load and store accesses to external memory. The data cache can return up to a quad word (128 bits) to the register file in a single clock cycle on a cache hit.

External memory is configured as cacheable or non-cacheable on a region-by-region basis, using special bits in the memory region configuration registers MCON0-15. This makes it easy to partition a system into cacheable regions (local memory) and non-cacheable regions.

The i960CF processor implements a simple coherency mechanism. The data cache can also be enabled, disable or invalidated on a global basis through programming.

This example uses the Minimum Edit Distance (MED) algorithm in order to show the effectiveness of using the data cache. The MED algorithm finds the minimum number of edit steps required to change one string into another.

This example is a real world example of using the data cache. This algorithm maintains a cost matrix to determine which change to the string being edited would incur the least cost. The cost matrix is a 2-D array [1..n][1..m], where n and m are the sizes of the two strings.

The algorithm really shows the speed of the data cache due to three reads for each write to the cost matrix. The algorithm reads from the cache to determine which step to take next, then writes its choice in the cost matrix. Since the writes to the data cache are write-through, there is no improvement for writes to the data cache. The Write-Through feature maintains coherency between the data cache and external memory.

The source code includes system files, `system.c` and `system.h`, that includes a macro and an assembly function that simplifies issuing data cache control instructions.

1. Choose **Cache Examples**.
2. Choose **Data Cache**.
3. Choose **Qv Code**.
4. Scroll through the `dcache.c` code to see the calls to the macro, `dcctl_contrl`.
5. Open and scroll through the `system.h` and `system.c` code to see the macro and assembly function, `dcctl_control` and `i960_dcctl`.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `dcache`. Type:

   `run`

   The debugger responds by displaying:

```
Minimum Edit Distance algorithm makes reads from the data cache.
This routine will determine how many steps are needed to convert:
StringA:  80960 QUICKval EvalKit
TO StringB:  i960(R) HxJxCxSx & Kx
Starting timed routine with data cache on ...
RESULT: 18 moves are required to convert string A to string B
Elapsed Time On = 0.001446 seconds
Elapsed Time for routine with data cache off ...
```

```
RESULT: 18 moves are required to convert string A to string B.
Elapsed Time Off = 0.003189 seconds
IMPROVEMENT: 54.7 percent
(gdb960)
```

8. Type: `quit`
9. Select **Results**.

NOTE. *Your actual run times may vary.*

Instruction Cache Tutorial (80960CF Only)

The instruction cache enhances performance by reducing the number of instruction fetches from external memory. The cache provides fast execution of cached code and loops of code in the cache and also provides more bus bandwidth for data operations in external memory. The i960 Cx processors' instruction cache is a two-way set associative cache, organized in two sets of eight-word lines. Each line is composed of four two-word blocks which can be replaced independently.

- The i960CA processor cache is 1 Kbyte, organized as two sets of 16 eight-word lines.
- The i960CF processor cache is 4 Kbyte, organized as two sets of 64 eight-word lines.

The loop.c program demonstrates the performance boost obtained by running a loop completely within versus outside of the instruction cache.

The source code includes system files, system.c and system.h, that includes a macro and an assembly function that simplifies issuing instruction cache control instructions.

1. Choose **Cache Examples**.
2. Choose **Instruction Cache**.
3. Choose **Qv Code.**
4. Scroll through the `loop.c` code to see the calls to the macro, `icache_control`.

5. Open and scroll through the `system.h` and `system.c` code to see the macro and assembly function, `icache_control` and `i960_icctl`.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `loop`. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/loop
   Simple loop timed with instruction cache off ...
   Elapsed Time Off = 3.341 seconds
   Simple loop timed with instruction cache on ...
   Elapsed Time On = 0.873 seconds
   IMPROVEMENT : 73.9 percent
   Program exited with code 01
   (gdb960)
   ```
8. Type: **`quit`**
9. Select **Results**.

External Interrupts Tutorial

The purpose of this program, `cyint.c`, is to show the steps required when dealing with an interrupt triggered externally by the evaluation board timers. The `cyint.c` source code contains step-by-step instructions to save you time when you program interrupts for your application. `int_proc.s` is the interrupt handler, and `t85c36.c` contains the functions to program the evaluation board timers.

The example performs the following steps in the handling of a hardware interrupt.

- Modify the ICON register
- Modify the IMAP register
- Cache the interrupt vector and the interrupt handling procedure
- Lower the processor priority
- Modify the IMSK register
- Clear the IPND register
- Generate the hardware interrupt using the evaluation board timers

Complete these steps:

1. Choose **Interrupt Examps**.
2. Choose **External Interrupts**.
3. Choose **Qv Code**.
4. Scroll through the `cyint.c` source to see the code for setting up and handling a hardware interrupt triggered by the evaluation board timers.
5. Open and scroll through the `t85c36.c` and `t85c36.h` files to see the definitions and routines for programming the evaluation board timers. You can simplify the programming of the evaluation board timers by including this code in your own applications.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `cyint`. Type:

   `run`

   The debugger responds by displaying:

   ```
   interrupt count = 70
   interrupt count = 85
   interrupt count = 98
   interrupt count = 110
   interrupt count = 122
   interrupt count = 134
   interrupt count = 147
   interrupt count = 159
   interrupt count = 171
   Program exited with code 020.
   (gdb960)
   ```
8. Type: **`quit`**

NOTE. *Your actual interrupt counts may vary.*

Fault Handling

These programs, `fault.c`, `flt_proc.c`, `asm_flt.s`, and `system.c`, show the steps taken in setting up the fault handling procedures in the fault and system procedure tables. The faults are then triggered one by one.

Table 7-2 i960 Cx Processor Fault Types and Subtypes

Fault Type		Fault Subtype		Fault Record
Number	**Name**	**Number or Bit Position**	**Name**	
0H	PARALLEL	NA		See your microprocessor user's manual
1H	TRACE	Bit 1	INSTRUCTION	0001 0002H
		Bit 2	BRANCH	0001 0004H
		Bit 3	CALL	0001 0008H
		Bit 4	RETURN	0001 0010H
		Bit 5	PRERETURN	0001 0020H
		Bit 6	SUPERVISOR	0001 0040H
		Bit 7	MARK	0001 0080H
2H	OPERATION	1H	INVALID_OPCODE	0002 0001H
		2H	UNIMPLEMENTED	0002 0002H
		3H	UNALIGNED	0002 0003H
		4H	INVALID_OPERAND	0002 0004H
3H	ARITHMETIC	1H	INTEGER_OVERFLOW	0003 0001H
		2H	ZERO-DIVIDE	0003 0002H
4H	Reserved			
5H	CONSTRAINT	1H	RANGE	0005 0001H
6H	Reserved			
7H	PROTECTION	Bit 1	LENGTH	0007 0002H
9H	Reserved			
AH	TYPE	1H	MISMATCH	000A 0001H
BH - FH	Reserved			

1. Choose **Fault Handling**.
2. Choose **QV Code**.
3. Scroll through the `fault.c` code to see a call to the function load_flt_proc(). This function loads the fault handling procedures into the fault and/or the system table.
4. Open and scroll through the `flt_proc.c` and `asm_flt.s` files. The `flt_proc.c` file contains the fault handling procedures, and the file `asm_flt.s` is used to help generate the faults.
5. Choose **Make** to compile, link, and download the program automatically.

NOTE. *When compiling, disregard the compiler warning:*
```
Warning: unaligned register
```
This is one of the faults that will be handled.

6. Use the gdb960 debugger to execute `fault`. Type:

   `run`

   The debugger responds by displaying the Fault Type and the Fault Subtype for each fault handled. The address of the faulting instruction is given (see Table 7-2).
7. Type: **`quit`**

DMA Tutorial

A key enhancement of the i960 Cx processor — only available on this i960 processor family member — is the integrated DMA controller.

The DMA controller concurrently manages up to four independent DMA channels. Each channel supports memory-to-memory transfers where the source and destination can be any combination of internal data RAM or external memory. The DMA mechanism provides two unique methods for performing DMA transfers:

- Demand-mode transfers (synchronized to external hardware). Typically used for transfers between an external device and memory.

- Block-mode transfers (non-synchronized). Typically used to move blocks of data within memory.

To perform a DMA operation, the DMA controller uses microcode, the core's multi-process resources, the bus controller and internal hardware dedicated to the DMA controller. For more information, please reference the *i960 Cx Processor User's Manual*, chapter 13.

1. Choose **DMA**.
2. Choose **QV Code**.
3. Scroll through the `dma.c` code to see the steps for setting up the DMA controller which are listed directly to the left.
4. Open and scroll through `int_rout.c` and `sdma.s` files. `int_rout.c` is the DMA interrupt handling routine, and `sdma.s` will configure DMA channel 0 and provide chained linked buffers.
5. Choose **Make** to compile, link, and download the program automatically.
6. Use the gdb960 debugger to execute `dma`. Type:

   ```
   run
   ```

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/dma
   Buffer transfer #1 complete.
   Buffer transfer #2 complete.
   Buffer transfer #3 complete.
   Program exited with code 041.
   (gdb960)
   ```

   The Buffer Transfers above used block-mode transfers (non-synchronized). This method moved blocks of data within memory.
7. Exit the debugger, type:

   ```
   quit
   ```

Static, Global, and Profile-Driven Optimizations

Optimizing compilers provide you with a means of developing high performance code without detailed knowledge of the architecture. Engineers who understand the features of the i960 architecture developed gcc960 to provide optimizations that take full advantage of the i960 processor. In general, optimizing compilation takes more time and may require more memory for large functions. However, the benefit in runtime performance is well worth it.

There are several levels of optimization available. Typically, low levels of optimizations are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once your application is functioning properly, you can increase its runtime performance by using a higher level of optimization.

Release 5.0 and later of the development tools support the ELF object module format and DWARF version 2.0 debug information format. The new format enables more accurate mapping between source and object code at higher optimization levels and ease debugging of production code.

The C optimization example uses a program called `chksum.c`. The C++ examples use a program called `optimize.cpp`

C No Optimization

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.
4. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 12.928249 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**

C Static Optimization

Use the following commands to compile the `chksum.c` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.
4. Choose **Make -O4** to compile with optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 1.967685 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ No Optimization

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.
5. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 7.1205 seconds.
   Program exited normally
   ```
7. Type: **`quit`**

C++ Static Optimization

Use the following commands to compile the `optimize.cpp` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.
5. Choose **Make -O4** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 5.80158 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

C Global Optimization

Use the following commands to compile the `chksum.c` with program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C Global Optimizations**.
4. Choose **Make +O5** to compile with optimizations, link, and download the program automatically.

5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 1.945978 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ Global Optimization

Use the following commands to compile the `optimize.cpp` program using the program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Global Optimizations**.
5. Choose **Make+05** to compile with optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 5.82517 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation

A 85% improvement in C code performance is significant, but there is another level of optimization that is uniquely available through Intel's CTOOLS compilers: profile-driven optimization. This two-pass compilation procedure allows the compiler to make optimizations based on runtime behavior as well as the static information used by conventional 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. Standard (i.e., non-program-wide) optimizations are referred to as 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 before the end of the linking phase.

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.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Profiling Lab**.
4. Follow the **Profiling Tutorial** link in the online help.

Using profile-driven optimization, an increase in runtime performance of 1.1% is obtained. The average 80960 application can expect to gain 15 to 30% performance improvement through the use of this technology. This boost in performance is available to you without any further investment in hardware.

C++ Virtual Function Optimizations

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, however, 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.

Use the following commands to compile the `optimize.cpp` program.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Virtual Opts**.
5. Choose **Make -NoVOpt** to compile without virtual function optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 5.82517 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Make -VOpt** to compile with virtual function optimizations, link, and download the program automatically.

9. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 4.941154 seconds.
   Program exited normally
   ```
10. Type: **`quit`**
11. Choose **Results**.

The virtual function optimizations yielded a 15.2% improvement.

Note the runtime performance at each optimization level as shown below.

Table 7-3 i960 Processor Optimization Results

Optimization Level	C Execution Time	C++ Execution Time
no optimization (-O0)	12.928249 seconds	7.1205 seconds
maximum static (-O4)	1.967685 seconds	5.80158 seconds
global optimization	1.945978 seconds	5.82517 seconds
profile-driven	1.945967seconds	NA
Virtual Function Optimization	NA	4.941154 seconds

Building Self-contained Profiles with gmpf960

A *raw* profile contains program counters that record 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 has a very short useful life. When changes are made in the 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 used to optimize a program 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.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Self-Contained**.
4. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
5. Specify the program database directory.

   The PDB can be specified by setting the environment variable `G960PDB`.

   For example, if you chose the default directory during installation, enter:

   ```
   SET G960PDB=C:\quickval\prof_lab\lab_pdb
   ```

   Or, specify the PDB at compiler invocation time with the `Z`*`dir`* option, as shown in the example below.

   ```
   gcc960  -Zmypdb  foo.o
   ```
6. Compile for profile instrumentation.

   Insert profile instrumentation into *quick* so that when the linked program is executed, a profile can be collected. Type:

   ```
   gcc960 -Fcoff  -T{Link-dir} -A{arch} -fdb
   -gcdm,subst=:*+fprof -o quick quick.c
   ```

   The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.

`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-gcdm,subst=:*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only eligible modules not linked in from libraries.
`+fprof`	causes generation of profile instrumentation.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

7. Collect 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. Type:

   ```
   gdb960 -t mon960 -b 115200 -r com1 -D lpt1 quick
   ```

   The options in this gdb960 compiler command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`quick`	the executable file

8. Use the gdb960 debugger to execute `quick`. Enter:

   `run`

9. Exit the debugger. Enter:

   `quit`

10. Enter the command:

    `gmpf960 -spf quick.pf default.pf`

    The options in this gmpf960 compiler command are:

`-spf`	causes a self-contained profile, `quick.pf`, to be produced as output.
`default.pf`	The input profile.

11. Recompile the `quick.c` source code using the profiling information obtained by the instrumentation. Type:

    `gcc960 -Fcoff -T`*`{Link-dir}`*` -A`*`{arch}`*` -fdb`
    `-gcdm,iprof=quick.pf -o quick quick.c`

    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=quick.pf`	This supplies a profile file `quick.pf` to the global decision making and optimization step.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

12. Change the control structure of `quick.c.`

    Edit `quick.c`. Find the procedure called QUICK. In this procedure, there is a control structure:

```
for(i = 2; i <= SORTELEMENTS; i+=1)
{
    (LOGIC)
}
```

Change the control structure to:

```
i = 2;
while (i <= SORTELEMENTS)
{
    (LOGIC)
    i+=1;
}
```

13. Compile the new `quick.c` using the interpolated profile. Type:

    `gcc960 -Fcoff -T`*`{Link-dir}`* `-A`*`{arch}`* `-fdb`
    `-gcdm,iprof=quick.pf -o quick quick.c`

    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=quick.pf`	This supplies a profile file `quick.pf` to the global decision making and optimization step.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

Notice that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`.

NOTE. *The beauty of this example is that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`, not the results of running this example.*

Profiling A Program In Pieces

Suppose that the target execution environment is memory limited so that all your programs cannot be instrumented for profiling at the same time. You can use substitutions to make partially instrumented versions of the final executable, and then create self-contained profiles for each piece. Each executable created in this way has a limited set of instrumented modules.

After you've created the self-contained profiles, you can use gmpf960 to create a single merged self-contained profile. The final, merged self-contained profile is identical to a profile obtained by instrumenting the entire program at once.

In this example, you use the fault handling example programs to show incremental profiling.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations.**
3. Choose **Incremental**.
4. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
5. Specify the program database directory.

   You can specify the PDB by setting the environment variable `G960PDB`. For example, if you chose the default directory during installation, enter:

   ```
   SET G960PDB=C:\quickval\prof_lab\lab_pdb
   ```

   Or, specify the PDB at compiler invocation time with the `Zdir` option, as shown in the example below.

   ```
   gcc960  -Zmypdb  foo.o
   ```
6. Insert profile instrumentation into `fault` so that when the linked program is executed, a profile can be collected. The instrumented modules in this version of `fault` are from the files `fault.c` and `flt_proc.c`. Type:

   ```
   gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
   -gcdm,subst=:f*+fprof -o fault fault.c flt_proc.c
   asm_flt.s system.c
   ```

`-Fcoff`	creates a COFF format output file.

`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-fdb`	all modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-gcdm,subst=:f*`	
	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only the files that begin with *f*.
`+Fprof`	causes generation of profile instrumentation.
`-o fault`	names the executable file `fault`.
`fault.c`	the source files.
`flt_proc.c`	the fault procedures.
`asm_flt.s`	the assembly file to generate faults.
`system.c`	system file.

7. Collect the profile.

   When 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. Type:

   `gdb960 -t mon960 -b 9600 -r com1 -D lpt1 fault`

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`fault`	the executable file

8. Use the gdb960 debugger to execute `fault`. Enter:

   `run`

9. Exit the debugger. Enter:

   `quit`

10. Build the self-contained profiles with gmpf960.

    To create a self-contained profile, use the gmpf960 profile merger tool. gmpf960 is invoked with the raw profile as an input file. Enter:

    `gmpf960 -spf prof1.pf default.pf`

`-spf`	causes a self-contained profile, `prof1.pf`, to be produced as output.
`default.pf`	The input profile.

    The resultant self-contained profile, `prof1.pf`, has a limited set of instrumented modules.

11. Insert profile instrumentation into `fault` so that when the linked program is executed, a profile can be collected. The instrumented modules in this version of `fault` are from the file `system.c`. Type:

    ```
    gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
    -gcdm,subst=:s*+fprof -o fault fault.c flt_proc.c
    asm_flt.s system.c
    ```

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,subst=:s*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only the files that begin with `s`.
`+fprof`	causes generation of profile instrumentation.
`-o fault`	names the executable file `fault`.
`fault.c`	the source files

`flt_proc.c`	the fault procedures
`asm_flt.s`	the assembly file to generate faults
`system.c`	system file

12. Collect the 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. Type:

    ```
    gdb960 -t mon960 -b 9600 -r com1 -D lpt1 fault
    ```

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`fault`	the executable file

13. Use the gdb960 debugger to execute `fault`. Enter:

    ```
    run
    ```

14. Exit the debugger. Enter:

    ```
    quit
    ```

15. Build the self-contained profiles with gmpf960.

    To create a self-contained profile, use the gmpf960 profile merger tool. gmpf960 is invoked with the raw profile as an input file. Enter:

    ```
    gmpf960 -spf prof2.pf default.pf
    ```

`-spf`	causes a self-contained profile, `prof2.pf`, to be produced as output.
`default.pf`	the input profile.

    The resultant self-contained profile, `prof2.pf`, has a limited set of instrumented modules.

16. Merge all the self-contained profiles into one.

    The final `prof.pf` profile is identical to a profile obtained by instrumenting the entire program at once. Type:

    ```
    gmpf960 -spf prog.pf prof1.pf prof2.pf
    ```

`-spf`	causes a self-contained profile, `prog.pf`, to be produced as output.

`prof1.pf`	an input self-contained profile.
`prof2.pf`	an input self-contained profile.

17. Recompile the fault handling source code using the profiling information obtained by the instrumentations. Type:

```
gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
-gcdm,iprof=prog.pf -o fault fault.c flt_proc.c
asm_flt.s system.c
```

`-Fcoff`	create a COFF format output file.
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-fdb`	all modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=prog.pf`	This supplies a profile file `prog.pf` to the global decision making and optimization step.
`-o fault`	`names the executable file fault.`
`fault.c`	the source file.
`flt_proc.c`	the fault procedures.
`asm_flt.s`	the assembly file to generate faults.
`system.c`	system file.

NOTE. *The beauty of this example is the methodology of incremental profiling, not the result of running the example.*

Compression Assisted Virtual Execution (CAVE)

This CTOOLS feature allows non-critical parts of an application's machine code to be stored in memory in compressed form resulting in reduced target memory requirements. The code is expanded into native machine code on demand for 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 of 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, 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, that 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.

Note that due to the overhead of decompressing code at runtime, only non-performance critical code should be secondary functions, such as error handling code or initialization code. You can use runtime profile information generated by gcov960 to aid in selecting the set of secondary functions.

This example uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.

For the sake of demonstration, we compress performance-critical code in the tic-tac-toe program. The purpose of this example is to show the reduced text section of the executable, not demonstrate run times.

C Example

1. Choose **Compiler**.
2. Choose **C Cave**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Use the gcc960 `mcave` option or `#pragma cave` to designate the specified functions as secondary. In the tic-tac-toe example, `ttt.c`, the following `#pragma` has been added:

   ```
   #pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)
   ```

   where `Initialize, Winner, Other, Play, Evaluate, Best_Move, Describe, Move,` and `Game` are all functions to be compressed.
5. Edit `ttt.c`. Make sure the `#pragma cave` program line is commented out:

   ```
   /*#pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)*/
   ```
6. Compile the tic-tac-toe program. Enter:

```
gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c
```

   The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies mcycx.gld.
`-o ttt`	names the executable file ttt
`ttt.c`	input file

7. Check the text section size of the uncompressed program. Enter:

   ```
   gsize960 ttt
   ```

   The option in this command is:

`ttt`	name of the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.

8. Edit `ttt.c`. Make sure the `#pragma cave` program line is uncommented:

```
#pragma cave(Initialze, Winner, Other, Play,
Evaluate, Best_Move, Describe, Move, Game)
```

9. Compile the tic-tac-toe program with the pragma program line. Enter:

   `gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c`

   The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies mcycx.gld.
`-o ttt`	names the executable file ttt
`ttt.c`	input file

10. Check the text section size of the compressed program. Enter:

   `gsize960 ttt`

   The option in this command is:

`ttt`	executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 7-4 Uncompressed Text Sections

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	33,764	32,944	32,768	32,976	31,600

Table 7-5 After Function Compression

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	31,908	30,832	30,816	30,832	29,648
Cave Section	1,818	1,770	1,746	1,800	1,776
Total	33,726	32,602	32,562	32,632	31,424

Table 7-6 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
0.1%	1.0 %	0.6 %	1.0 %	0.6 %

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

C++ Compression Assisted Virtual Execution (CAVE)

1. Choose **Compiler**.
2. Choose **C++ Cave**.
3. Choose **Make**.The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Use the *gcc960* `mcave` option or `#pragma cave` designate the specified functions as secondary. In the C++ example, `cavecpp.cpp`, the following `#pragma` has been added:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

   where `initSetName`, `initSetDept`, `initSetGpa`, `initSetNumPubs`, `isOutstanding`, `printName`, and `InitializeRecords` are all functions to be compressed, i.e., all functions are secondary functions. All other functions of the program are primary functions.

The primary set should contain performance-critical functions that are not to be affected by the CAVE mechanism; the secondary set is subject to compression. Secondary functions are compressed by the linker and reside in memory in compressed form.

The C++ compiler behaves in essentially the same manner as the C compiler when the mcave or Gcave options are used - generating all functions in the compilation unit for which this option is in effect as secondary.

A user typically designates a single function as secondary through the use of `pragma cave`. The following statement for example designates the function max as secondary.

```
# pragma cave max
```

However in C++ overloaded functions have the same name. Member functions of two different classes are also allowed to have the same name and these member functions can in turn have the same name as a function with file scope.

When a user specifies a function as secondary through the use of `pragma cave`, the C++ compiler treats all functions with this name as secondary. To illustrate, consider the following example:

```
# ifdef PRAGMA
# pragma cave max
# endif

int max(int a, int b)
{
return a > b ? a : b;
}

float max(float a, float b)
{
return a > b ? a : b;
}

class Tclass1 {
int a, b;
public:
int max();
};
```

```
int Tclass1::max()
{
return a > b ? a : b;
}

class Tclass2 {
float a, b;
public:
float max();
};


float Tclass2::max()
{
return a > b ? a : b;
}

Tclass1 t1;
Tclass2 t2;
```

The Compiler treats all the following functions as secondary.

```
int max(int, int);
float max(float, float);
int Tclass1::max();
float Tclass2::max();
```

5. Choose **Qv Code**. Edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is commented out:

   ```
   //#pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

6. Compile the C++ program. Enter:

   `gcc960 -A{`*`arch`*`} -Felf -T{`*`Link-dir`*`} -stdlibcpp -o cavecpp cavecpp.cpp`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A{`*`arch`*`}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{`*`Link-dir`*`}`	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.

`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `cavecpp`
`cavecpp.cpp`	input file

7. Check the text section size of the uncompressed program. Enter:

   ```
   gsize960 cavecpp
   ```

   The option in this command is:

`cavecpp`	specifies the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.

8. Choose **Qv Code** and edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is uncommented:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

9. Compile the C++ program with the pragma program line. Enter:

   ```
   gcc960 -A{arch} -Felf -T{Link-dir} -stdlibcpp
   -o cavecpp cavecpp.cpp
   ```

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, -AHD specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `ttt`
`cavecpp.cpp`	specifies the input file

10. Check the text section size of the compressed program. Enter:

    ```
    gsize960 cavecpp
    ```

    The option in this command is:

`cavecpp`	executable file

The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 7-7 Uncompressed Text Sections

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	89,788	84,196	83,512	84,196	81,764

Table 7-8 After Function Compression

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	87,612	81,892	81,512	81,892	79,796
Cave Section	1,920	1,546	1,514	1,546	1,512
Total	89,532	83,438	83,026	83,438	81,308

Table 7-9 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
1%	1%	1%	1%	1%

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

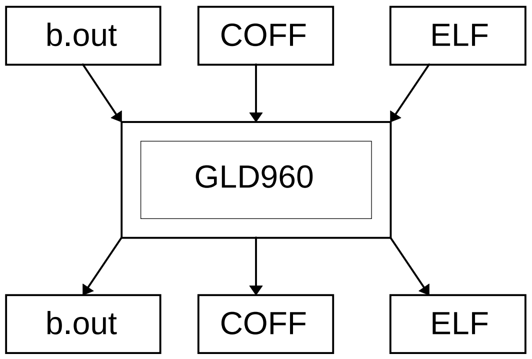
Linker Consumption

You can link b.out-format, COFF or ELF object files and libraries in any combination. To determine a file format, the linker examines the first two bytes of the file. An unrecognized value indicates a linker-directive file. This feature is useful when using third-party archives with CTOOLS runtime libraries and your application code. The runtime libraries are shipped in ELF format *only* (effective with the 5.0 version of the tools). Each can potentially have a different OMF, and the linkage still completes.

NOTE. *As of version 5.0 of the tools, all runtime libraries are shipped in ELF format only.*

If the linker generates a different output format than the input, the linker does not copy debug information from the input file to the output file. Because of this, you should use only one OMF.

The symbol tables of each OMF are abbreviated when crossing OMF boundaries. For example, when you include a b.out OMF file in a linkage where the output file OMF is COFF format, none of the debug information from the b.out file is copied into the output COFF file.



1. Choose **Linker and Utilities**.
2. Choose **Linker Consumption**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

4. Compile the first file in COFF format. Enter:

   `gcc960 -Fcoff -A`*`{arch}`*` -c t85c36.c`

   The options in this command are:

`-Fcoff`	creates a COFF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-c`	compile, but do not link.
`t85c36.c`	input file.

5. Compile the second file in ELF format. Enter:

   `gcc960 -Felf -A`*`{arch}`*` -c system.c`

   The options in this command are:

`-Felf`	creates an ELF format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-c`	compiles, but does not link.
`system.c`	input file.

6. Compile the third file in b.out format. Enter:

   `gcc960 -Fbout -A`*`{arch}`*` -c -r cyint.c int_proc.s`

   The options in this command are:

`-Fbout`	creates a b.out format output file.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-c`	compiles, but does not link.
`-r`	allows unresolved references.
`cyint.c`	the source file.
`int_proc.s`	the interrupt handler.

7. Generate an absolute file in ELF format by linking files in b.out-format, ELF format, and COFF format. The absolute file could have also been in b.out-format or COFF format. Enter:

   ```
   gld960 -Felf -T{Link-dir} -A{arch} -o elf t85c36.o
   system.o cyint.o int_proc.o
   ```

The options in this command are:

`-Felf`	specifies the absolute file as ELF format.
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tcycx` specifies `cycx.gld`.
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-o elf`	names the executable file `elf`.
`cyint.o`	file in b.out-format.
`int_proc.o`	file in b.out-format.
`t85c36.o`	file in COFF format.
`system.o`	file in ELF format.

NOTE. *The beauty of this example is the functionality of the linker, not the result of running the example.*

Assembler Pseudo-instruction Tutorial

This tutorial demonstrates the use of pseudo-instructions that have been added to the CTOOLS assembler to ease migration between processors. The tutorial that follows demonstrates how to enable and disable the instruction cache for the i960 Cx, Hx, Jx, and Rx microprocessors using microprocessor specific instructions. The tutorial then demonstrates how easy it is to enable and disable the instruction cache using only one pair of pseudo-instructions.

What are Pseudo-instructions?

A number of pseudo-instructions (pseudo-ops) have been added to the CTOOLS assembler to ease the migration between processors. These pseudo-ops provide an architecture-independent method for performing some of the more common low-level processing operations. Using these pseudo-ops should reduce the number of changes required when moving assembly code from one i960 processor to another.

When you use any of the new i960 pseudo-instructions, you are required to re-assemble your source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best processor instructions to replace the pseudo-instructions based on the processor targeted.

pseudop.c: Editing the File for the Cx Microprocessor

1. If you are using the Hx Jx Cx & Sx QUICK*val* software, choose **Linker** and **Utilities**. If using the Rx QUICK*val* software, this step is not necessary.
2. Choose **Pseudo-op Tutorial**.
3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`. Both procedures contain no code initially. `cache_off()` looks like:

   ```
   cache_off()
   {

   }
   ```
5. Add the code necessary to disable the instruction cache for the Cx microprocessor. Between the brackets of the `cache_off()` procedure, add the following line exactly:

   ```
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   ```

The `cache_off()` procedure should look like this:

```
cache_off()
{
__asm__ __volatile__("sysctl %0,%1,%2" ::"d"
(((CONFIGURE_ICACHE)<<8)|
(DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
}
```

This procedure, `cache_off()`, uses the instruction cache control processor instruction `sysctl`. This instruction is valid in the i960 Cx processor for managing and controlling the instruction cache. `sysctl` is used above to disable the instruction cache. Also, the `CONFIGURE_ICACHE` and `DISABLE_ICACHE` constants are found in the `system.h` file that is included in the `pseudop.c` file.

6. Likewise, edit the `cache_on()` procedure adding the following line exactly:

   ```
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (ENABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   ```

   The `cache_on()` procedure should look like this:

   ```
   cache_on()
   {
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (ENABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   }
   ```

   Similarly, the `cache_on()` procedure for the Cx microprocessor uses the instruction cache control processor instruction `sysctl`. `sysctl` is used directly above to enable the instruction cache.

7. Save the `pseudop.c` file.

Running pseudop.c for the Cx Microprocessor

1. Compile and run the `pseudop.c` program to show that it works as desired.

NOTE. *If you do not have an i960 Cx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

---

2. In the Command Prompt window, enter the following commands:

```
gcc960 -AC{F|A} -Fcoff -Tmcycx -o pseudop pseudop.c
```

The options in this command are:

`-AC{F|A}`	sets the target architecture for the compiler. For this example, choose the Cx architecture, `-ACF` or `-ACA`
`-Fcoff`	sets the object file type as coff.
`-Tmcycx`	sets the linker directive file for the Cx architecture.
`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

If you have a Cx microprocessor and want to run the program, enter:

```
gdb960 -t mon960 -b {baudrate} -r {comport} -pci
pseudop
```

The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.

`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

3. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works with instructions chosen specifically for the Cx architecture. Of course, this is what is expected. This program becomes more interesting when you start using pseudo-instructions.*

4. At the (gdb960) prompt, enter: **`quit`**

pseudop.c: Migrating the File to the Jx/Hx/Rx Microprocessor

Since the i960 Jx, Hx, and Rx microprocessors use the same processor instruction to enable and disable the instruction cache, this migration supports all three processors.

In order to use the program, `pseudop.c`, modified in the first part of this tutorial to support the Jx, Hx, or Rx microprocessor, it must first be migrated to those processors since they do not use the `sysctl` instruction to enable and disable the instruction cache.

1. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`.

   `cache_off()` contains the Cx specific code and looks like:

   ```
   cache_off()
   {
   __asm__ __volatile__("sysctl %0,%1,%2" ::"d"
   (((CONFIGURE_ICACHE)<<8)|
   (DISABLE_ICACHE)|((0)<<16)),"d"(0),"d"(0));
   }
   ```

2. Change the code to disable the instruction cache for the i960 Jx/Hx/Rx microprocessors. Between the brackets of the `cache_off()` procedure, delete the previously added line and insert the following line exactly:

   ```
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   ```

   The `cache_off()` procedure should now look like this:

   ```
   cache_off()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   }
   ```

   This procedure, `cache_off()`, uses the instruction cache control processor instruction `icctl`. This instruction is valid in the 80960 Jx/Hx/Rx processors for managing and controlling the instruction cache. `icctl` is used above to disable the instruction cache. Also, the `ICACHE_OFF` constant is found in the `system.h` file that is included in the `pseudop.c` file.

3. Likewise, edit the `cache_on()` procedure deleting the previously added line and inserting the following line exactly:

   ```
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_ON),"d"(0),"d"(0));
   ```

   The `cache_on()` procedure should now look like this:

   ```
   cache_on()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_ON),"d"(0),"d"(0));
   }
   ```

   Similarly, the `cache_on()` procedure for the Jx/Hx/Rx microprocessors use the instruction cache control processor instruction `icctl`. `icctl` is used directly above to enable the instruction cache.

4. Save the `pseudop.c` file.

Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors

1. Compile and run the `pseudop.c` program to show that it works as desired.

NOTE. *If you do not have an i960 Jx, Hx, or Rx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

2. In the Command Prompt window, enter the following commands:

   For the Jx Microprocessor:

   ```
   gcc960 -AJ{F|D|A} -Fcoff -Tmcyjx -o pseudop pseudop.c
   ```

   The options in this command are:

`-AJ{AF|D|T}`	sets the target architecture for the compiler. For this example, to choose the Jx architecture, `-AJA`, `-AJF`, `-AJD`, or `-AJT`
`-Fcoff`	sets the object file type as coff.
`-Tmcyjx`	sets the linker directive file for the Jx architecture.

`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

For the Hx Microprocessor:

`gcc960 -AH{D|A} -Fcoff -Tmcyhx -o pseudop pseudop.c`

The options in this command are:

`-AH{`*`D`*`|`*`A`*`}`	sets the target architecture for the compiler. For this example, to choose the Hx architecture, `-AHD` or `-AHA`
`-Fcoff`	sets the object file type as coff.
`-Tmcyhx`	sets the linker directive file for the Hx architecture.
`-o pseudop`	sets the object file name as pseudop (optional).
`pseudop.c`	specifies the input source file.

For Rx Microprocessor:

`gcc960 -AR{`*`P`*`/`*`D`*`} -Fcoff -Tmcyrx -o pseudop pseudop.c`

The options in this command are:

`-AR{`*`P`*`/`*`D`*`}`	sets the target architecture for the compiler. For this example, to choose the Rx architecture: `-ARP` or `-ARD`
`-Fcoff`	sets the object file type as coff.
`-Tmcyrx`	sets the linker directive file for the Rx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

3. If you have a Jx, Hx, or Rx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works with instructions chosen specifically for the architecture in question. Of course, this is what is expected. This program becomes more interesting when* you *start using pseudo-instructions.*

5. At the (gdb960) prompt, enter: **`quit`**

pseudop.c: Adding Pseudo-Ops to the Program

As can be seen, it is neither easy nor fun migrating code from one processor to another, especially when your code is many thousands of lines long. Fortunately, pseudo-instructions have been added to the CTOOLS assembler to ease migration between processors.

1. Choose **Qv Code**. View the file `pseudop.c` loaded into the editor. You are ready now to rewrite this program using pseudo-instructions. Scroll down the file to view the two procedures: `cache_off()` and `cache_on()`.

   `cache_off()` contains the i960 Jx/Hx/Rx microprocessor specific code:

   ```
   cache_off()
   {
   __asm__ __volatile__("icctl
   %0,%1,%2"::"d"(ICACHE_OFF),"d"(0),"d"(0));
   }
   ```

2. Change the code to disable the instruction cache for ALL processors. Between the brackets of the `cache_off()` procedure, delete the previously added line and insert the following line exactly:

   ```
       __asm__ __volatile__("ic_disable r5");
   ```

   The `cache_off()` procedure should now look like this:

   ```
   cache_off()
   {
       /* local register r5 is used to hold the status
   returned */
       __asm__ __volatile__("ic_disable r5");
   }
   ```

   This procedure, `cache_off()`, uses the pseudo-instruction `ic_disable`. When this program, `pseudop.c`, is compiled for a specific 80960 processor by using a `-A` architecture flag, the best instructions for that architecture are chosen to replace the `ic_disable` pseudo-op. Thus, pseudo-ops ease migration between processors. Also, notice only one argument to the pseudo-op is necessary. The `icctl` instruction requires three arguments. Programming with pseudo-ops can be simpler. Pseudo-instructions are also available to perform the other instruction cache management and controlling functions, such as cache invalidation.

3. Likewise, edit the `cache_on()` procedure deleting the previously added line and inserting the following line exactly:

   ```
   __asm__ __volatile__("ic_enable r5");
   ```

   The `cache_on()` procedure should now look like this:

   ```
   cache_on()
   {
         /* local register r5 is used to hold the status
   returned */
       __asm__ __volatile__("ic_enable r5");
   }
   ```

   Similarly, `cache_on()` uses a pseudo-instruction: `ic_enable`. When this program, `pseudop.c`, is compiled for a specific 80960 processor, the best instruction for that architecture is chosen to replace the `ic_enable` pseudo-op.
4. Save the `pseudop.c` file.

Running pseudop.c with Pseudo-instruction

1. Compile and run the `pseudop.c` program to show that the pseudo-instructions work as desired. To prove that the best instruction is chosen for the architecture, compile the code for the Cx microprocessor and then the Jx, Hx, or Rx microprocessor.
2. In the Command Prompt window, enter the following command:

   `gcc960 -AC{`*`F/A`*`} -Fcoff -Tmcycx -o pseudop pseudop.c`

   The options in this command are:

`-AC{`*`F/A`*`}`	sets the target architecture for the compiler. For this example, choose the Cx architecture, `-ACF` or `-ACA.`
`-Fcoff`	sets the object file type as coff.
`-Tmcycx`	sets the linker directive file for the Cx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

   When compiled, warnings may be generated. The warnings are generated just to point out this simple fact: when you use any of the new i960 pseudo-instructions, you are required to re-assemble your

source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best instructions to replace the pseudo-instructions based on the processor targeted.

3. If you have a Cx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, com3, and com4.
`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

4. At the (gdb960) prompt, enter: **`run`**

   The program prints out the performance advantage of running the program with the instruction cache enabled versus disabled.

NOTE. *The beauty of this example is not the results of the running program, but the fact that the code works as expected with pseudo-instructions.*

The result of this example is similar to using instructions specifically chosen for the Cx architecture. So, using pseudo-instructions can maintain the logic of your code, while easing migration to future i960 microprocessors.

5. At the (gdb960) prompt, enter: `quit`

NOTE. *If you do not have an i960 Jx, Hx, or Rx microprocessor, you cannot run this example; however, you can still compile the code to verify that it compiles without error.*

Running pseudop.c for the i960 Jx/Hx/Rx Microprocessors

Now you are ready to compile the code for the Jx, Hx, or Rx microprocessor to demonstrate similar results on a different processor.

1. In the Command Prompt window, enter the following commands:

   For the Jx Microprocessor:

   `gcc960 -AJ{`*`F|D|A`*`} -Fcoff -Tmcyjx -o pseudop pseudop.c`

   The options in this command are:

`-AJ{`*`A|F|D|T`*`}`	sets the target architecture for the compiler. For this example, to choose the Jx architecture, `-AJA`, `-AJF`, `-AJD`, or `-AJT`
`-Fcoff`	sets the object file type as coff.
`-Tmcyjx`	sets the linker directive file for the Jx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

   For the Hx Microprocessor:

   `gcc960 -AH{`*`D|A`*`} -Fcoff -Tmcyhx -o pseudop pseudop.c`

   The options in this command are:

`-AH{`*`D|A`*`}`	sets the target architecture for the compiler. For this example, to choose the Hx architecture, `-AHD` or `-AHA`
`-Fcoff`	sets the object file type as coff.

`-Tmcyhx`	sets the linker directive file for the Hx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

For Rx Microprocessor:

`gcc960 -AR{`*`P|D`*`} -Fcoff -Tmcyrx -o pseudop pseudop.c`

The options in this command are:

`-AR{`*`P|D`*`}`	sets the target architecture for the compiler. For this example, to choose the Rx architecture, `-ARP` or `-ARD`
`-Fcoff`	sets the object file type as coff.
`-Tmcyrx`	sets the linker directive file for the Rx architecture.
`-o pseudop`	sets the object file name as `pseudop` (optional).
`pseudop.c`	specifies the input source file.

2. If you have a Jx, Hx, or Rx microprocessor and want to run the program, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -pci pseudop`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional).
`-b 115200`	sets the baud rate for serial communication (optional). This option is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication. This option is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, com3, and com4.

`-pci`	sets the code download option for the PCI bus. When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`pseudop`	specifies the executable file.

3. At the (gdb960) prompt, enter: **`run`**

   The result of this example is the same as using instructions specifically chosen for the Jx, Hx, or Rx architecture. So, using pseudo-instructions does not change the logic of the program. It only eases future migration of your code to future i960 microprocessors.
4. At the (gdb960) prompt, enter: **`quit`**

   When compiled, warnings may be generated. The warnings are generated just to point out this simple fact: When you use any of the new i960 pseudo-instructions, you are required to re-assemble your source code before running it on a new target platform (e.g., from Cx to Jx). The assembler selects the best instructions to replace the pseudo-instructions based on the processor targeted.

CONGRATULATIONS! You can now start using pseudo-instructions in your code to ease migration of your code to future i960 processors.

Debugging with gdb960

A software debugger is a useful tool that allows you to learn more about the behavior of an application program while it is running on a target or simulator. gdb960 is a source-level debugger that allows you to interact with your application program running on a target system through the debug monitor, MON960. MON960 is resident on the Cyclone CPU module.

This example uses the card game, Go Fish, and is designed to teach you a few debugger commands so that you can further examine the example programs provided with this kit or your own programs. In the card game, Go Fish, you and the computer each get several cards. You take turns guessing which cards are in each other’s hands. When you guess correctly, you acquire that card. If you don’t guess correctly, you need to “Go Fish” and draw another card from the pack. When you get four-of-a-kind, you

remove those cards from your hand. The objective of the game is to have the most sets of four-of-a-kind when either you or the computer has no cards remaining in your hands.

NOTE. *This example uses the command line interface to gdb960. The program also features a Graphical User Interface in both Windows and UNIX. See The gdb960 User's Manual for more information.*

1. Choose **Debugger**.
2. Choose **gdb960 Tutorial**.
3. Choose **Make** to compile, link, and download the program automatically.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

NOTE. *DEBUGGING SHORTCUTS*
Abbreviations for gdb960 commands are accepted as long as they are unambiguous.
*To **run,** enter: **r***
*To **break,** enter: **br***
*To **list,** enter: **l***
*To **continue,** enter: **c***
*To **print,** enter: **p***
*To **clear,** enter: **cl***
*To **quit,** enter: **qu***
*For **help,** enter: **he***

4. **Do Not Type Run!** First, use the gdb960 debugger to set a breakpoint at function `main()`. Type:

   `break main`

   The debugger responds by displaying:

```
Breakpoint 1 set at 0xa0008570: file fish.c, line 209.
```

5. Set a second breakpoint at line 275. Type:

   `break 275`

   The debugger responds by displaying:

```
Breakpoint 2 set at 0xa0008bc4: file fish.c, line 275.
```

6. To execute the program from the beginning, type:

   `run`

   The debugger responds by displaying:

```
Starting program: C:\QUICKVAL/fish
Breakpoint 1, main() at fish.c, 209.
209    srand();
```

7. To display the code at the breakpoint, type:

   `list`

   The debugger displays lines 204-213 of the fish.c source. To see the next ten lines, type `list` again.

8. To continue executing the program from this location, type:

   `continue`

   The debugger responds by displaying:

```
Continue.
Would you like instructions[n]?
```

9. Reply by typing **`y`** for yes or <Enter> or **`n`** for no.

```
your hand is: A A 6 6 8 8 9
Breakpoint 2, game() at fish.c:275.
275    if(!move(yourhand,myhand,g=guess(),0))break;
```

10. In the source code in step 9, there are two variable arrays, `myhand` and `yourhand`. `Myhand` is the computer’s hand and `yourhand` is yours. To look at the card in the computer’s hand, type:

    `print myhand`

The debugger responds by displaying:

```
$1="000\000\000\001\000\002\000\001\000\000\001\002\000"
```

`myhand[0]` does not represent a card.

`myhand[1]` represents the number of Aces.

`myhand[2]` represents the number of 2s, and so on.

The same order of cards is represented in the array, `yourhand`.

If a King is drawn by either player, `myhand[13]` or `yourhand[13]` will appear when you print the array.

11. Using the ability to see the computer's hand, you are able to beat the computer every time. Clear the first breakpoint at the function `main()` and continue playing the game, looking at the computer's hand any time you need to. To clear the breakpoint at `main()`, type:

    `clear main`

    The debugger responds by displaying:

    ```
    Deleted breakpoint 1
    ```

12. To continue executing the program, type:

    `continue`

13. If you need further assistance beating the computer, contact the 80960 Technical Support Group for more hints.
14. Type: **`quit`**

Debugging Optimized Code

CTOOLS can use the ELF object module format and DWARF Version 2 debug information format as described in the *80960 Embedded Application Binary Interface (ABI) Specification* (order number 631999). The new formats enable more accurate mapping between source and object code at higher optimization levels and ease production code debugging.

This example shows that at the highest level of module-local optimization, it is possible to set a breakpoint on an inline function using ELF/DWARF, while with COFF this is not possible.

1. Choose **Debugger**.
2. Choose **C ELF/DWARF Format**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

4. Compile *swap.c* with no module-local optimizations (no inlining). This shows that the procedure *swap* is not inlined. Enter:

   `gcc960 -Felf -T`*`{Link-dir}`*` -A`*`{arch}`*` -O0 -S swap.c`

   The options in this command are:

`-Felf`	creates an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-O0`	no module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

5. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, see the call to the procedure swap:

   ```
   callj _swap
   ```

   This is an out-of-line call to the procedure swap. The function swap has not been inlined.

6. Now, compile `swap.c` with the highest level of module-local optimizations. This inlines the procedure `swap`.

   `gcc960 -Felf -T`*`{Link-dir}`*` -A`*`{arch}`*` -O4 -S swap.c`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-O4`	highest level of module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

7. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, note the call to the procedure `swap` does not exist:

   ```
   callj _swap  /* Does Not Exist*/
   ```

   The procedure swap has been inlined.

8. Recompile using the `-O4` optimization level, the ELF/DWARF format, and add debugging information.

```
gcc960 -Felf -T{Link-dir} -A{arch} -O4 -g -o swap swap.c
```

The options in this command are:

`-Felf`	create an ELF format output file
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcycx` specifies mcycx.gld.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o swap`	names the executable file swap
`swap.c`	input file

9. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

```
gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap
```

The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`swap`	the executable file

10. **Do Not Type Run!**

First, set a breakpoint on the procedure *swap.* Enter:

```
break swap
```

The debugger responds by displaying:

```
breakpoint 1 @0xa00080f0:file swap.c, line 43
breakpoint 2 @0xa0008148:file swap.c, line 54
```

Breakpoint 1 is the out-of-line reference to the procedure `swap`. Breakpoint 2 is the inline reference to the procedure `swap`.

`Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is still possible to set a breakpoint on the inline function. Breakpoint 2 stops program execution.

11. To execute the program, enter:

    `run`

    The debugger responds by displaying:

    ```
    Breakpoint 2, main() @ swap.c: 54
    54 printf(ìThe smallest number is %d\nî,a);
    ```

12. To continue the program, enter:

    `c`

    When the program has finished, enter:

    `quit`

13. Compile using the `-O4` optimization level, the COFF format, and add debugging information.

```
gcc960 -Fcoff -T{Link-dir} -A{arch} -g -O4 -o swap swap.c
```

The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o swap`	names the executable file `swap`
`swap.c`	input file

14. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

    `gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap`

    The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`swap`	the executable file

15. **Do Not Type Run!!**

    First, set a breakpoint on the procedure `swap`. Enter:

    `break swap`

    The debugger responds by displaying:

    ```
    breakpoint 1 @0xa00080f0
    ```

    Breakpoint 1 is the out-of-line reference to the procedure `swap`. Notice that no inline breakpoint has been set. This breakpoint does not stop execution of the program.

    `Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is not possible to set a breakpoint on the inline function. Program execution does not stop.

16. To execute the program, enter:

    `run`

    The debugger responds by displaying the smallest number from the swap. There is no break in program execution.

17. When the program has finished, enter:

    `quit`

    You have now seen that with the ELF/DWARF format, it is now possible to debug your production code, even after high levels of program optimization.

Debugging Optimized C++ Code Tutorial

The C++ compiler generates debug information using the DWARF format when the `-g` option is specified with the `-Felf` option. This debug information format is richer than that of other supported OMFs, and allows more reliable debugging under optimization.

This tutorial demonstrates that at the highest level of module-local optimization, debugging a C++ application is still possible due to the DWARF debug format.

In this example, you compile a C++ program using the `-O0` optimization compiler option, which disables all optimizations, including those that may interfere with debugging. The same C++ program is then compiled using the highest-level of module-local optimization, `-O4`.

There are several levels of program optimization available with the CTOOLS development tool suite. Typically, low levels of optimization are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once the application is functioning properly, the application's performance may be increased by using a higher level of optimization. The static optimization options are:

`O0`	Turn optimization off
`O1`	Basic optimization
`O2`	strength-reduction, instruction scheduling for pipelining, etc...
`O3`	`O2` plus `fconstprop, finline-functions`, etc...
`O4`	`O3` plus `fsplit-mem, fmarry-mem, fcoalesce`

Level O4 is the highest level of static optimization. Please refer to the *i960 Processor Compiler User's Guide* for more information on ELF/DWARF and compiler optimizations.

In this tutorial, you compile and debug a C++ program, `cppdwarf.cpp`, that contains many of the advanced features of the C++ language, including:

- Classes
- Public, protected, and private variable accessibility
- Virtual functions
- Scope operators
- Overloaded functions
- Class inheritance

Using ELF/DWARF, both levels of optimization, `-O0` and `-O4`, retain the C++ program structure so that the above features may be investigated.

1. Choose **Debugger**
2. Choose **C++ ELF/DWARF Format**
3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Compile the program using the `-O0` optimization level. In the Command Prompt window, enter the following command:

```
gcc960 -Felf -A{arch} -T{Link-dir} -stdlibcpp -O0 -g
-o cppdwarf cppdwarf.cpp
```

The options in this command are:

`-Felf`	creates an ELF format output file.
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD.
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcycx` specifies mcycx.gld.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-O0`	specifies the lowest level of module-local optimizations.
`-g`	includes debug information in object file.
`-o cppdwarf`	specifies the executable file `cppdwarf`.
`cppdwarf.cpp`	specifies the input file `cppdwarf.cpp`.

5. Run the program using the debugger, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -D {`*`parallel port`*`} -pci cppdwarf`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, ... com99.
`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are lpt1 and lpt2.
`-pci`	sets the code download option for the PCI bus (optional). When no serial port is specified, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`cppdwarf`	specifies the executable file `cppdwarf`.

6. **Do Not Enter Run!**

   Now you are ready to examine some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes.

At the (gdb960) prompt, enter:

`ptype person`

The following data type information concerning the class `person` appears:

Example 7-1 person Class

```
  type = class person {
    protected:
      char name[40];
      char dept[40];
    public:
      void setName ();
      void setName (char *);
      void setDept ();
      void setDept (char *);
      void printName ();
      virtual int isOutstanding ();
      virtual char * getDept ();
}
```

Please note the following concerning the above output:

- The entire class information for person is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.

Another C++ class in the program is `professor`, which inherits from the person class. Again, you use the gdb960 command `ptype` to display a description of the `professor` class.

7. At the (gdb960) prompt, enter:

   `ptype professor`

The following data type information concerning the class `professor` appears:

Example 7-2 professor Class

```
type = class professor : public person {
  private:
    int numPubs;
  public:
    void setNumPubs ();
    void setNumPubs (int);
    virtual int isOutstanding ();
}
```

Please note the following concerning the above output:

- The entire class information for `professor` is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.
- `type = class professor : public person` indicates that the `professor` class inherits from the `person` class.

8. You are ready to set some breakpoints.
    a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

       `break professor::setNumPubs`

       The following information concerning breakpoints is displayed:

       ```
       [0] cancel
       [1] all
       [2] professor::setNumPubs(int) at
       cppdwarf.cpp:125
       [3] professor::setNumPubs(void) at
       cppdwarf.cpp:118
       ```

Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.

b. Set a breakpoint on all `professor::setNumPubs` functions. At the `>` prompt, enter: **`1`**

   The following information about breakpoints is displayed:

   ```
   Breakpoint 1 at 0xa00083d0: file cppdwarf.cpp,
   line 125.
   ```

   ```
   Breakpoint 2 at 0xa0008358: file cppdwarf.cpp,
   line 118.
   ```

c. Set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

   `break professor::isOutstanding`

   The following information concerning breakpoints is displayed:

   ```
   Breakpoint 3 at 0xa0009080: file cppdwarf.cpp,
   line 110.
   ```

9. You are now ready to start the program. At the (gdb960) prompt, enter:

   `run`

   Notice that the program stops at all three of the breakpoints.

10. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.

11. At the (gdb960) prompt, enter: **`quit`**

    The results of the debug session were as expected because no optimizations had been performed on the source code during compilation. You can now recompile the `cppdwarf.cpp` program using the highest-level of module-local optimization and repeat the previous debug session.

12. Compile the program using the `-O4` optimization level. In the Command Prompt window, enter the following command:

    `gcc960 -Felf -A{`*`arch`*`}-T{`*`Link-dir`*`} -stdlibcpp -O4 -g -o cppdwarf cppdwarf.cpp`

    The options in this command are:

`-Felf`	create an ELF format output file
`-A{`*`arch`*`}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{`*`Link-dir`*`}`	specifies the linker directive file. For example, `-Tmcycx` specifies `mcycx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o cppdwarf`	specifies the executable file `cppdwarf`
`cppdwarf.cpp`	input file

13. Run the program using the debugger, enter:

    `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -D {`*`parallel port`*`} -pci cppdwarf`

    The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, ... com99.

`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are: lpt1 and lpt2.
`-pci`	sets the code download option for the PCI bus (optional). When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used.)
`cppdwarf`	specifies the executable file.

14. **Do Not Enter Run!**

    You are now ready to investigate some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes. At the (gdb960) prompt, enter:

    `ptype person`

    Please note, the output matches that of Example 7-1, "person Class". Optimizations did not affect the `person` class output. It is the same as the first debug session.

15. Another C++ class in the program is `professor`, which inherits from the person class. Once again, you use the gdb960 command `ptype` to display a description of the `professor` class. At the (gdb960) prompt, enter:

    `ptype professor`

    Again please note, the output matches that of Example 7-2, "professor Class". Optimizations did not affect the `professor` class output. It is the same as the first debug session.

16. You are now ready to set some breakpoints.
    a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

       `break professor::setNumPubs`

       The following information concerning breakpoints is displayed:

       ```
       [0] cancel
       [1] all
       [2] professor::setNumPubs(int) at
       cppdwarf.cpp:125
       [3] professor::setNumPubs(void) at
       cppdwarf.cpp:118
       ```

       Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 only sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.
    b. Set a breakpoint on all `professor::setNumPubs` functions, so At the `>` prompt, enter: **`1`**.

       The following information about breakpoints is displayed:

       ```
       Breakpoint 1 at 0xa00082e4: file cppdwarf.cpp,
       line 125.
       Breakpoint 2 at 0xa0008294: file cppdwarf.cpp,
       line 118.
       ```
    c. Finally, set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

       `break professor::isOutstanding`

       The following information concerning breakpoints is displayed:

       ```
       Breakpoint 3 at 0xa0008960: file cppdwarf.cpp,
       line 111.
       ```

17. You are now ready to start the program. At the (gdb960) prompt, enter:
    `run`
    Notice that the program does not stop at all three of the breakpoints. As can be seen, the DWARF debug information format is very rich, 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.
18. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.
19. At the (gdb960) prompt, enter: **`quit`**

CONGRATULATIONS! You may now know how to use ELF/DWARF to debug your optimized C++ code.

Writing Flash

NOTE. *In order to write to flash on your Cyclone base board, you need a 12 volt power supply. Also, these instructions are used with the CTOOLS 6.0 and MON960 3.2.3 toolsets.*

This example teaches you the following:

- Writing to flash on the Cyclone base board.
- Booting off of the flash in socket U27 of the Cyclone base board, as opposed to the flash on the CPU Module.
- Setting the Cyclone base board to 12 volts.
- Using *mondb.exe* as a simple utility to download and execute an application program on the target board running MON960.
- Using *mondb.exe* to write flash.
- Building a new monitor for a particular i960 microprocessor family member.
- Retargeting MON960 for other boards.

Complete this step:

1. Choose **MON960**.
2. Choose **Writing Flash.**
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Identify the Flash on the Cyclone base board.

   A blank Flash chip ships on each Cyclone base board in socket U22. To write MON960 to Flash, you must move the blank Flash from socket U22 to socket U27.
5. Set the Cyclone base board voltage to 12 volts.

   Locate the four-position DIP switch labeled S1. Flip S1.1 to the *ON* position. This enables VPP to the Cyclone base board Flash.
6. Power up the Cyclone eval base board

   Locate the four-pin connector that interfaces to a secondary power supply labeled J6. Three of the connector pins connect to +5 VDC, +12 VDC and ground. (On the PCI-SDK Platform, +12 VDC and +5 VDC power is supplied through the edge connector.)
7. Edit `Version.c`.

   a. Change directories to where the `version.c` file resides. The default installation directory for CTOOLS is:

      `c:\intel960\src\mon960\common`

      If you cannot find the mon960 directory, You need to install MON960 as directed in the *MON960 Debug Monitor User's Manual*.

      Version.c contains the following information:

```
const char mon_version_byte =  nn;   /* version n.n = nn */
const char base_version[] = "MON960 n.n.n";
const char build_date[] = __DATE__;
```

   b. Change the file contents to reflect that this is your version of MON960. For example, change

```
const char base_version[] = "MON960 n.n.n";
```

      to:

```
const char base_version[] = "MY MON960";
```

   c. Save `Version.c`.

8. Build the new MON960 from source (optional)

   By default the source for MON960 is located at:
   `c:\intel960\src\mon960\common`

   You may use the pre-built version of MON960 there, or build a custom verion. To create a custom version:

   a. Copy `makefile.xxx` to
      `c:\intel960\src\mon960\common\makefile`.

      where xxx is one of the following make files:

`makefile.ic`	(ic960 interface, COFF format)
`makefile.ie`	(ic960 interface, ELF format)
`makefile.gc`	(gcc960 interface, COFF format)
`makefile.ge`	(gcc960 interface, ELF format)

   b. Issue the commands:

      ```
      nmake -s makefile
      cycx
      ```

      This creates a file called `cycx.fls`.

9. Write the Flash

   To write the Flash, use the mondb.exe utility located in the `intel960\bin\` directory. If you are going to use the pre-built MON960 files, they are located in the `intel960\roms` directory. For example, if you used the default installation directory and are using the pre-built MON960 files for the 80960Cx, enter:

   ```
   mondb -ser com1 -par lpt1 -ef -ne
   c:\intel960\roms\cycx.fls
   ```

   The options in this command are:

`-ser com1`	use serial port 1
`-par lpt1`	use parallel port 1
`-ne`	no execute
`-ef`	erase Flash
`cycx.fls`	input Flash filename

   Note also that if you built a version of MON960 from the source code as described previously, the `cycx.fls` file will be located in the `c:\intel960\src\mon960\common\` directory.

10. Set Board Voltage Back To +5 VDC

    Locate the four-position DIP switch labeled S1. Set S1.1 to the *OFF* position. This disables VPP to Cyclone EP base board Flash and protects the Flash. Note that the PCI80960DP and i960 `Cx` evaluation platforms do not boot when VPP is enabled and MON960 is running from the evaluation board Flash.
11. Set board to boot from U27 socket

    Locate the four-position DIP switch labeled S1. Set S1.3 ROMSWAP to the *ON* position. This exchanges the addresses of the CPU Module ROM and the base board ROMs. When the switch is *OFF* the processor boots from the CPU Module ROM; when the switch is *ON* the processor boots from the base board ROMs.
12. Reset Base Board

    Locate the reset pushbutton labeled S2. Use this button to manually reset the Cyclone base board and boot from the base board ROMs.

NOTE. *If you have trouble with this example, refer to Chapter 3 for troubleshooting tips.*

How to Add Benchmarking Routines to Your Code

Benchmarking is a common way to evaluate an architecture for its performance. CTOOLS comes with two routines for benchmarking code. These routines are called `bentime()` and `init_bentime()`. `init_bentime()` is called once to program the on-board Counter/Timer to periodically interrupt the processor. The `bentime()` routine returns the time in microseconds based on the count from the interrupt handler, `timer_isr`, and the current count read from the timer. By placing a call to `bentime()` at the start and end of the code you are timing, the elapsed time can be calculated by the difference between the second call to `bentime()` and the first.

1. Choose **Benchmarking**.
2. Choose **Qv Code**.

3. Scroll through the `chksum.c` code for comments that refer to “Benchmarking Routine”. You can add similar lines to the code that you want to time.
4. Choose **Make** to compile, link, and download the program automatically.
5. Execute the `chksum` program. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 2.609590 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**

Other i960 Processor Choices and the Remote Evaluation Facility

The i960 RISC processor family has a wide breadth of processors to match your design’s price and performance needs. If you wish to evaluate other i960 processor family members, contact your local distributor and order different Cyclone CPU modules, or visit the Remote Evaluation Facility at `http://developer.intel.com/design/i960/testcntr`

NOTE. *The i960 Rx Processor is not available through the Remote Evaluation Facility.*

If you choose to order more CPU modules, you may rest assured that all i960 processor modules plug-n-play with your QUICK*val* kit. This configuration was specifically designed to protect your investment and offer a low cost migration path for future needs.

The i960 Sx CPU Example Programs 8

With a full 32-bit internal architecture and a 16-bit external bus, the i960 SA/SB embedded processors are faster than all other 16-bit processors on the market. Although the powerful i960 SA/SB components are actually the low-end members of the i960 microprocessor family, these high-speed processors are ideal for today's more demanding applications, such as entry-level page printers, I/O controllers, games and communications products. The integrated burst control reduces bus bottlenecks. The powerful 32-bit CPU increases throughput. And the high level of integration minimizes chip count to lower system costs.

The i960 SB processor is pin-compatible with the i960 SA processor, and integrates an IEEE 754 compatible floating point unit. The i960 SA/SB microprocessors are object code-compatible with all of the i960 microprocessor family members, including the mid-range i960 KA/KB processors, the superscalar i960 CA/CF processors, and the military i960 MC processor.

Additionally, you can optimize your system's performance with CTOOLS, which includes a profile-driven compiler that can automatically optimize your code based on its runtime behavior.

The following pages describe the example programs included with this kit. Each example highlights a feature of the architecture or CTOOLS and provides you with source code that can help shorten your software development cycle. Table 8-1 provides descriptions of the tutorials included in the i960 Sx QUICK*val* kit.

Table 8-1 QUICK*val* i960 Processor Sample Programs

Tutorial Description	Source Files
Hello World: Uses simple printf statement to verify system integrity.	`hello.c`: source file `system.c`: system file
Memory Test: Used for system verification of external memory. The programs perform byte, short, or word writes to external memory, and then they check the addresses written for correctness.	`memtst8.c:` 8 bit memory test `memtst16.c`: 16 bit memory test `memtst32.c`: 32 bit memory test `system.c`: system file
External Interrupts: Shows how to configure the Cyclone board timers to trigger hardware interrupts. This is also an example of using interrupt handlers and placing the handlers in the interrupt table.	`cyint.c`: source file `asm_fns.s`: interrupt handler for Sx `int_proc.s`: interrupt handler-all processors but Sx `t85c36.c`: eval board timer file `system.c`: system file
C Local Optimizations: Shows how to use the C compiler with high levels of static optimization for improved runtime performance.	`chksum.c`, `system.c`: source files
C Global Optimizations: Shows how to use program-wide optimizations of the C compiler for increased performance.	`chksum.c`, `system.c`: source files
C++ Local Optimizations: Shows how to use the C++ compiler with high levels of static optimization for improved runtime performance.	`optimize.cpp`: source file
C++ Global Optimizations: Shows how to use program-wide optimizations of the C++ compiler for increased performance.	`optimize.cpp`: source file
C++ Virtual Function Optimizations: Shows how a call to a virtual function can be replaced by a direct call to a member function, and, if possible, it may be inlined at the call site. This improves the runtime performance of the code.	`optimize.cpp`: source file
Profiling Lab: Teaches you how to use some of CTOOLS advanced profiling features.	`chksum.c`: source file

continued ☛

Table 8-1 QUICK*val* i960 Processor Sample Programs (continued)

Tutorial Description	Source Files
Self-Contained Profile: Shows how to create a self-contained profile that captures the program structure and associates it with the program counters from a raw profile. When the source program changes, the global decision making step interpolates or stretches the counters in the self-contained profile to fit the changed program.	`quick.c`: source file
C Cave: Uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.	`ttt.c`: source file
C++ Cave: Shows how to reduce target memory requirements. The text sections of compressed and uncompressed C++ executables are compared. This example also shows how to specify functions for compression.	`cavecpp.cpp`: source file
Linker Directive Language: Provides a hyperlinked manual that describes the linker command options. This tutorial is found in the online help only, not in this manual.	
Debugging with gdb960: Uses the Go Fish card game to teach a few useful debugger commands.	`fish.c`: source file `system.c`: system file
ELF/DWARF Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to set a breakpoint on an in-line function.	`swap.c`: source file
C++ DWARF-2 Debugging Format: Demonstrates that at the highest level of module-local optimization, it is possible to debug a C++ application.	`cppdwarf.cpp`: source file
Retargeting MON960: Provides steps for retargeting MON960. This tutorial is found in the online help only, not in this manual.	

Table 8-1 QUICK*val* i960 Processor Sample Programs (continued)

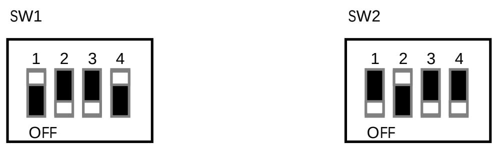
Tutorial Description	Source Files
Writing Flash: Demonstrates how to update the version of MON960 on your evaluation board.	
80960 Family Benchmark: Shows how to use this facility to compare your processor's performance with other i960 family members. This example uses a typical checksum routine to show how to add benchmarking routines into source code.	`chksum.c`, `system.c`: source files
Remote Evaluation Facility: Guides you through the use of this new benchmarking facility on the World-Wide Web.	

System Validation

Hello World

The program `hello.c` is used to verify your software and hardware system integrity. The following steps provide instructions on how to compile, link, download, and execute this program.

1. Verify that your software and hardware have been installed according to the instructions in Chapter 2 through 3 and the frequency switch on your CPU module is set as shown.
   - SW1 Set for Serial Port UART PLX PCI 9060.
   - SW2 Set for 20Mhz Sx CPU module frequency.



1. Power your Cyclone evaluation platform and i960 Sx CPU module
2. Double-click on the **Hx Jx Cx & Sx QUICK*val*** icon in the QUICK*val* program group.

3. Configure you hardware.
   - Select the **80960 Architecture** tab
   - Select **Sx**.
   - Depending on the board you have installed, select either the EP80960BB or PCI80960DP tab.
   - Configure the software communication options to match those of your evaluation board.
   - Choose **OK**
4. Choose **Hello World**.
5. Choose **Make** to compile, link, and download the program automatically.
6. Use the gdb960 debugger to execute hello. Type:

   `run`
7. The gdb960 debugger responds by displaying:

```
Hello...Welcome to the 80960SX QUICKval Kit!
SYSTEM CHECK COMPLETED!!
Now you may proceed with our Example Programs.
Program Exit: 01
(gdb960)
```

8. To exit the debugger, type: **`quit`**

CONGRATULATIONS! You have successfully installed your software and your hardware, compiled a program using gcc960, and downloaded and executed the program on your evaluation board using the gdb960 debugger.

If you received any error messages during this process, refer to "If Something Goes Wrong" on page 8-7.

Memory Test

The programs `memtst8.c`, `memtst16.c`, and `memtst32.c` are used to test the external memory on the Cyclone base board.

Depending on the test that is run, an 8, 16, or 32-bit test is run on an area of memory. The program writes F's and 0's to a memory location and reads the location to verify the integrity of what was written. All three programs are almost identical, with the exception of the casting of the variable *ADDR, which allows you to perform different test types.

NOTE. *Below,* `memtst*.c` *refers to either the byte, short, or word memory test example.*

1. Choose **Memory Test**.
2. Choose a memory test. The options are, **8-bit Memory Test**, **16-bit Memory Test**, or **32-bit Memory Test**.
3. Choose **Make** to compile, link, and download the program automatically.
4. Use the gdb960 debugger to execute memtst. Type:
   `run`
5. For the 8-bit test, `memtst8.c`, the gdb960 debugger responds by displaying:

```
This program will run a 8-bit test on the external memory.

Test to be implemented is byte test.
Starting address = a000dfb0
Ending address = a000ec30

Press enter to begin test with 0's.
Number of errors that occurred is 0.

Begin test for f's.

Press enter to continue.
Number of errors that occurred is 0.

All tests are complete.
Program exited with code 030.
(gdb960)
```

6. Exit the debugger. Type:
   `quit`

If Something Goes Wrong

The following section describes a few actions that may help resolve errors that may have occurred when invoking one of the tools. If you were unable to get the proper response from the gdb960 debugger after executing the above programs and the trouble-shooting hints described below do not help, contact the 80960 Technical Support Group by phone at 1-800-628-8686 or by E-mail at 960tools@intel.com.

MON960 Debug Monitor is Not Responding...

If the red FAIL LED (CR6) on the base board is lit, the monitor may not have booted up correctly. Press the reset button (S2). If the red FAIL LED remains lit, contact the 80960 Technical Support Group.

Invoking the gcc960 Compiler Resulted in Errors...

The environment must be set-up as described in Chapter 2. If you chose the default directories while installing CTOOLS, verify that the path names `C:\INTEL960\BIN` have been added to your PATH variable and that the following statement is in your `autoexec.bat` file. If you did not install these tools using the default directories, make the appropriate change.

```
SET G960BASE=C:\INTEL960
```

NOTE. *You did not use the default directories on installation, please make sure the G960BASE environment variable is assigned appropriately.*

NOTE. *Don't forget to re-boot your system once you have made any necessary changes to your* `autoexec.bat` *file.*

Invoking the gld960 Linker Resulted in Errors...

Verify that the directory that contains the `hello.c` and `memtst*.c` example programs also now has the object files, `hello.o` and `memtst*.o`. If `hello.o` and `memtst*.o` do not exist, then the gcc960 compiler command did not successfully create an object file. Re-compile `hello.c` and `memtst*.c` to see if an error occurred during the compilation.

If `hello.o` and `memtst*.o` do not exist, make note of the error message and contact the 80960 Technical Support Group.

Invoking the gdb960 Debugger Resulted in Errors...

NOTE. *If you are using the PCI-SDK evaluation platform, you may specify* `-pci` *for PCI download and PCI communication.*
For a list of all the gdb960 command line options, at a command prompt, enter: **`gdb960 -h | more`**

Serial communication error

A serial communication error causes the gdb960 debugger to respond by displaying:

```
HDIL error (10), communication failure
HDIL error (10), communication failure
You can’t do that when your target is ‘exec’
```

Verify that the serial port you are using is the one you specified in the gdb960 command line. Verify that your serial cable is properly connected to the board and to your PC.

Parallel communication error

A parallel communication error causes the gdb960 debugger to respond by displaying:

```
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type ‘show copying’ to see the conditions.
There is absolutely no warranty for GDB; type “show warranty” for
details.
gdb960.exe 6.0, Wed FEB 16 12:33:16 1998
GDB 5.10 (i486-intel-dos --target i960-intel-mon960), Copyright 1997
Free Software Foundation, Inc...(no debugging symbols found)...
Connected to com1 at 115200 bps.
(gdb960)
section 0, name .text, address 0xc0008000, size 0x50ec, flags 0x20
            writing section at 0xc0008000
```

Verify that the parallel port you are using is the one you specified in the gdb960 command line. Verify that your parallel cable is properly connected to the board and to your PC.

NOTE. *Your actual run times may vary.*

External Interrupts Tutorial

The purpose of this program, `cyint.c`, is to show the steps required when dealing with an interrupt triggered externally by the evaluation board timers. The `cyint.c` source code contains step-by-step instructions to save you time when you program interrupts for your application. `asm_fns.sam` is the interrupt handler, and `t85c36.c` contains the functions to program the evaluation board timers.

The example performs the following steps in the handling of a hardware interrupt.

- Modify the ICON register
- Modify the IMAP register
- Cache the interrupt vector and the interrupt handling procedure

- Lower the processor priority
- Modify the IMSK register
- Clear the IPND register
- Generate the hardware interrupt using the evaluation board timers

Complete these steps:

1. Choose **Interrupt Examps**.
2. Choose **External Interrupts**.
3. Choose **Qv Code**.
4. Scroll through the `cyint.c` source to see the code for setting up and handling a hardware interrupt triggered by the evaluation board timers.
5. Open and scroll through the `t85c36.c` and `t85c36.h` files to see the definitions and routines for programming the evaluation board timers. You can simplify the programming of the evaluation board timers by including this code in your own applications.
6. Choose **Make** to compile, link, and download the program automatically.
7. Use the gdb960 debugger to execute `cyint`. Type:

   `run`

   The debugger responds by displaying:

```
interrupt count = 314
    interrupt count = 328
    interrupt count = 343
    interrupt count = 358
    interrupt count = 373
    interrupt count = 388
    interrupt count = 403
    interrupt count = 418
    interrupt count = 432
    Program exited with code 020.
    (gdb960)
```

8. Type: **`quit`**

NOTE. *Your actual interrupt counts may vary.*

Static, Global, and Profile-Driven Optimizations

Optimizing compilers provide you with a means of developing high performance code without detailed knowledge of the architecture. Engineers who understand the features of the i960 architecture developed gcc960 to provide optimizations that take full advantage of the i960 processor. In general, optimizing compilation takes more time and may require more memory for large functions. However, the benefit in runtime performance is well worth it.

There are several levels of optimization available. Typically, low levels of optimizations are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once your application is functioning properly, you can increase its runtime performance by using a higher level of optimization.

Release 5.0 and later of the development tools support the ELF object module format and DWARF version 2.0 debug information format. The new format enables more accurate mapping between source and object code at higher optimization levels and ease debugging of production code.

The C optimization example uses a program called `chksum.c`. The C++ examples use a program called `optimize.cpp`

C No Optimization

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.
4. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.

5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 81.141323 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**

C Static Optimization

Use the following commands to compile the `chksum.c` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C Local Optimizations**.
4. Choose **Make -O4** to compile with optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 9.874086 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ No Optimization

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.

5. Choose **Make -O0** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 42.0705 seconds.
   Program exited normally
   ```
7. Type: **`quit`**

C++ Static Optimization

Use the following commands to compile the `optimize.cpp` program using the highest level of optimization without using runtime behavior, or program-wide optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Local Optimizations**.
5. Choose **Make -O4** to compile without optimizations, link, and download the program automatically.
6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 31.2296 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

C Global Optimization

Use the following commands to compile the `chksum.c` with program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C Global Optimizations**.
4. Choose **Make +O5** to compile with optimizations, link, and download the program automatically.
5. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 7.254407 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**
7. Choose **Results**.

C++ Global Optimization

Use the following commands to compile the `optimize.cpp` program using the program program-wide optimizations, which are sophisticated, inter-module optimizations.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Global Optimizations**.
5. Choose **Make+05** to compile with optimizations, link, and download the program automatically.

6. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 28.0329 seconds.
   Program exited normally
   ```
7. Type: **`quit`**
8. Choose **Results**.

Instrumentation, Profile Creation, Decision-making, and Profile-Driven Re-Compilation

An 91% improvement in C code performance is significant, but there is another level of optimization that is uniquely available through Intel's CTOOLS compilers: profile-driven optimization. This two-pass compilation procedure allows the compiler to make optimizations based on runtime behavior as well as the static information used by conventional 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. Standard (i.e., non-program-wide) optimizations are referred to as 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 before the end of the linking phase.

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.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Profiling Lab**.
4. Follow the **Profiling Tutorial** link in the online help.

Using profile-driven optimization, an increase in runtime performance of 27% is obtained. The average 80960 application can expect to gain 15 to 30% performance improvement through the use of this technology. This boost in performance is available to you without any further investment in hardware.

C++ Virtual Function Optimizations

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, however, 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.

Use the following commands to compile the `optimize.cpp` program.

1. Choose **Compiler.**
2. Choose **Static Optimizations**.
3. Choose **C++ Optimizations**.
4. Choose **C++ Virtual Opts**.
5. Choose **Make -NoVOpt** to compile without virtual function optimizations, link, and download the program automatically.
6. To execute the program, type:

   ```
   run
   ```

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 28.0329 seconds.
   Program exited normally
   ```

7. Type: **`quit`**
8. Choose **Make -VOpt** to compile with virtual function optimizations, link, and download the program automatically.
9. To execute the program, type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/optimize
   Now starting C++ routine ...
   Time for C++ routine was 24.0449 seconds.
   Program exited normally
   ```
10. Type: **`quit`**
11. Choose **Results**.

The virtual function optimizations yielded a 14.2% improvement.

Note the runtime performance at each optimization level as shown below.

Table 8-2 i960 Processor Optimization Results

Optimization Level	C Execution Time	C++ Execution Time
no optimization (-O0)	81.141323 seconds	42.0705 seconds
maximum static (-O4)	9.874086 seconds	31.2296 seconds
global optimization	7.254407 seconds	28.0329 seconds
profile-driven	7.20885seconds	NA
Virtual Function Optimization	NA	24.0449 seconds

Building Self-contained Profiles with gmpf960

A *raw* profile contains program counters that record 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 has a very short useful life. When changes are made in the 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 used to optimize a program 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.

1. Choose **Compiler**.
2. Choose **Profiling Optimizations**.
3. Choose **Self-Contained**.
4. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
5. Specify the program database directory.

   The PDB can be specified by setting the environment variable `G960PDB`.

   For example, if you chose the default directory during installation, enter:

   ```
   SET G960PDB=C:\quickval\prof_lab\lab_pdb
   ```

   Or, specify the PDB at compiler invocation time with the `Z`*`dir`* option, as shown in the example below.

   ```
   gcc960  -Zmypdb  foo.o
   ```
6. Compile for profile instrumentation.

   Insert profile instrumentation into *quick* so that when the linked program is executed, a profile can be collected. Type:

   ```
   gcc960 -Fcoff  -T{Link-dir} -A{arch} -fdb
   -gcdm,subst=:*+fprof -o quick quick.c
   ```

The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcysx` specifies `mcysx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-gcdm,subst=:*`	The tool that performs the global decision making and optimization step is invoked from within the linker when the `gcdm` option is used. The substitution control specifies a module-set specification of only eligible modules not linked in from libraries.
`+fprof`	causes generation of profile instrumentation.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

7. Collect 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. Type:

   ```
   gdb960 -t mon960 -b 115200 -r com1 -D lpt1 quick
   ```

   The options in this gdb960 compiler command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 115200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`quick`	the executable file

8. Use the gdb960 debugger to execute `quick`. Enter:

   `run`

9. Exit the debugger. Enter:

   ```
   quit
   ```

10. Enter the command:

    ```
    gmpf960 -spf quick.pf default.pf
    ```

    The options in this gmpf960 compiler command are:

`-spf`	causes a self-contained profile, `quick.pf`, to be produced as output.
`default.pf`	The input profile.

11. Recompile the `quick.c` source code using the profiling information obtained by the instrumentation. Type:

    ```
    gcc960 -Fcoff -T{Link-dir} -A{arch} -fdb
    -gcdm,iprof=quick.pf -o quick quick.c
    ```

    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcysx` specifies `mcysx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=quick.pf`	This supplies a profile file `quick.pf` to the global decision making and optimization step.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

12. Change the control structure of `quick.c.`

    Edit `quick.c`. Find the procedure called QUICK. In this procedure, there is a control structure:

    ```
    for(i = 2; i <= SORTELEMENTS; i+=1)
    {
        (LOGIC)
    }
    ```

Change the control structure to:

```
i = 2;
while (i <= SORTELEMENTS)
{
    (LOGIC)
    i+=1;
}
```

13. Compile the new `quick.c` using the interpolated profile. Type:

    `gcc960 -Fcoff -T`*`{Link-dir}`*`-A`*`{arch}`*`-fdb`
    `-gcdm,iprof=quick.pf -o quick quick.c`

    The options in this gcc960 compiler command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcysx` specifies `mcysx.gld`.
`-fdb`	All modules subject to program-wide optimization must be initially compiled with the `fdb` option.
`-Gcdm,iprof=quick.pf`	This supplies a profile file `quick.pf` to the global decision making and optimization step.
`-o quick`	the executable file will be named `quick`
`quick.c`	the source file

Notice that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`.

NOTE. *The beauty of this example is that the global decision making and optimization option (`-gcdm`) accepts the interpolated profile, `quick.pf`, not the results of running this example.*

Compression Assisted Virtual Execution (CAVE)

This CTOOLS feature allows non-critical parts of an application's machine code to be stored in memory in compressed form resulting in reduced target memory requirements. The code is expanded into native machine code on demand for 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 of 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, 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, that 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.

Note that due to the overhead of decompressing code at runtime, only non-performance critical code should be secondary functions, such as error handling code or initialization code. You can use runtime profile information generated by gcov960 to aid in selecting the set of secondary functions.

This example uses a tic-tac-toe game to show how to reduce target memory requirements. The text sections of compressed and uncompressed tic-tac-toe executables are compared. Additionally, this example demonstrates how to specify functions for compression.

For the sake of demonstration, we compress performance-critical code in the tic-tac-toe program. The purpose of this example is to show the reduced text section of the executable, not demonstrate run times.

C Example

1. Choose **Compiler**.
2. Choose **C Cave**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Use the gcc960 `mcave` option or `#pragma cave` to designate the specified functions as secondary. In the tic-tac-toe example, `ttt.c`, the following `#pragma` has been added:

   ```
   #pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)
   ```

   where `Initialize, Winner, Other, Play, Evaluate, Best_Move, Describe, Move,` and `Game` are all functions to be compressed.
5. Edit `ttt.c`. Make sure the `#pragma cave` program line is commented out:

   ```
   /*#pragma cave(Initialze, Winner, Other, Play,
   Evaluate, Best_Move, Describe, Move, Game)*/
   ```
6. Compile the tic-tac-toe program. Enter:

```
gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c
```

   The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcysx` specifies mcysx.gld.
`-o ttt`	names the executable file ttt
`ttt.c`	input file

7. Check the text section size of the uncompressed program. Enter:

   `gsize960 ttt`

   The option in this command is:

`ttt`	name of the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.

8. Edit `ttt.c`. Make sure the `#pragma cave` program line is uncommented:

```
#pragma cave(Initialze, Winner, Other, Play,
Evaluate, Best_Move, Describe, Move, Game)
```

9. Compile the tic-tac-toe program with the pragma program line. Enter:

   `gcc960 -A{arch} -Fcoff -T{Link-dir} -o ttt ttt.c`

   The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcysx` specifies mcysx.gld.
`-o ttt`	names the executable file ttt
`ttt.c`	input file

10. Check the text section size of the compressed program. Enter:

    `gsize960 ttt`

    The option in this command is:

`ttt`	executable file

The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 8-3 Uncompressed Text Sections

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	33,764	32,944	32,768	32,976	31,600

Table 8-4 After Function Compression

State	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	31,908	30,832	30,816	30,832	29,648
Cave Section	1,818	1,770	1,746	1,800	1,776
Total	33,726	32,602	32,562	32,632	31,424

Table 8-5 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
0.1%	1.0 %	0.6 %	1.0 %	0.6 %

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

C++ Compression Assisted Virtual Execution (CAVE)

1. Choose **Compiler**.
2. Choose **C++ Cave**.
3. Choose **Make**.The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Use the *gcc960* `mcave` option or `#pragma cave` designate the specified functions as secondary. In the C++ example, `cavecpp.cpp`, the following `#pragma` has been added:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

   where `initSetName`, `initSetDept`, `initSetGpa`, `initSetNumPubs`, `isOutstanding`, `printName`, and `InitializeRecords` are all functions to be compressed, i.e., all functions are secondary functions. All other functions of the program are primary functions.

The primary set should contain performance-critical functions that are not to be affected by the CAVE mechanism; the secondary set is subject to compression. Secondary functions are compressed by the linker and reside in memory in compressed form.

The C++ compiler behaves in essentially the same manner as the C compiler when the mcave or Gcave options are used - generating all functions in the compilation unit for which this option is in effect as secondary.

A user typically designates a single function as secondary through the use of `pragma cave`. The following statement for example designates the function max as secondary.

```
# pragma cave max
```

However in C++ overloaded functions have the same name. Member functions of two different classes are also allowed to have the same name and these member functions can in turn have the same name as a function with file scope.

When a user specifies a function as secondary through the use of `pragma cave`, the C++ compiler treats all functions with this name as secondary. To illustrate, consider the following example:

```
# ifdef PRAGMA
# pragma cave max
# endif

int max(int a, int b)
{
return a > b ? a : b;
}

float max(float a, float b)
{
return a > b ? a : b;
}

class Tclass1 {
int a, b;
public:
int max();
};
```

```
int Tclass1::max()
{
return a > b ? a : b;
}

class Tclass2 {
float a, b;
public:
float max();
};


float Tclass2::max()
{
return a > b ? a : b;
}

Tclass1 t1;
Tclass2 t2;
```

The Compiler treats all the following functions as secondary.

```
int max(int, int);
float max(float, float);
int Tclass1::max();
float Tclass2::max();
```

5. Choose **Qv Code**. Edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is commented out:

   ```
   //#pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

6. Compile the C++ program. Enter:

   `gcc960 -A{`*`arch`*`} -Felf -T{`*`Link-dir`*`} -stdlibcpp -o cavecpp cavecpp.cpp`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A{`*`arch`*`}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{`*`Link-dir`*`}`	specifies the linker directive file. For example, `-Tmcysx` specifies `mcysx.gld`.

`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `cavecpp`
`cavecpp.cpp`	input file

7. Check the text section size of the uncompressed program. Enter:

   ```
   gsize960 cavecpp
   ```

   The option in this command is:

`cavecpp`	specifies the executable file

   The sizer responds by displaying the sizes of the various code sections. Write down the size of the uncompressed text section.

8. Choose **Qv Code** and edit `cavecpp.cpp`. Make sure the `#pragma cave` program line is uncommented:

   ```
   #pragma
   cave(initSetName,initSetDept,initSetGpa,initSetNumPu
   bs,isOutstanding,printName,InitializeRecords)
   ```

9. Compile the C++ program with the pragma program line. Enter:

   `gcc960 -A{`*`arch`*`} -Felf -T{`*`Link-dir`*`} -stdlibcpp`
   `-o cavecpp cavecpp.cpp`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, -AHD specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcysx` specifies `mcysx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-o cavecpp`	specifies the executable file `ttt`
`cavecpp.cpp`	specifies the input file

10. Check the text section size of the compressed program. Enter:

    `gsize960 cavecpp`

    The option in this command is:

`cavecpp`	executable file

The sizer responds by displaying the sizes of the various code sections. Write down the size of the compressed text section. In this example, you can expect a code size reduction of approximately 1 percent. Here are some typical results for the supported processor types:

Table 8-6 Uncompressed Text Sections

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Uncompressed	89,788	84,196	83,512	84,196	81,764

Table 8-7 After Function Compression

	80960Rx Size	80960Hx Size	80960Jx Size	80960Cx Size	80960Sx Size
Compressed Text	87,612	81,892	81,512	81,892	79,796
Cave Section	1,920	1,546	1,514	1,546	1,512
Total	89,532	83,438	83,026	83,438	81,308

Table 8-8 Improvement

80960Rx	80960Hx	80960Jx	80960Cx	80960Sx
1%	1%	1%	1%	1%

Note that the purpose of this example is to teach you how to use the CAVE feature with programs. Though the improvements are small, you can expect much better results with real-world programs of approximately 100 Kbytes and larger, especially if the software has many non-critical functions.

Debugging with gdb960

A software debugger is a useful tool that allows you to learn more about the behavior of an application program while it is running on a target or simulator. gdb960 is a source-level debugger that allows you to interact with your application program running on a target system through the debug monitor, MON960. MON960 is resident on the Cyclone CPU module.

This example uses the card game, Go Fish, and is designed to teach you a few debugger commands so that you can further examine the example programs provided with this kit or your own programs. In the card game, Go Fish, you and the computer each get several cards. You take turns guessing which cards are in each other's hands. When you guess correctly, you acquire that card. If you don't guess correctly, you need to "Go Fish" and draw another card from the pack. When you get four-of-a-kind, you remove those cards from your hand. The objective of the game is to have the most sets of four-of-a-kind when either you or the computer has no cards remaining in your hands.

NOTE. *This example uses the command line interface to gdb960. The program also features a Graphical User Interface in both Windows and UNIX. See The gdb960 User's Manual for more information.*

1. Choose **Debugger**.
2. Choose **gdb960 Tutorial**.
3. Choose **Make** to compile, link, and download the program automatically.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.

NOTE. *DEBUGGING SHORTCUTS*
Abbreviations for gdb960 commands are accepted as long as they are unambiguous.
To ***run,*** *enter:* ***r***
To ***break,*** *enter:* ***br***
To ***list,*** *enter:* ***l***
To ***continue,*** *enter:* ***c***
To ***print,*** *enter:* ***p***
To ***clear,*** *enter:* ***cl***
To ***quit,*** *enter:* ***qu***
For ***help,*** *enter:* ***he***

4. **Do Not Type Run!** First, use the gdb960 debugger to set a breakpoint at function `main()`. Type:

   `break main`

   The debugger responds by displaying:

```
Breakpoint 1 set at 0xa0008570: file fish.c, line 209.
```

5. Set a second breakpoint at line 275. Type:

   `break 275`

   The debugger responds by displaying:

```
Breakpoint 2 set at 0xa0008bc4: file fish.c, line 275.
```

6. To execute the program from the beginning, type:

   `run`

   The debugger responds by displaying:

```
Starting program: C:\QUICKVAL/fish
Breakpoint 1, main() at fish.c, 209.
209    srand();
```

7. To display the code at the breakpoint, type:

   `list`

   The debugger displays lines 204-213 of the `fish.c` source. To see the next ten lines, type **`list`** again.

8. To continue executing the program from this location, type:

   `continue`

   The debugger responds by displaying:

   ```
   Continue.
   Would you like instructions[n]?
   ```

9. Reply by typing **`y`** for yes or <Enter> or **`n`** for no.

   ```
   your hand is: A A 6 6 8 8 9
   Breakpoint 2, game() at fish.c:275.
   275    if(!move(yourhand,myhand,g=guess(),0))break;
   ```

10. In the source code in step 9, there are two variable arrays, `myhand` and `yourhand`. `Myhand` is the computer's hand and `yourhand` is yours. To look at the card in the computer's hand, type:

    `print myhand`

    The debugger responds by displaying:

```
$1="000\000\000\001\000\002\000\001\000\000\001\002\000"
```

    `myhand[0]` does not represent a card.

    `myhand[1]` represents the number of Aces.

    `myhand[2]` represents the number of 2s, and so on.

    The same order of cards is represented in the array, `yourhand`.

    If a King is drawn by either player, `myhand[13]` or `yourhand[13]` will appear when you print the array.

11. Using the ability to see the computer's hand, you are able to beat the computer every time. Clear the first breakpoint at the function `main()` and continue playing the game, looking at the computer's hand any time you need to. To clear the breakpoint at `main()`, type:

    `clear main`

    The debugger responds by displaying:

    ```
    Deleted breakpoint 1
    ```

12. To continue executing the program, type:

    `continue`

13. If you need further assistance beating the computer, contact the 80960 Technical Support Group for more hints.
14. Type: **`quit`**

Debugging Optimized Code

CTOOLS can use the ELF object module format and DWARF Version 2 debug information format as described in the *80960 Embedded Application Binary Interface (ABI) Specification* (order number 631999). The new formats enable more accurate mapping between source and object code at higher optimization levels and ease production code debugging.

This example shows that at the highest level of module-local optimization, it is possible to set a breakpoint on an inline function using ELF/DWARF, while with COFF this is not possible.

1. Choose **Debugger**.
2. Choose **C ELF/DWARF Format**.
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Compile *swap.c* with no module-local optimizations (no inlining). This shows that the procedure *swap* is not inlined. Enter:

```
gcc960 -Felf -T{Link-dir} -A{arch} -O0 -S swap.c
```

   The options in this command are:

`-Felf`	creates an ELF format output file
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcysx` specifies `mcysx.gld`.
`-O0`	no module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

5. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, see the call to the procedure swap:

   ```
   callj _swap
   ```

   This is an out-of-line call to the procedure swap. The function swap has not been inlined.

6. Now, compile `swap.c` with the highest level of module-local optimizations. This inlines the procedure `swap`.

   `gcc960 -Felf -T`*`{Link-dir}`*` -A`*`{arch}`*` -O4 -S swap.c`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcysx` specifies `mcysx.gld`.
`-O4`	highest level of module-local optimizations
`-S`	generate assembly code from the source code
`swap.c`	input file

7. Edit `swap.s` (the generated assembly file from `swap.c`). In the function `_main`, note the call to the procedure `swap` does not exist:

   ```
   callj _swap  /* Does Not Exist*/
   ```

   The procedure swap has been inlined.

8. Recompile using the `-O4` optimization level, the ELF/DWARF format, and add debugging information.

`gcc960 -Felf -T`*`{Link-dir}`*` -A`*`{arch}`*` -O4 -g -o swap swap.c`

   The options in this command are:

`-Felf`	create an ELF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcysx` specifies mcysx.gld.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file

`-o swap`	names the executable file swap
`swap.c`	input file

9. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

```
gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap
```

   The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`swap`	the executable file

10. DO NOT TYPE RUN!

    First, set a breakpoint on the procedure *swap.* Enter:

```
break swap
```

    The debugger responds by displaying:

```
breakpoint 1 @0xa00080f0:file swap.c, line 43
breakpoint 2 @0xa0008148:file swap.c, line 54
```

    Breakpoint 1 is the out-of-line reference to the procedure `swap`. Breakpoint 2 is the inline reference to the procedure `swap`.

    `Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is still possible to set a breakpoint on the inline function. Breakpoint 2 stops program execution.

11. To execute the program, enter:

```
run
```

    The debugger responds by displaying:

```
Breakpoint 2, main() @ swap.c: 54
54 printf(ìThe smallest number is %d\nî,a);
```

12. To continue the program, enter:

```
c
```

    When the program has finished, enter:

```
quit
```

13. Compile using the `-O4` optimization level, the COFF format, and add debugging information.

```
gcc960 -Fcoff -T{Link-dir} -A{arch} -g -O4 -o swap swap.c
```

The options in this command are:

`-Fcoff`	create a COFF format output file
`-A`*`{arch}`*	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T`*`{Link-dir}`*	specifies the linker directive file. For example, `-Tmcysx` specifies `mcysx.gld`.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o swap`	names the executable file `swap`
`swap.c`	input file

14. Download the executable file, `swap`, to the Cyclone eval board memory. Enter:

    `gdb960 -t mon960 -b 115200 -r com1 -D lpt1 swap`

    The options in this command are:

`-t mon960`	MON960 is on the target
`-b 115200`	use 155200 baud rate
`-r com1`	use serial port 1
`-D lpt1`	use parallel port 1
`swap`	the executable file

15. **Do Not Type Run!!**

    First, set a breakpoint on the procedure `swap`. Enter:

    `break swap`

    The debugger responds by displaying:

    ```
    breakpoint 1 @0xa00080f0
    ```

    Breakpoint 1 is the out-of-line reference to the procedure `swap`. Notice that no inline breakpoint has been set. This breakpoint does not stop execution of the program.

    `Swap.c` was compiled with a high level of module-local optimizations that included function inlining, and it is not possible to set a breakpoint on the inline function. Program execution does not stop.

16. To execute the program, enter:

    `run`

    The debugger responds by displaying the smallest number from the swap. There is no break in program execution.

17. When the program has finished, enter:

    `quit`

    You have now seen that with the ELF/DWARF format, it is now possible to debug your production code, even after high levels of program optimization.

Debugging Optimized C++ Code Tutorial

The C++ compiler generates debug information using the DWARF format when the `-g` option is specified with the `-Felf` option. This debug information format is richer than that of other supported OMFs, and allows more reliable debugging under optimization.

This tutorial demonstrates that at the highest level of module-local optimization, debugging a C++ application is still possible due to the DWARF debug format.

In this example, you compile a C++ program using the `-O0` optimization compiler option, which disables all optimizations, including those that may interfere with debugging. The same C++ program is then compiled using the highest-level of module-local optimization, `-O4`.

There are several levels of program optimization available with the CTOOLS development tool suite. Typically, low levels of optimization are used during the debugging phase. Certain optimizations can cause significant code changes that may make high-level debugging difficult. Once the application is functioning properly, the application's performance may be increased by using a higher level of optimization. The static optimization options are:

`O0`	Turn optimization off
`O1`	Basic optimization
`O2`	strength-reduction, instruction scheduling for pipelining, etc...

`O3`	`O2` plus `fconstprop, finline-functions`, etc...
`O4`	`O3` plus `fsplit-mem, fmarry-mem, fcoalesce`

Level O4 is the highest level of static optimization. Please refer to the *i960 Processor Compiler User's Guide* for more information on ELF/DWARF and compiler optimizations.

In this tutorial, you compile and debug a C++ program, `cppdwarf.cpp`, that contains many of the advanced features of the C++ language, including:

- Classes
- Public, protected, and private variable accessibility
- Virtual functions
- Scope operators
- Overloaded functions
- Class inheritance

Using ELF/DWARF, both levels of optimization, `-O0` and `-O4`, retain the C++ program structure so that the above features may be investigated.

1. Choose **Debugger**.
2. Choose **C++ ELF/DWARF Format**.
3. Choose **Make**. The following tutorial is displayed in the QUICK*val* browser, and the command lines may be entered at the Command Prompt window.
4. Compile the program using the `-O0` optimization level. In the Command Prompt window, enter the following command:

   ```
   gcc960 -Felf -A{arch} -T{Link-dir} -stdlibcpp -O0 -g
   -o cppdwarf cppdwarf.cpp
   ```

   The options in this command are:

`-Felf`	creates an ELF format output file.
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD.
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcysx` specifies mcysx.gld.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.

`-O0`	specifies the lowest level of module-local optimizations.
`-g`	includes debug information in object file.
`-o cppdwarf`	specifies the executable file `cppdwarf`.
`cppdwarf.cpp`	specifies the input file `cppdwarf.cpp`.

5. Run the program using the debugger, enter:

   `gdb960 -t mon960 -b {`*`baudrate`*`} -r {`*`comport`*`} -D {`*`parallel port`*`} -pci cppdwarf`

   The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are 1200, 2400, 9600, 19200, 38400, 57600, and 115200.
`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are com1, com2, ... com99.
`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are lpt1 and lpt2.

`-pci`	sets the code download option for the PCI bus (optional). When no serial port is specified, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used).
`cppdwarf`	specifies the executable file `cppdwarf`.

6. **Do Not Enter Run!**

   Now you are ready to examine some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes.

   At the (gdb960) prompt, enter:

   `ptype person`

   The following data type information concerning the class `person` appears:

Example 8-1 person Class

```
   type = class person {
     protected:
       char name[40];
       char dept[40];
     public:
       void setName ();
       void setName (char *);
       void setDept ();
       void setDept (char *);
       void printName ();
       virtual int isOutstanding ();
       virtual char * getDept ();
}
```

Please note the following concerning the above output:

- The entire class information for person is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.

Another C++ class in the program is `professor`, which inherits from the person class. Again, you use the gdb960 command `ptype` to display a description of the `professor` class.

7. At the (gdb960) prompt, enter:

   `ptype professor`

   The following data type information concerning the class `professor` appears:

Example 8-2 professor Class

```
  type = class professor : public person {
    private:
      int numPubs;
    public:
      void setNumPubs ();
      void setNumPubs (int);
      virtual int isOutstanding ();
}
```

Please note the following concerning the above output:

- The entire class information for `professor` is displayed, including variables and member functions.
- The `public`, `protected`, and `private` variable accessibility qualifiers are displayed for variables and member functions.
- All member functions are displayed, including virtual functions and overloaded functions.
- `type = class professor : public person` indicates that the `professor` class inherits from the `person` class.

8. You are ready to set some breakpoints.

   a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

      `break professor::setNumPubs`

      The following information concerning breakpoints is displayed:

      ```
      [0] cancel
      [1] all
      [2] professor::setNumPubs(int) at
      cppdwarf.cpp:125
      [3] professor::setNumPubs(void) at
      cppdwarf.cpp:118
      ```

      Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.

   b. Set a breakpoint on all `professor::setNumPubs` functions. At the `>` prompt, enter: **`1`**

      The following information about breakpoints is displayed:

      ```
      Breakpoint 1 at 0xa00083d0: file cppdwarf.cpp,
      line 125.
      Breakpoint 2 at 0xa0008358: file cppdwarf.cpp,
      line 118.
      ```

   c. Set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

      `break professor::isOutstanding`

      The following information concerning breakpoints is displayed:

      ```
      Breakpoint 3 at 0xa0009080: file cppdwarf.cpp,
      line 110.
      ```

9. You are now ready to start the program. At the (gdb960) prompt, enter:

   `run`

   Notice that the program stops at all three of the breakpoints.

10. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.

11. At the (gdb960) prompt, enter: **`quit`**

    The results of the debug session were as expected because no optimizations had been performed on the source code during compilation. You can now recompile the `cppdwarf.cpp` program using the highest-level of module-local optimization and repeat the previous debug session.

12. Compile the program using the `-O4` optimization level. In the Command Prompt window, enter the following command:

    ```
    gcc960 -Felf -A{arch}-T{Link-dir} -stdlibcpp -O4 -g
    -o cppdwarf cppdwarf.cpp
    ```

    The options in this command are:

`-Felf`	create an ELF format output file
`-A{arch}`	specifies the architecture. For example, `-AHD` specifies an 80960HD
`-T{Link-dir}`	specifies the linker directive file. For example, `-Tmcysx` specifies `mcysx.gld`.
`-stdlibcpp`	instructs the compiler to link in the standard C++ libraries when creating an absolute module.
`-O4`	highest level of module-local optimizations
`-g`	include debug information in object file
`-o cppdwarf`	specifies the executable file `cppdwarf`
`cppdwarf.cpp`	input file

13. Run the program using the debugger, enter:

    ```
    gdb960 -t mon960 -b {baudrate} -r {comport} -D
    {parallel port} -pci cppdwarf
    ```

    The options in this command are:

`-t mon960`	specifies that MON960 is on the target (optional). `-t mon960` is optional since `mon960` is the default.
`-b 115200`	sets the baud rate for serial communication (optional). This option, `-b 115200`, is not needed when the serial port is not being used. Possible baud rates are: 1200, 2400, 9600, 19200, 38400, 57600, and 115200.

`-r com1`	sets the port to use for serial communication (optional). This option, `-r com1`, is not needed when the serial port is not being used; however, the `-pci` option is required when no serial port is used. Possible serial ports are: com1, com2, ... com99.
`-D lpt1`	sets the code download option for the parallel port (optional). This option, `-D lpt1`, is not needed when the serial port or PCI bus is used for code download. Possible parallel ports are: lpt1 and lpt2.
`-pci`	sets the code download option for the PCI bus (optional). When no serial port is given, the PCI bus is used for serial communication also. The `-r comx` option is required when the PCI bus is not used (i.e., when the `-pci` option is not used.)
`cppdwarf`	specifies the executable file.

14. **Do Not Enter Run!**

    You are now ready to investigate some features of the downloaded C++ program, `cppdwarf.cpp`. A C++ class in the program is `person`. The gdb960 command `ptype` may be used to display a description of a data type, including classes. At the (gdb960) prompt, enter:

    `ptype person`

    Please note, the output matches that of Example 8-1, "person Class". Optimizations did not affect the `person` class output. It is the same as the first debug session.

15. Another C++ class in the program is `professor`, which inherits from the person class. Once again, you use the gdb960 command `ptype` to display a description of the `professor` class. At the (gdb960) prompt, enter:

    `ptype professor`

    Again please note, the output matches that of Example 8-2, "professor Class". Optimizations did not affect the `professor` class output. It is the same as the first debug session.

16. You are now ready to set some breakpoints.
    a. First, set a breakpoint on the overloaded function `setNumPubs` in the `professor` class. At the (gdb960) prompt, enter:

       `break professor::setNumPubs`

       The following information concerning breakpoints is displayed:

       ```
       [0] cancel
       [1] all
       [2] professor::setNumPubs(int) at
       cppdwarf.cpp:125
       [3] professor::setNumPubs(void) at
       cppdwarf.cpp:118
       ```

       Option 0 cancels the breakpoint operation. Option 1 sets a breakpoint on all the `professor::setNumPubs` functions. Option 2 only sets a breakpoint on `professor::setNumPubs(int)` on line 125 of `cppdwarf.cpp`. Similarly, option 3 only sets a breakpoint on `professor::setNumPubs(void)` on line 118 of `cppdwarf.cpp`.
    b. Set a breakpoint on all `professor::setNumPubs` functions, so At the `>` prompt, enter: **`1`**.

       The following information about breakpoints is displayed:

       ```
       Breakpoint 1 at 0xa00082e4: file cppdwarf.cpp,
       line 125.
       Breakpoint 2 at 0xa0008294: file cppdwarf.cpp,
       line 118.
       ```
    c. Finally, set a breakpoint on the virtual function `professor::isOutstanding`. At the (gdb960) prompt, enter:

       `break professor::isOutstanding`

       The following information concerning breakpoints is displayed:

       ```
       Breakpoint 3 at 0xa0008960: file cppdwarf.cpp,
       line 111.
       ```

17. You are now ready to start the program. At the (gdb960) prompt, enter:
    `run`
    Notice that the program does not stop at all three of the breakpoints. As can be seen, the DWARF debug information format is very rich, 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.
18. To continue after a break, use the gdb960 command **`continue`**, or enter the keyboard shortcut **`c`**.
19. At the (gdb960) prompt, enter: **`quit`**

CONGRATULATIONS! You may now know how to use ELF/DWARF to debug your optimized C++ code.

Writing Flash

NOTE. *In order to write to flash on your Cyclone base board, you need a 12 volt power supply. Also, these instructions are used with the CTOOLS 6.0 and MON960 3.2.3 toolsets.*

This example teaches you the following:

- Writing to flash on the Cyclone base board.
- Booting off of the flash in socket U27 of the Cyclone base board, as opposed to the flash on the CPU Module.
- Setting the Cyclone base board to 12 volts.
- Using *mondb.exe* as a simple utility to download and execute an application program on the target board running MON960.
- Using *mondb.exe* to write flash.
- Building a new monitor for a particular i960 microprocessor family member.
- Retargeting MON960 for other boards.

Complete this step:

1. Choose **MON960**.
2. Choose **Writing Flash.**
3. Choose **Make**.

   The following tutorial is also displayed in the browser. Enter your commands in the Command Prompt window provided.
4. Identify the Flash on the Cyclone base board.

   A blank Flash chip ships on each Cyclone base board in socket U22. To write MON960 to Flash, you must move the blank Flash from socket U22 to socket U27.
5. Set the Cyclone base board voltage to 12 volts.

   Locate the four-position DIP switch labeled S1. Flip S1.1 to the *ON* position. This enables VPP to the Cyclone base board Flash.
6. Power up the Cyclone eval base board

   Locate the four-pin connector that interfaces to a secondary power supply labeled J6. Three of the connector pins connect to +5 VDC, +12 VDC and ground. (On the PCI-SDK Platform, +12 VDC and +5 VDC power is supplied through the edge connector.)
7. Edit `Version.c`.

   a. Change directories to where the `version.c` file resides. The default installation directory for CTOOLS is:

   `c:\intel960\src\mon960\common`

   If you cannot find the mon960 directory, You need to install MON960 as directed in the *MON960 Debug Monitor User's Manual*.

   Version.c contains the following information:

```
const char mon_version_byte =  nn;   /* version n.n = nn */
const char base_version[] = "MON960 n.n.n";
const char build_date[] = __DATE__;
```

   b. Change the file contents to reflect that this is your version of MON960. For example, change

```
const char base_version[] = "MON960 n.n.n";
```

   to:

```
const char base_version[] = "MY MON960";
```

   c. Save `Version.c`.

8. Build the new MON960 from source (optional)

   By default the source for MON960 is located at:
   `c:\intel960\src\mon960\common`

   You may use the pre-built version of MON960 there, or build a custom verion. To create a custom version:

   a. Copy `makefile.xxx` to
      `c:\intel960\src\mon960\common\makefile`.

      where xxx is one of the following make files:

      `makefile.ic` (ic960 interface, COFF format)

      `makefile.ie` (ic960 interface, ELF format)

      `makefile.gc` (gcc960 interface, COFF format)

      `makefile.ge` (gcc960 interface, ELF format)

   b. Issue the commands:

      ```
      nmake -s makefile
      cysx
      ```

      This creates a file called `cysx.fls`.

9. Write the Flash

   To write the Flash, use the mondb.exe utility located in the `intel960\bin\` directory. If you are going to use the pre-built MON960 files, they are located in the `intel960\roms` directory. For example, if you used the default installation directory and are using the pre-built MON960 files for the 80960Sx, enter:

   ```
   mondb -ser com1 -par lpt1 -ef -ne
   c:\intel960\roms\cysx.fls
   ```

   The options in this command are:

`-ser com1`	use serial port 1
`-par lpt1`	use parallel port 1
`-ne`	no execute
`-ef`	erase Flash
`cysx.fls`	input Flash filename

   Note also that if you built a version of MON960 from the source code as described previously, the `cysx.fls` file will be located in the `c:\intel960\src\mon960\common\` directory.

10. Set Board Voltage Back To +5 VDC

    Locate the four-position DIP switch labeled S1. Set S1.1 to the *OFF* position. This disables VPP to Cyclone EP base board Flash and protects the Flash. Note that the PCI80960DP and i960 `Sx` evaluation platforms do not boot when VPP is enabled and MON960 is running from the evaluation board Flash.
11. Set board to boot from U27 socket

    Locate the four-position DIP switch labeled S1. Set S1.3 ROMSWAP to the *ON* position. This exchanges the addresses of the CPU Module ROM and the base board ROMs. When the switch is *OFF* the processor boots from the CPU Module ROM; when the switch is *ON* the processor boots from the base board ROMs.
12. Reset Base Board

    Locate the reset pushbutton labeled S2. Use this button to manually reset the Cyclone base board and boot from the base board ROMs.

NOTE. *If you have trouble with this example, refer to Chapter 3 for troubleshooting tips.*

How to Add Benchmarking Routines to Your Code

Benchmarking is a common way to evaluate an architecture for its performance. CTOOLS comes with two routines for benchmarking code. These routines are called `bentime()` and `init_bentime()`. `init_bentime()` is called once to program the on-board Counter/Timer to periodically interrupt the processor. The `bentime()` routine returns the time in microseconds based on the count from the interrupt handler, `timer_isr`, and the current count read from the timer. By placing a call to `bentime()` at the start and end of the code you are timing, the elapsed time can be calculated by the difference between the second call to `bentime()` and the first.

1. Choose **Benchmarking**.
2. Choose **Qv Code**.

3. Scroll through the `chksum.c` code for comments that refer to "Benchmarking Routine". You can add similar lines to the code that you want to time.
4. Choose **Make** to compile, link, and download the program automatically.
5. Execute the `chksum` program. Type:

   `run`

   The debugger responds by displaying:

   ```
   Starting program: C:\QUICKVAL/chksum
   Now starting Comersum routine ...
   Time for Checksum was 18.173298 seconds.  Value was
   869e7960.
   Program exited with code 01
   ```
6. Type: **`quit`**

Other i960 Processor Choices and the Remote Evaluation Facility

The i960 RISC processor family has a wide breadth of processors to match your design's price and performance needs. If you wish to evaluate other i960 processor family members, contact your local distributor and order different Cyclone CPU modules, or visit the Remote Evaluation Facility at `http://developer.intel.com/design/i960/testcntr`

NOTE. *The i960 Rx Processor is not available through the Remote Evaluation Facility.*

If you choose to order more CPU modules, you may rest assured that all i960 processor modules plug-n-play with your QUICK*val* kit. This configuration was specifically designed to protect your investment and offer a low cost migration path for future needs.

Communicating with MON960 via Serial Port

A

MON960 is an Intel debug monitor resident on the Cyclone CPU module board.   It is a full-featured monitor, providing the capability to read and write to memory, disassemble processor instructions, set breakpoints, step through instructions, display and modify registers, trace variables, and download executables via the XMODEM protocol.

MON960 is used to provide the communication link between a program executing on the target board and a host-resident debugger such as gdb960. You can also use MON960 to download and execute programs without a debugger via the use of the terminal emulator that is provided with QUICK*val*, as in the following example.

1. Make sure you have compiled and linked the `hello` program, as described in Chapters 4 through 8.
2. Open the **Options** menu on the QUICK*val* main window, choose **Terminal**.
3. Open the **Setup** menu on the terminal window, and choose **Port**.
    - Set the baud rate field to 9600 baud.
    - Set the Data bits to 8.
    - Set the Stop bits to 1.
    - Set the Parity to None.
    - Set the Flow Control to None.
    - Set the Connector to the serial port you are using.
    - Choose **OK**.

NOTE. *The Cyclone board default baud rate is 9600 baud. However, the Cyclone board supports a maximum baud rate of 115200. Your initial diagnostic messages are not readable until after you press <Enter> several times in the terminal window.*

4. Open the **File Transfer** menu and choose **Protocol**.
5. From the drop-down box on the Protocol window:
   - Choose **XModem-CRC**.
   - Choose **OK**.
6. Enter <cr> several times to invoke MON960. The monitor responds by displaying:

```
MON960 User Interface: Version MON960 x.x MM/DD/YYYY
Cyclone Baseboard; for i960 xx at xx Mhz with xMB DRAM;
xx stepping number xx
Copyright Intel Corporation
=>
```

7. From the terminal emulation prompt, type:

   `download`

   MON960 responds by displaying:

   `Downloading`
8. Open the **File Transfer** menu, and choose **Upload**.
9. in the Send File window, select the file `hello` and choose **OK**. If you installed QUICK*val* to the default directory and compiled `hello.c` as instructed in chapters 4 through 8, the file is located at `c:\quickval`.

   Downloading the program to the Cyclone board begins once you have selected your file. After downloading has completed, MON960 displays:

```
-- Download complete --
Start address is : A0008000
=>
```

To execute the program, type:

`go`

MON960 responds by displaying:

```
Hello... Welcome to the 80960xx QUICKval Kit!
System Check Completed!!
Now you may proceed with our Example Programs.
Program Exit: 01
=>
```

10. Open the File menu and choose Exit to terminate this session.

NOTE. *Refer to the MON960 Debug Monitor User's Guide for more information on performing simple debugging tasks with the MON960 debug monitor's user interface (UI).*

The Saxsoft Webster* Browser

B

The 80960 QUICK*val* includes an integrated web browser to display the online help and provide you with tutorials. If you have an internet connection, you can even use this program to browse the web. This is handy because there are some places in the online information system that point you to the Intel World-Wide Web Site (`www.intel.com`) for specification updates, technical support, and many other resources. To use Webster to browse the web via proxy server, open the Settings menu and set the connection settings to the same as your existing browser (e.g., Microsoft Internet Explorer* or Netscape Navigator*).

NOTE. *When inputting your proxy settings to Webster, it is recommended that you manually type the information. Do not simply cut and paste the settings from your existing browser. The Windows Clipboard may carry over non-printing (non-viewable) characters that are not part of the correct text. The path may look correct, but the hidden characters may cause Webster to fail to reach your proxy server.*

If you use a dialer to connect to the internet, Webster asks you if you want to connect when you click on a remote link.