Stardent

Programmer's Guide

Stardent Computer Inc. Part Number: 340-0116-01

Copyright © 1990
an unpublished work of Stardent Computer Inc.
All Rights Reserved.

This document has been provided pursuant to an agreement with Stardent Computer Inc. containing restrictions on its disclosure, duplication, and use. This document contains confidential and proprietary information constituting valuable trade secrets and is protected by federal copyright law as an unpublished work. This document (or any portion thereof) may not be: (a) disclosed to third parties; (b) copied in any form except as permitted by the agreement; or (c) used for any purpose not authorized by the agreement.

Restricted Rights Legend for Agencies of the U.S. Department of Defense

Use, duplication or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013 of the DoD Supplement to the Federal Acquisition Regulations. Stardent Computer Inc., 880 West Maude Avenue, Sunnyvale, California 94086.

Restricted Rights Legend for civilian agencies of the U.S. Government

Use, reproduction or disclosure is subject to restrictions set forth in subparagraph (a) through (d) of the Commercial Computer Software—Restricted Rights clause at 52.227-19 of the Federal Acquisitions Regulations and the limitations set forth in Stardent's standard commercial agreement for this software. Unpublished—rights reserved under the copyright laws of the United States.

Stardent™, Doré™, and Titan™ are trademarks of Stardent Computer Inc. UNIX® is a registered trademark of AT&T.

CONTENTS

Preface

1 Program Creation & Maintenance Tools

Learning vi 1-1
The Basics Of Editing With vi 1-2
How To Start vi 1-2
Two Modes Of Operation 1-3
How To Move The Cursor 1-3
How To Enter Text 1-4
How To Delete Text 1-5
Undoing A Command 1-5
Restoring A Line That Was Most Recently Deleted 1-6
How To Save Your Work 1-6
Exiting vi 1-7
Ex Commands 1-7
Generic Form Of An *ex* Command 1-8
Entering Input Mode, In Detail 1-10
Moving the Cursor 1-13
Character Positioning 1-13
More Positioning By Character 1-13
Words, Sentences, Paragraphs 1-14
Moving To Explicit Onscreen Positions 1-14
Scrolling Through The Edit Buffer 1-14
Moving To A Position Defined By A Character String 1-15
Moving To A Specified Line Number 1-16
Marking and Moving To A Position In The File 1-17
Buffers In vi 1-17
Deleting Objects In Vi 1-18
Moving Text Blocks 1-19
Moving Text Blocks By An Alternate Method 1-19
Copying Text Blocks 1-20
Copying Text Blocks By An Alternate Method 1-21
Changing Text 1-21
Changing Text By An Alternate Method 1-22

Using Numbered Buffers 1-23
Using Named Buffers 1-24
Deleting To, Copying To, and Appending To A Named Buffer 1-25
Copying Text Into A Named Buffer 1-25
Appending Text To A Named Buffer 1-25
Commands That Delete To Or Copy To Named Buffers 1-26
Command Shortcuts 1-27
Copying A Named Buffer Into The Edit Buffer 1-28
Command Shortcuts 1-28
Moving A Block Of Text From One File To Another 1-29
Regular Expressions 1-30
Reading The Contents Of An External File or Command 1-32
Temporarily Exiting Vi To Use Other Programs 1-32
Editing Multiple Files 1-33
Repeating Commands 1-35
String Substitution 1-36
Which Line, and Which File 1-37
Specifying Literal Characters 1-37
Vi Command Summary 1-38
Sample commands 1-38
Counts before vi commands 1-38
Interrupting, canceling 1-38
File manipulation 1-39
Positioning within file 1-39
Adjusting the screen 1-40
Marking and returning 1-40
Line positioning 1-40
Character positioning 1-40
Words, sentences, paragraphs 1-41
Corrections during insert 1-41
Insert and replace 1-42
Operators 1-42
Miscellaneous Operations 1-42
Yank and Put 1-43
Undo, Redo, Retrieve 1-43
Bibliography 1-43

2 Using The Stardent 1500/3000 Compilers

The Stardent 1500/3000 Compilation System 2-1
Functions of the Stardent 1500/3000 Compilers 2-2
Specifying the Output Files 2-2
Compilation Control 2-2
Command Line Options 2-2

Preprocessor Options 2-5
Compiler Options 2-6
Loader Options 2-10
Preprocessor Control Statements 2-12
The #include Statement 2-13
The #define Statement 2-13
The #ifdef Statement 2-14
The #ifndef Statement 2-16
The #endif Statement 2-17
The #if Statement 2-17
The #elif Statement 2-18
The #else Statement 2-19
The #undef Statement 2-20
The #line Statement 2-20
Other Forms Of Define and Include 2-20
Compiler Directives 2-21
Linking Multiple Files 2-21
Vector Reporting Facility 2-21
–vsummary 2-21
–vreport 2-22
–full_report 2-26
Vectorizer Strategy 2-31
Vectorizing C Programs 2-32
The Stardent 1500/3000 Compilation System and Fortran 2-39
Calling The Fortran Compiler 2-39
Form of the Options 2-40
Fortran Compiler Options 2-41
Compilation Control Statements 2-47
Format of the Fortran Listing 2-47
Source Code Section 2-47
Storage Map 2-48
Cross Reference 2-49
Compilation Summary 2-50
Errors and Compiler Diagnostics 2-50
The Stardent 1500/3000 Compilation System and C 2-51
Elements of the Stardent 1500/3000 C Compiler 2-52
Calling the Stardent 1500/3000 C Compiler 2-52
C Compiler Options 2-53
Extensions to the Compiler 2-54
Compilation Control Statements 2-54
Argument (Function) Prototypes 2-55
Storage Class threadlocal 2-57
Comment Delimiters 2-58
Vector Math Functions 2-58
Compiler Directives 2-59
Address (&) Arguments 2-60

3 Using Libraries and the Link Editor

Archive Libraries 3-1
Creating an Archive File 3-1
EXAMPLE 3-2
Loading Libraries and Their Order 3-3
Available Libraries 3-4
Creating a Library Abbreviation 3-5
The ld command 3-5
ld and Archive Libraries 3-6

4 Methods For Debugging Code

Stardent 1500/3000 Versus Standard UNIX Systems 4-1
When to use a Debugger 4-2
Debugging Tools 4-2
dbg 4-2
nm 4-3
od 4-3
prof 4-3
size 4-4

5 Running the Debugger

A Simple Initial Session 5-1
Preparing For A Debug Session 5-5
Starting dbg 5-5
Specifying A Search Path For Source and Object Files 5-6
dbg Startup 5-6
Scope 5-7
Location Counter 5-7
dbg Commands 5-9
dbg Keyword Commands 5-10
Simple Expression Commands 5-14
Compound Expression Commands 5-14
DO ... ENDDO Statement 5-15
WHILE statement 5-15
FOR statement 5-16
IF Statement 5-16
Colon-Separated Statement 5-16
Brace-Enclosed Colon-Separated Statement 5-16
Specifying Program Locations For dbg 5-16
Typical Commands 5-17
Starting The Program Running 5-18
Starting A Post-Mortem Debug Session 5-18

Listing The Source Code 5-20
Breakpoints and Actionpoints 5-21
Setting A Breakpoint 5-21
Getting A Summary Of Breakpoints and Actionpoints 5-21
Suspending Actions At Breakpoints 5-22
Restoring Actions At Breakpoints 5-23
Cancelling A Breakpoint 5-24
Continuing After A Break 5-24
Viewing or Modifying Program Variables 5-25
Using Debuggee Functions Within dbg 5-25
Getting Help 5-26
Exiting dbg 5-26
dbg Command Language 5-27
dbg Data Declarations And Operators 5-28
Applying Type Casts To Variables To Match Types 5-28
Operators Understood By dbg 5-28
Radix 5-30
The Status Command 5-31
Controlling Execution Of Debuggee Processes 5-32
Signals And Interrupts 5-32
Miscellaneous Commands 5-33
Examples of often-used dbg commands 5-35
Setting and deleting break/watch points 5-35
Printing values of variables and looking at memory 5-35
Setting dbg parameters (hex, FORTRAN, etc.) 5-36
Looking at the Floating Point Unit 5-36
Abbreviating dbg Commands 5-37
A Sample Debugging Session In C 5-37
Machine Code Related Commands 5-47
Exploring On Your Own 5-48

6 Vector & Parallel Optimization

Fundamental Concepts 6-1
Compiler Techniques 6-6
Program Transformations 6-7
Induction Variable Elimination 6-7
Constant Propagation 6-8
Dead Code Elimination 6-9
Loop Distribution 6-10
Loop Interchange 6-10
Scalar Expansion 6-12
Reduction Recognition 6-13

7 Efficient Programming Techniques

Write Vectorizable Loops 7-2
  Write Convertable Alternate Loops 7-2
  Write Vectorizable DO Loops 7-6
Use COMMON and EQUIVALENCE Carefully 7-7
  Avoid EQUIVALENCES into COMMON 7-8
  Avoid EQUIVALENCED Scalars 7-8
Use Recognized Patterns 7-9
Coding Conditional Vectors 7-11
  Understand Conditional Vector Hardware 7-11
  Use Structured IF-THEN-ELSE Statements 7-13
  Avoid && and || 7-15
  Avoid Unnecessary Error Checks 7-16
  Don't Single Out Specific Iterations 7-17
  Choose Appropriate Conditional Hardware 7-18
Use Double Precision 7-19
Use Integers; Avoid Shorts 7-20
Loop Unrolling 7-20
Data Storage 7-21
Scalar Temporary Variables 7-22
Ensure The Right Loop Vectorizes 7-24
Beware Coupled Reductions 7-27
Avoid Vectorized Structures 7-28
Use –fast 7-29
Avoid Slow Operations 7-29
Summary 7-29
Unformatted I/O In Fortran 7-30
  Data Alignment 7-31
  Arrays Versus DO Loops 7-31

8 Explicit Parallel Programming

Serial and Parallel Programs 8-2
Determining Whether Explicit Parallelization Is Appropriate 8-3
Levels of Parallelism 8-4
Memory Conflicts 8-5
  Private Memory 8-5
  Public Memory 8-6
  Static Memory 8-6
  Dynamic Memory 8-7
  Volatile Memory 8-8
  Summary Of Memory Conflict Situations 8-8
Critical Sections Of Code 8-8
Allocating Private Memory In Fortran 8-9
  Using THREADLOCAL COMMON In Fortran 8-10

Creating Local Stacks (Processor Private Memory) In Fortran 8-11
Allocating Private Memory In C 8-11
Using Threadlocal Variables in C 8-12
Allocating Dynamic Threadlocal Storage In C 8-13
Creating Local Stacks (Processor Private Memory) In C 8-13
Synchronizing Through Public Memory in Fortran 8-14
Synchronizing Through Public Memory In C 8-15
Restriction On Library Access (Fortran or C) 8-17
Creating A Fortran Parallel Procedure 8-17
Calling A Parallel Fortran Procedure 8-19
A Fortran Parallel Processing Example 8-20
Summary For Fortran Parallel Programming 8-24
Miscellaneous Functions 8-25
Operating System Implementation Highlights 8-25
Parallel Processes 8-26
Creating A C Parallel Procedure 8-26

9 Tuning Code

Profiling Programs 9-1
Output Description 9-3
-ploop Option 9-4
-p Option 9-7
Interpreting Profiled Programs 9-7
Other Timing Options for mkprof 9-9
Compiler Directives 9-10
ASIS 9-12
C$DOIT ASIS 9-12
INLINE 9-12
IVDEP 9-12
C$DOIT IVDEP 9-13
IPDEP 9-13
C$DOIT IPDEP 9-13
VBEST 9-14
C$DOIT VBEST 9-14
PBEST 9-15
C$DOIT PBEST 9-15
VPROC 9-16
C$DOIT VPROC *fname, vfname* *9-16*
PPROC 9-16
C$DOIT PPROC SUBR_NAME 9-16
THREADLOCAL 9-17
C$DOIT THREADLOCAL COMMON_NAME 9-17
STATIC 9-17

C$DOIT STATIC COMMON_NAME 9-17
VREPORT 9-18
C$DOIT VREPORT 9-18
NO_PARALLEL 9-18
C$DOIT NO_PARALLEL 9-18
NO_VECTOR 9-18
C$DOIT NO_VECTOR 9-18
SCALAR 9-18
C$DOIT SCALAR 9-18
OPT_LEVEL 9-19
C$DOIT OPT_LEVEL *n* 9-19

10 Language Interfacing

Register Sets 10-1
CPU Registers 10-1
Scalar Registers 10-2
Vector Registers 10-2
Floating Point Computations 10-3
Stack Frame 10-18
Calling Subprograms 10-19
Data Layout In Memory 10-20
Integer Format 10-20
Short Integer Format 10-21
Real Format 10-21
Double Precision Format 10-22
Complex Format 10-23
Double Complex Format 10-24
Logical Format 10-24
Short Logical Format 10-24
Character Format 10-25

11 Porting Code

Data Types 11-2
Data Alignment 11-2
Floating Point Representation 11-3
Machine Representations: IEEE Floating Point 11-4
Error Conditions During Floating Point Operations 11-5
Trapping Arithmetic Exceptions 11-6
Performance By Data Type 11-7
Cray Directives 11-8
CDIR$ IVDEP 11-8
CDIR$ NORECURRENCE = N 11-8

CDIR$ NOVECTOR = N 11-8
CDIR$ VECTOR 11-9
Stardent 1500/3000Fortran Syntax 11-9
Fortran 77 Extensions 11-9
Common Porting Questions 11-11
About DCL Define 11-11
About Async I/O 11-11
About Polling Devices 11-11
About Interfacing C and Fortran 11-14
Inline Expansion 11-14
Inline Functions 11-18
About FFT Support 11-21
One Dimensional FFT 11-22
Two Dimensional FFT 11-23
About Performance 11-23

12 Library Functions

Asynchronous I/O 12-1
Access To Command Line Arguments 12-2
Date And Time 12-2
Filing System Control 12-2
Random Number Generation 12-2
File I/O 12-3
Miscellaneous Functions 12-3
System Interface Functions 12-4

A Asynchronous Input/Output

B Using The Postloader

List of Tables

Table 2-1. Command Line Preprocessor Options 2-3
Table 2-2. Command Line Compiler Options 2-4
Table 2-3. Command Line Loader Options 2-5
Table 2-4. Examples Of Valid Expressions for #if 2-19
Table 2-5. Form of Compiler Options 2-40
Table 2-6. Fortran Compiler Options 2-42
Table 2-7. Suboptions for -standard option 2-46

Table 2-8. C Compiler Options 2-53
Table 3-1. Available Libraries for C 3-4
Table 10-1. CPU Registers 10-2
Table 10-2. Scalar Floating Point Registers 10-2
Table 10-3. Vector Registers 10-3
Table 10-4. FPU Instructions, by Mnemonic 10-4
Table 10-5. Integer Data Format (INTEGER*4) 10-20
Table 10-6. Short Integer Format (INTEGER*2) 10-21
Table 10-7. Real Format 10-22
Table 10-8. Double Precision Format 10-23
Table 10-9. Complex Format 10-23
Table 10-10. Double Complex 10-24
Table 10-11. Logical Data Format 10-24
Table 10-12. Short Logical Data Format 10-25
Table 10-13. Character Data Format 10-25
Table 11-1. BLAS Function Names 11-18

PREFACE

This manual is a guide for the Stardent 1500/3000 programming languages. It contains information and detailed explanations that describe tools for creating programs, how to use the Fortran and C compilers, how to debug programs, how to create more efficient code, as well as how to port and tune code for optimal operation on the Stardent 1500/3000.

Structure Of This Manual

Here is a brief description of the contents of each chapter and the appendices.

- Chapter 1 — *Creating And Maintaining Programs*
  Briefly describes certain tools that can be used to write and maintain programs for the Stardent 1500/3000.

  There are commercially available reference books which provide detailed tutorials about the various topics covered in this chapter. However, Chapter 1 is designed to provide a reference for these tools, as well as a brief introduction to interest the user in locating and using other tutorials on these topics for more information.

- Chapter 2 — *Using The Compilers*
  Describes the C and Fortran compilers.
- Chapter 3 — *Using Libraries and Linking*
  Describes the linker and methods for maintaining libraries of code.
- Chapter 4 — *Methods For Debugging Code*
  Describes the tools available for debugging.
- Chapter 5 — *Running the Debugger*
  Provides a tutorial for *dbg*.

- Chapter 6 — *Optimization*
  Covers the basic theory of vectorization and parallelization emphasizing the fact that programs may have to be written in a certain form to allow the compiler to take advantage of the architecture of the machine.
- Chapter 7 — *Efficient Programming*
  Expands on the basic theory and shows that certain programming constructs run more efficiently than others.
- Chapter 8 — *Explicit Parallel Programming*
  Delves more deeply into exploiting opportunities for using multiple processors and shows the system functions and directives that aid in a user-directed parallel programming effort. Both Fortran and C directives are described.
- Chapter 9 — *Tuning Code*
  Describes tools and techniques that you use to tune code for faster operation. The chapter concentrates on the profiler.
- Chapter 10 — *Language Interfacing*
  Describes data layouts in memory and the parameter passing conventions employed by C and Fortran, thus enabling the creation of programs that employ multiple languages if necessary.
- Chapter 11 — *Porting Code*
  Describes moving programs to the Stardent 1500/3000 from other environments, including data types and precision of the floating point values. This chapter also describes the compiler directives that control vectorization and parallelization and directives provided for compatibility with other programming environments.
- Chapter 12 — *Library Functions*
  Provides a brief description of Fortran library functions.
- Appendix A covers Asynchronous I/O, as implemented on the Stardent 1500/3000.
- Appendix B describes how to postload executable files from the Stardent 1500 to the Stardent 3000.

Related Documentation

More information on Fortran, C, and the UNIX Operating System can be found in the following Stardent 1500/3000 manuals:

- *Programmer's Reference Manual, Vol. 1*

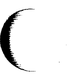
- *Programmer's Reference Manual, Vol. 2*
- *Fortran Reference Manual*
- *Commands Reference Manual*

Using This Manual

Beginning programmers use the manual from the beginning, learning about the tools available for creating programs and proceding through the rest of the material.

System programmers, that is, those who need only to port existing programs to the machine with few concerns about efficiency can begin with the compiler and debugger sections and ignore the rest of the book.

Experienced programmers who are already well versed in creating programs and who are greatly concerned about efficiency of their code should probably begin with the compiler section and proceed to use the rest of the sections as well.



CREATING & MAINTAINING PROGRAMS

CHAPTER ONE

This chapter provides a tutorial for *vi,* a full screen visual editor. There are certain other tools that fall into the category of program maintenance tools. Among these tools are:

- *SCCS* - a Source Code Control System
- *make* - a file maintenance program

There are other programs that are supplied as part of the standard system software. Though the other tools are not covered here in detail, you can find a brief description of some of them at the end of this chapter. Also included are references to commercially available texts that you can use to learn more about them.

Learning vi

This section introduces *vi,* the terminal oriented editor. Instead of working on a line at a time, as with some editors, *vi* lets you see and edit an entire screen at a time. This section contains:

- a segment that explains many basic operations; enough to perform the most often used editing tasks. This segment is titled *The Basics Of Editing With Vi.*
- a segment that explains some of the more advanced features. This segment begins with at the title *Ex Commands.*
- a quick reference segment for *vi* commands.

Finally, at the end of the chapter, the bibliography refers you to commercially available textbooks and other references in which you can find additional tutorial or reference material for *vi.*

The Basics Of Editing With vi

This section provides you with enough information to do the most basic editing tasks. As you become more adept, (and perhaps more demanding), you can proceed at your own pace, using other published documents about the many features that *vi*

provides.

When you edit a file, the entire file is copied into memory at the same time and is accessible to *vi* as the contents of the current *edit buffer*. Any changes made to the edit buffer are not written to the file unless you explicitly issue a write command.

For editing text files, the basics you'll want to know are:

- how to start the program
- how to move the cursor
- how to enter text
- how to delete text
- how to save your work
- how to exit

This section begins by answering just those basic questions, then describes a few, more advanced topics.

Note: for all of the examples in this initial basic section, the notation **ESC** is used to refer to the ESC key on the keyboard. This *escapes* from Input mode into Command mode.

How To Start vi

This tutorial assumes that you are running under the C shell. Assume that you want to create a new text file in your */tmp* directory. First change directories to */tmp*. Then copy a file into this directory that can be used for practice.

```
prompt> cd /tmp
prompt> cp ~/.login /tmp/sample
```

To start vi, simply type 'vi sample' at the prompt.

```
prompt> vi sample
```

If the system responds 'vi: command not found', it means that your command search path does not include the location at which the *vi* command is stored.

Set the path to include */usr/ucb* as follows:

```
prompt> set path=($path /usr/ucb)
```

then start *vi* as shown above.

If you start *vi* without specifying a file name, your screen becomes entirely blank, with the cursor positioned in the upper lefthand corner and the left edge of the screen has a tilde (~) as the first character of each line. This indicates that you are working on a new file, containing zero characters. The tilde lines are simply there to fill out the screen indicating that there is nothing available from the file to fill the screen area.

Note that there are many other options that you can specify on the command line when you start *vi*. See the man-page for *vi* for an explanation of those other options.

Two Modes Of Operation

Vi has two modes of operation: *Input mode* and *Command mode*.

In Input mode, everything that you type is inserted into the file at the position of the cursor or is used to overwrite what might already be onscreen. You escape from Input mode into Command mode by pressing the **ESC** key.

In Command mode, any keys that are typed perform some action that either moves the cursor or has some effect on the contents of the file. Many Command mode keys begin Input mode. Thus it is very easy to move between the *vi* modes. If you are unsure of the mode that *vi* is in, you can press the ESC key even though *vi* may already be in Command mode. All that happens is that you terminate a command that you may have partially entered, and the terminal may beep or flash.

How To Move The Cursor

From Command Mode, use the arrow keys on the far right side of the keyboard to move the cursor. Other methods of moving the cursor are outlined in the *vi* reference section of this chapter.

How To Enter Text

You enter the text by entering Input mode. In the topic *Entering Input Mode* below, many different ways to start Input mode are described. However the most common way, from Command mode, is to move the cursor to the position that you wish to enter text, and type an *i*. As noted above, you exit from Input mode into Command mode by using the **ESC** key.

Try the following example:

```
itextESC
```

This enters the word **text** at the current cursor position. In particular, the **i** puts *vi* into Input Mode, the **text** types these characters onto the screen at the current cursor position, and the **ESC** key press takes you back to Command Mode.

To simply add a new blank line at the current cursor position, just enter Input Mode and press the ENTER key.

Two other very common ways of entering Input mode are to begin Input immediately following the current text character.

This is a good time to point out that *vi* recognizes commands on a case-sensitive basis. That is, an upper case letter entered from Command Mode often does something different than that same letter entered as lower case. Most of the commands that are described in this chapter are the lower-case commands. You can find the description of the mapping of the upper-case letters in the summary table of *vi* commands near the end of the chapter.

The Command mode key for appending something immediately after the current cursor position is *a*; to enter text at the end of the current line, use an *A* instead (the **A** command automatically moves the cursor to the end of the current line and appends following the last character on that line.

To try this, move the cursor to the first **t** of the word **text**. Then enter:

```
aest tESC
```

The result is: *test text*

To add to the end of the line containing this phrase, use the *A* command:

```
A added to the endESC
```

The result is: *test text added to the end*

If you make a typing mistake when you are in Input mode, you can use the terminal's BACK SPACE key to move the cursor backward in the line. You can then type over the text to correct it.

To add a new blank line at the end of the edit buffer, use the cursor keys (or any other cursor move instruction) to move the cursor as far toward the end of the file as it will go, then enter the **A** (append to end of line) command. Anything you type will be entered there, beyond the current end of the buffer, establishing a new end position in the process.

How To Delete Text

There are several methods to delete text. This section only covers a few of them. Later in this chapter, all of the text deletion methods are shown in detail.

Delete a character by moving the cursor onto the character you want to delete and use the *x* key. Delete from the current cursor position to the end of a word by entering the command *dw* (delete word). Delete from the current cursor position to the end of a line by entering the command *d$*. Delete an entire line by using the command *dd*.

Undoing A Command

You can undo a change that a command has made. Some word processors allow you to undo several sequential changes, one after another. Vi only allows a single level of undo. That is, if you undo something twice, you undo what you undid before. In other words, a second undo is treated as a redo. Once you are safely back in Command mode, pressing the *u* key undoes whatever command you had most recently entered. This is often useful for beginners to know.

Restoring A Line That Was Most Recently Deleted

When you use the undo command, any text that was deleted is replaced in exactly the same position from which the deletion occurred. You could use a text delete command (such as, perhaps, a line-delete) to transport the deleted text from one part of the edit buffer to another part.

When a line is deleted, for example, using a *dd* command, that line is actually copied into an internal memory area called the *unnamed buffer*. The contents of the unnamed buffer can be inserted into the edit buffer either above or below the line in which the cursor is currently resting. The *P* command puts the text above the cursor; the *p* command puts the text below the line that contains the cursor. Either the *P* or the *p* command copies text out of the unnamed buffer into the edit buffer. This means that you can copy the same text into several places within the edit buffer if you wish.

Each time you use a delete command of some kind to delete a block of text, that new block of text replaces the previous block that you deleted as the **most recently deleted block**. Up to 9 previous deletions are available for retrieval, as explained later in this section (Beyond the most recent deletion, something more than just the P or p command is required. See the explanation of the use of the numbered buffers.

How To Save Your Work

When you make changes to the edit buffer, the changes you make do not affect the file you are editing until you actually tell *vi* to write those changes to the file. From Command mode, you write the contents to a file by using the *:w* command. The *:w* command takes two different primary forms:

:w If you don't specify a file name, *vi* assumes that you wish to write to the file that you had specified on the command line when you started *vi*.

Example:

```
prompt> vi testfile
...
...
:w    (writes into testfile)
```

:w newfilename

To write into a different file name. *Vi* remembers the name of the original file that you specified on the command line. So that original file name remains the default. If you again give a *:w* command without specifying a file name, the default file name is again used.

Sometimes you receive a message that the file exists and you should use *:w!* to overwrite it. The command *:w!* is the write command with the exclamation point as an option that says you really mean to overwrite the current contents of a file.

If you discover that you are working in read-only mode, sometimes even the command :w! does not respond and you should choose another file name or perhaps even a different directory in which to save your file. Here is an example:

```
:w! /tmp/myversion
```

Exiting vi

You exit *vi* from Command mode by using the *:q* command. If you have not saved your most recent changes, *vi* refuses to quit, and tells you to use *:q!* to exit if you do not want to save your changes.

Now that you have seen some of the most basic aspects of using vi, the remaining *vi* section provides details about the rest of vi's commands. For completeness, the command reference section that follows also includes the commands that were previously discussed.

Ex Commands

The *vi* program is a screen based editor based on a line-editor named *ex*. *ex* commands can be made more descriptive than *vi* commands. Some of the *ex* commands duplicate an action that is assigned to single letters in *vi*.

All *ex* commands begin with a colon (:). The advantage to using *ex* commands in place of their *vi* single letter equivalents is that when you type a colon in Command Mode, the cursor jumps to the bottom line of your editing window and you have the chance to see the command as it is being formed. There is less chance of an error if you can examine the entire command before you execute it by pressing the ENTER key.

There are often different methods of accomplishing the same task (such as deleting lines, copying blocks of text and so on). Rather than provide a separate *vi*-section and *ex*-section, each telling you how to accomplish a task, this section merges the two types of commands together so that you only go to one place to find several ways of doing a particular job. You can then choose the method that is most comfortable for you.

Generic Form Of An *ex* Command

An **ex** command takes the form:

: *[address] [command] [!] [parameters] [count]*

All of the parts of this generic command are optional, so if you enter *ex* by typing the leading colon from vi's Command mode, you can just press return. This enters a blank line as an *ex* command which means that there were none of the optional command components. The only result is that your cursor is moved down one line.

The *address* parameter can take many different forms. Note that all examples here are shown with the command *d* which is shorthand for *delete*.

- numeric or a numeric pair to indicate either a specific numbered line or a range of lines inclusive of the line numbers specified.

  Examples:

  ```
  :34d
          deletes line 34
  :12,17d
          deletes lines 12-17 inclusive
  ```

- a period (.) as one of the address parameters can be used to indicate the current line.

  Examples:

  ```
  :.d
          deletes the current line
  :15,.d
          deletes from line 15 to
          the current line inclusive
  ```

- a cursor move relative to the current cursor location can also serve as an address for the current command.

  Example:

  ```
  :.,.+12d
          delete the current line and all
          lines from here to 12 lines ahead
          of the current line, inclusive.
  ```

- a cursor move command that moves to a marked line can also serve as the target address. The topic *Marking and Returning To A Position* shows how to mark specific line numbers and how to use them to specify an address for a command.

[command] is a full word or an acceptable abbreviation. The abbreviation for the commands is provided with each command description. For example, the command

```
:1,10de
```

takes the form :*[address][command]* and deletes lines 1 through 10 of the edit buffer.

[!] a line or a range of lines is to be passed to the shell to use as the input to a command. The output of the command replaces that range of lines in the edit buffer. Example:

```
:1,32!sort
```

sends lines 1–32 to the shell, executes the *sort* command, and returns the output of sort to the edit buffer, replacing the original lines entirely. Be certain to know how shell commands function before you try this command. However, if the command does not do what you had expected, you can always undo the command before you do anything else.

[parameters] are parameters (if any) for the command. The most common parameter is the name of the buffer into which lines of text should be placed or from which lines are to be copied. Example:

```
:1,12de e
```

deletes 12 lines into buffer *e*.

[count] indicates how many lines are to be involved in this command. Count is not used when the address portion specifies the line range. For example, the same effect could have been obtained by placing the cursor on line 1 of the edit buffer and giving the command *:de e 12*.

Entering Input Mode, In Detail

When you first start the program, *vi* begins in Command mode. You enter Input mode using any of the following Command mode key commands:

a append—Insert text after the character on which the cursor is now resting.

A Append-to-end - Insert text at the end of the line in which the cursor is located.

i insert—Insert text to the left of the character on which the cursor is resting.

I Insert—Insert text to the left of the first non-whitespace character on the line. This means that if the text begins indented from the left edge, and the indent consists of tabs or spaces, the insert begins where the first real character occurs on the line (the leftmost text character position).

o open the line—Push all text lines, beneath the one the cursor is resting on, down one position (opening up a new line on which to type) and begin to insert characters at the leftmost position of this new line.

O Open the line—like *o*, but put the new line above where the cursor is residing and insert text on that new line.

c change—the change command is usually associated with an object. That is, you specify, by means of a cursor move, what the extent of the change will be, and the text that exists between the current position of the cursor and the place to which the cursor move command would move the cursor is treated as an object. Typical objects are a word (w), a word to the immediate left of the cursor, that is, backwards from the cursor (b), a character to the left of the cursor (h), to the right of the cursor (l). The topic *Deleting Objects In Vi* lists additional cursor move commands that can be treated as though they define objects.

Examples:

```
cw - change a word
c$ - change to the end of this line
c0 - change from start of this line to here
c( - change from start of sentence to here
```

Note that many vi commands accept a numeric parameter to say how many times a command should be executed. For a change command, the numeric parameter says how many objects to change:

```
4cw - change the next four words
```

In this example, a dollar-sign appears in the edit buffer replacing the last character of the third word following the word in which the cursor is resting (change encompasses 4 words total). If you type fewer characters than occupied now by those four words, all of the excess characters are deleted when you press the ESC key to exit Input Mode, leaving only the characters you typed.

If you continue to type beyond the end of the object for which a change is specified, *vi* enters Input mode, thus continuing to accept your input instead of halting when the change to the object has been completed.

C Change to end of the current line—this is shorthand for a *c$* command which means overtype from the current cursor position to the end of the current line. If you type past the end of the line, *vi* enters Input mode which is explained for *c* above.

s substitute for the character on which the cursor is currently resting—if you type more than one character after an *s*, *vi* enters Input mode in order to accept the other characters (until Input mode is terminated).

S Substitute from where the cursor is now to the end of the line—as with C or *c$*, if you type past the end of the line, you are in Input mode.

R Replace all text—on the current line, character by character, as you type, starting from the current character position. If you type past the last character on the current line, *vi* enters Input mode.

Moving the Cursor

Character Positioning

There are many ways to move the cursor in *vi*. The most basic ways are up, down, right, and left. The designers of *vi* made this very easy for touch typists by assigning these motions to the home keys for the right hand:

h Move left one character (left arrow key).

j Move down one character (down arrow key).

k Move up one character (up arrow key).

l Move right one character (right arrow key).

Note that on the Stardent 1500/3000, the arrow keys are acceptable alternatives to the h, j, k, and l keys.

More Positioning By Character

Additional positioning by characters is also possible.

0 Move to the beginning of the line.

$ Move to the end of the line.

BACKSPACE
Move back one character position.

SPACE
Move forward one character position.

^ Move to the first non-tab non-whitespace character on the line.

| Move to a specific column. Example:12| moves to col 12)

% If the cursor is on left or right parentheses, or on a left or right curly brace, this command moves the cursor to the matching paretheses or curly brace, moving either forward or backward to find it. If there is no match, the cursor does not move and the terminal bell is rung.

f*x* Move forward onto the next character *x* on this line

F*x* Same as *fx* but move backwards to the next character *x*.

t*x* Move forward to the character just to the left of the next character *x* on this line.

T*x* Same as *tx* but move backwards to the character just to the left of the next character *x*.

; Repeat the last *f*, *F*, *t* or *T* command.

, Repeat the last *f*, *F*, *t* or *T* command as the inverse of the command performed. That is, if the last command was an *f*, repeat it but substitute an *F* in its place so as to move the cursor in the opposite direction.

If the search fails for *f*, *t*, *F* or *T*, the cursor does not move at all and the terminal bell is rung.

Words, Sentences, Paragraphs

The cursor may also be positioned on a word, sentence or paragraph basis:

w Move to the first character of the next word.

b Move to the first character of the previous word .

e Move to the end of the current word.

) Move to the first character of the next sentence.

} Move to the first character of the next paragraph.

(Move to the first character of the previous sentence.

{ Move to the first character of the previous paragraph.

Moving To Explicit Onscreen Positions

You can quickly move the cursor to a specific position on the screen such as the top, bottom, or middle of the screen:

H move to the top line on the screen

M move to the middle line on the screen

L move to the bottom line on the screen

Scrolling Through The Edit Buffer

You can ask *vi* to scroll through the edit buffer by using the CONTROL (sometimes labeled *Ctrl*) key in combination with other keys. The CONTROL key with other keys is represented by a ^ character preceding the key. For example, a CONTROL-F is

represented as ^F. Note that though the notation used here shows an upper case letter, the actual letter as entered is lower-case, along with the control key. Thus the notation refers only to the actual keycap symbol.

Here are the scrolling moves:

^F scroll one page forward in the buffer

^B scroll one page backward in the buffer

^D scroll one half page forward in the buffer

^U scroll one half page backward in the buffer

You can move the cursor to the first nonwhitespace character on the previous or next line as follows:

– Move to the first nonwhitespace character on the previous line.

CR Same as –, but move to the next line instead of previous (CR stands for the ENTER key on the keyboard)

Moving To A Position Defined By A Character String

In this section, you see that the slash mark (/) and the question mark (?) can be used to specify a pattern of characters for which a search is to be conducted.

You can tell *vi* to find the first occurence of a character string and move the cursor onto the first character of that character string, or onto the *n*th character before or after the first character of that string:

/xyz search forward for character string *xyz*

/xyz/ this command begins to work as soon as you press the ENTER key. If the terminating slash is not used, the search assumes that all characters entered prior to the ENTER key are part of the search pattern (that is, as though the terminating slash has actually been typed prior to the typing of the ENTER key).

?xyz search backward for the pattern *xyz*.

?xyz? search backward for the pattern *xyz*. For this form, the terminating question mark is optional.

/xyz/+*n* same as /xyz/ but go n characters beyond the start of *xyz*

/xyz/-*n*	same as /xyz/ but go n characters before the start of *xyz*
?xyz?+*n*	a backwards search version of /xyz/+n
?xyz?-*n*	a backwards search version of /xyz/-n
n	repeat the most recent / search
N	same as *n* but reverse the search direction

The pattern defined by *xyz* is what is called a *regular expression*. This means that instead of being able to search for the literal occurence of an exact string of characters, you can conduct a search for an arbitrary character sequence where special character patterns determine rules for substitution of real character strings.

For example, if you wish to search for the words *farther* and *further*, you could specify the pattern search as: */f.rther*, where the period (.) represents **any character**. As another quick example, consider searching for the word **the** at the end of a line. The pattern specification for this is: */the$*, where the dollar sign is the special character that represents the end of a line. See the topic *Regular Expressions* for more details.

Pattern match searches normally wrap around the buffer. That is, if you begin a forward search near the end of the buffer, but that pattern occurs somewhere towards the beginning of the buffer instead, *vi* searches from the current cursor position towards the end, then restarts the search at the beginning of the buffer. You can use one of the *vi* options (shown later in this chapter) to disable this automatic wrap feature.

Moving To A Specified Line Number

You can move to a specific line number by entering that line number followed by the letter *G* (It must be upper case). For example, enter:

34G

This moves the cursor to line 34. The letter G alone causes a move to the end of the file.

Marking and Moving To A Position In The File

You can mark a position in the file and *vi* remembers that position. Later you can move to that position, or having marked that position, you can specify it as the object for a delete or a copy command.

To mark a position, just issue the *m* command, from Command mode, along with a letter, a-z, that you wish to use to identify that position in the file.

m*n* Mark a position in the file, where *n* is a letter from a-z.

To quickly return to that position, no matter where you are in the edit buffer, use the command named *tick*, a single-quote mark (') or backtick (`). Tick moves the cursor to the line at which the mark was placed; backtick moves the cursor to the exact character position at which the mark had been entered. Example:

> Mark a position as position *a* (issue the command: **ma**). Move the cursor anywhere else in the file. Now return to that position by issuing the command: **'a**.

Because this command is a cursor move, it qualifies as an *object* for commands that demand a target address.

Buffers In vi

When you tell *vi* to edit a file, *vi* does not modify the contents of that file until you explicitly tell it to write to that file. *vi* opens up a buffer (an area in memory and on disk) that it uses to show the changes to the file. These changes only become part of the file when you issue the *:w* (write), or ZZ (write and quit) command. The buffer in which the editing is taking place is called the edit buffer.

There are several additional buffers that *vi* maintains:

- An *unnamed* buffer holds the most recently deleted block of text when a delete command has been issued.

- A set of 9 *numbered buffers* hold the additional blocks of deleted text so that a deletion up to 9 delete-commands ago can be retrieved if desired. These buffers are maintained as a push down stack. This means that if you issue three deletes, the most recent one is in the unnamed buffer, the next most recent is in buffer 1, and the next most recent to that is in

buffer 2 (and so on). The nth most recent delete could be retrieved by accessing buffer number *n* where n is less than 9. Beyond that number, the block of text is permanently deleted.

- A set of 26 *named buffers* (a-z) can be used to hold blocks of text while the edit session is active. Later in this chapter, some examples of filling and using the edit buffers are provided.

Deleting Objects In Vi

You can delete single characters or delete *objects* in *vi*. An object is a block of text that is defined by a cursor move command of some type. Deleted text goes into the unnamed buffer. Each new delete command causes the unnamed buffer contents to be pushed into numbered buffer 1, buffer 1's contents to be pushed into buffer 2 and so on, with the most recently deleted text in the unnamed buffer. Here are some commonly used delete commands.

x delete the character that the cursor is resting on.
d*n* delete the object represented by *n*.

Objects include the following:

Character	Defines the Object	Example
w	a word	dw - delete a word 4dw - delete 4 words
b	a word to the left of the cursor	db
d	the line that contains the cursor	dd
$	from here to end of the line	d$
)	from here to end of sentence	d)
}	from here to end of paragraph	d}
{	from here to start of paragraph	d{

Commands given here might be better treated as command shortcuts. They offer no feedback until the command is completed. The section titled *Deleting To, Copying To, and Appending To Named Buffers* describes a delete command that does provide immediate feedback as you enter the command.

Any other cursor move command can also be treated as the object that the delete command operates on. For example, the object *n* may also be a position that you have marked.

d'*n* If *n* is a position that has been marked with the mark-command (e.g. *mn*), then this command tells *vi* to delete the text between where the cursor is located now, and where the mark *n* had been placed.

Experiment on your sample edit file if you wish. For example, try deleting from the current cursor position to the first character of the next sentence — answer: the command would be: **d)**

Moving Text Blocks

Deleted text can be retrieved from the unnamed buffer by *putting* it somewhere in the edit buffer, either above or below the current cursor position. Simply reposition the cursor, then type:

p to put the item immediately below the line in which the cursor currently resides.

P to put the item immediately above that line in which the cursor currently resides.

Examples:

8dd Delete 8 lines, including the line the cursor was resting in, and move the block of text into the unnamed buffer as the most recently deleted item.

P Put the contents of the unnamed buffer just above the current cursor position. If the cursor did not move between the *8dd* delete command and the *P* command, this has the same effect as an undo.

Moving Text Blocks By An Alternate Method

In the above forms of moving, the blocks of text are deleted into the unnamed or named buffers for later use. You can directly move text from one place in the file to another by using the *move* command. It takes the form:

:*[address]***move** *whereto*

(Notice that this *move* command is a colon-command.)

This command moves the lines within the addressed range to the line just following the line number specified in *whereto*, which may be zero to represent the beginning of the file. The move command can be abbreviated to *mo*. The cursor moves to the last line of the moved block.

Here are examples:

:mo 0	Move the current line to become the first line in the file (no address range is specified, therefore it operates on the current line.)
:'a,'bm $	Move the block designated by marks *a* and *b* to the end of the edit buffer.

Copying Text Blocks

In *vi* terminology, copying an object or a range of lines is called *yanking* the object. Again an object is designated by a cursor move command of some type. The commands for copying objects are:

y*n* copy the object *n*. See the delete command explanation for the types of objects that can be copied.

y*y* copy an entire line (into the *unnamed* buffer for later use.)

y'*n* If *n* is a position that has been marked with the mark-command (that is, *mn*), then this command tells *vi* to copy the text between where the cursor is located now, and where the mark *n* had been placed.

Examples:

10yy	Copy this and the next 9 lines into the unnamed buffer. Now move the cursor to the position where the lines are to be inserted.
p	Put the contents of the unnamed buffer at a position immediately below where the cursor is located now.

As with the delete command, copying text to the unnamed buffer can be specified with any form of cursor move command defining the terminating position of the copy (yank).

If you are editing multiple files, note that the contents of the named and numbered buffers is preserved across file-edit boundaries. That is, you can still insert from a named buffer even though you began to edit a new file. But the unnamed buffer's contents are lost when you switch files.

Copying Text Blocks By An Alternate Method

In the above forms of copying, the blocks of text are placed in the unnamed or named buffers for later use. You can also directly copy text from one place in the file to another by using the *copy* command. It takes the form:

:*[address]* **copy** *whereto*

This command copies the lines within the addressed range to the line just following the line number specified in *whereto*, which may be zero to represent the beginning of the file. The copy command can be abbreviated to *co* or may be expressed by its synonym *t*. The cursor moves to the last line of the copied block of text at the destination location. Here are examples:

:co 0 Copy the current line and write it as the first line of the current edit buffer.

:'a,'bco $ Copy the lines between mark *a* and mark *b* to become a new segment of text at the end of the file.

Note that the copy command modifies the contents of the unnamed buffer (it is empty following the copy operation).

Changing Text

There are many ways to change text that you have already typed. Here are just a few:

r*n* Change only the character on which the cursor is currently resting. The command is *r*, and *n* represents the character to which the text is changed. For example, typing **r0** changes the current character to a zero.

R<text>ESC
An uppercase R (remember it as an abbreviation for "Replace") begins a change that lets you overtype existing text for as many characters as you wish. If, as you are typing, you go past the end of existing text on a line, *vi* enters Input mode, allowing you to continue for as long as you wish. Pressing the ESC key terminates Input Mode.

cn The *c* is the change command, and *n* represents some cursor move command that defines the extent of the change you

want to make. A dollar-sign ($) appears in the edit buffer at the ending point for your change. When you wish to terminate this command, press the ESC key and all text from the cursor position to the terminating dollar sign disappears, leaving only what you typed as changes. Here are a few examples:

- **cw** Change from the cursor to the end of the current word.
- **c$** Change to the end of the line.
- **c)** Change from here to the beginning of the next sentence.

Changing Text By An Alternate Method

The alternate method for changing text is a colon command, *:change*.

: *[address]***change***[count]*

This command uses an optional *[address]* to specify the range of lines that is to be replaced by the text you type. If *[address]* is not included in the command, then the *[count]* value, if provided, specifies how many lines (including the current one) are to be affected by the text you type. If neither *[address]* or *[count]* is provided, *ex* assumes you wish to change only the current line. A dollar-sign ($) appears at the end of the text block, and the cursor is positioned on the first character of the text block that you have specified.

When you finish typing, press **ESC** and the replacement of that block will be complete. If you type more text than is in the marked block, *vi* enters Input Mode allowing all of your text to be entered.

Using Numbered Buffers

Numbered buffers hold the 9 most recently deleted blocks of text and operate as a pushdown stack, along with the unnamed buffer. As an example, if there is a deleted line currently in the unnamed buffer, and another line is deleted, then the first line is pushed into numbered buffer 1, and the unnamed buffer now has the most recently deleted line or block of text. The former contents of buffer 1 are pushed into buffer 2, the contents of 2 goes to 3 and so on, for a total of 9 numbered buffers.

To reinsert any blocks of text from these buffers, you use the put command (*p* or *P*) as follows. Assume you want to restore something that you deleted three deletes ago. The unnamed buffer contains the most recent deletion, Buffer 1 the second most recent, and buffer 2 contains that third most recent deletion, the one you wish to restore. Move the cursor to the postion at which to insert the text and type:

```
"2p
```

The double–quote character says that you are referring to a buffer. The *2* is the number of the buffer that you are using. Finally the *p* is the command to put the contents into the edit buffer.

The contents of all of the numbered buffers remains the same until the next delete happens, even across edit-file boundaries.

An interesting trick is possible using *vi*'s undo command. Lets say you know that what you want to insert somewhere was deleted recently (within the last 9 block deletes), but you are not quite certain which one of those deletes it was. You can use the repeat command (.) feature, along with undo, to view the contents of the numbered buffers sequentially as shown in the example here. First, issue a command that inserts the first of these numbered buffers:

```
"1p
```

If this is not the one you want, issue the command

```
u.
```

This has the same effect as though you had typed *"2p*. Internal to *vi*, the effect is to increment the buffer number, undo your original put, and try it again with the next numbered buffer.

If you wish to try this command, create a text file that contains the following lines:

```
9th deleted line
8th deleted line
7th deleted line
6th deleted line
5th deleted line
4th deleted line
3rd deleted line
2nd deleted line
1st deleted line
Here is where to put the stuff
The bottom line
```

Place the cursor somewhere in the first line, then enter the following command characters:

```
dd........
```

(dd followed by 8 periods) This fills all 9 delete buffers, one with each sentence. Now position the cursor in the line containing the word *Here*, and issue the command:

```
"1p
```

It brings back the line *1st deleted line*. Now issue the command:

```
u.
```

Each time you issue this command, the current deleted line disappears and is replaced by the next one in sequence. Whatever was the deleted block, in each case, appears in response to this command. The deleted block might contain a word, a line, or a large block of text. The only disadvantage to using the double-quote commands is that there is no user feedback provided until you complete the command. However, you can always use undo if you discover you have made a mistake.

Using Named Buffers

Named buffers, like numbered buffers, retain their contents even when a new file is to be edited. Thus named buffers are good for moving blocks of text from one file to another.

Filling a numbered buffer is done automatically as noted above, as a result of performing a delete command. Filling a named buffer must be done explicitly by yanking or deleting blocks of text and specifying to which buffer the yank or delete is to go.

There are 26 buffers, named a-z. You refer to a named buffer with the double-quote sign while in Command mode.

You copy items TO a named buffer by yanking or deleting something. You copy items FROM a named buffer by putting the named buffer's contents into the edit buffer. Here are a couple of examples that show filling of a named buffer. There are many examples of copying and deleting to named buffers in the next section (*Deleting To, Copying To, and Appending To A Named Buffer*).

Examples:

:de j	Delete the current line into buffer j.
:'a,'bya k	Copy the range of lines from that line marked as line *a* to that line marked as line *b* into buffer k.

Deleting To, Copying To, and Appending To A Named Buffer

Copying Text Into A Named Buffer

Instead of using the command shortcuts (such as the *d* or the *y* command) to delete or yank (that is, copy) blocks of text to the unnamed buffer, you might choose to save a block of text for future use in a named buffer. You do this by using a colon command for the yank or delete, and specify one of the 26 available buffers (a-z) as the target into which the block of text should be placed.

Appending Text To A Named Buffer

Instead of selecting a buffer by specifying its name as a lower case letter, you can tell *vi* to *append* the newly deleted text or block of text to a named buffer by specifying its name in uppercase letters A-Z. The same buffer is used, only the type of operation on the buffer is different.

Commands That Delete To Or Copy To Named Buffers

Here are the commands that fill or append to named buffers. There are shortcuts for these same commands in the next subtopic, but this form provides immediate feedback (you can see what you are typing).

> :*[address]* **yank** *[buffer] [count]*
> :*[address]* **delete** *[buffer] [count]*

The yank command can be abbreviated *ya* or simply as *y*. The delete command can be abbreviated as *de* or simply as *d*.

[address] is optional and can be represented by an explicit line number or by the name of a marked line, along with the tick mark, such as 'a to refer to the line number marked as *a*. The range of lines between the line marked as line *a*, and the line marked as line *b* would be represented as an address with the notation 'a,'b. If *[address]* is not provided, the target of the command is the line in which the cursor currently resides. This address is also known by the name "." (that is, "dot").

[buffer] is the single character name of the buffer to use. Use a-z as a name to replace the current contents of the buffer with the new contents, or A-Z to append the new contents to the end of that named buffer.

[count] is the count of the number of lines to operate on. This is used only when the *[address]* parameter is missing or is specified as dot.

Examples:

:1,12ya b	Copy lines 1 - 12 into buffer b.
:ya c 12	Copy the current line and the next 11 lines into buffer c.
:15ya e	Copy line 15 into buffer e.
:'aya d	Copy the line marked as *a* into buffer d.

:'aya d 8	Copy 8 lines from the position marked as line *a* into buffer d.
:.,.+5de H	Delete the line on which the cursor rests and the next five lines and append them to the end of buffer h.

Command Shortcuts

The command shortcut for using named buffers is a double-quote mark, followed by the name of the buffer, followed by a delete or yank command of some form. Using this shortcut should probably be considered as something only for the advanced user because double-quote shortcuts on named or numbered buffers offer no feedback while you are typing in the command. Nothing appears on the bottom line to tell you that you are indeed hitting the right keys. If you feel you might have made a typing mistake along the way, you can either complete the command and undo something if there was an error. Or you can hit the ESC key a few times to exit the command you were typing and start over. Here are some examples.

"a3dd	Delete this and the next two lines, and put them into named buffer a.
"bdG	Use buffer *b*; the command is i*d*; *d* operates on an object, which, in this case, is represented by a cursor move to the end of the edit buffer. So, delete from here to the end of the main buffer and put that block of text into named buffer *b*.
"c10yy	Copy this and the next 9 lines into named buffer c.
"fd$	Delete from here to the end of the line and put the text into buffer f.
"jy3w	Use buffer *j*; *y* is the command;

3w is the object
yank this and the next two words.

"ny'a Use buffer *n;*
y is the command to use on an object
'a is the object. This means copy the text block between here and wherever the *a* mark has been placed.

Copying A Named Buffer Into The Edit Buffer

Once a named buffer has been filled, you can put the contents of that named buffer back into the edit buffer wherever you wish. Putting the contents into the edit buffer copies the contents of the named buffer, leaving it intact. The put command is specified as:

:*[address]* **put** *[buffer]*

The put command can be abbreviated as *pu*. Here are some examples.

:pu d Copy the contents of named buffer *d* to become lines immediately below where the cursor is now resting.

:0pu f Copy the contents of named buffer *f* to a position immediately below line 0. That is, the put block appears as the first lines in the edit buffer (specifying address zero puts things before the first actual numbered line in the buffer. The cursor moves to the position of the last line that has been copied.

Command Shortcuts

Using the double-quote shortcut, you can copy the named buffer's contents by putting the contents from the buffer below (*p*) or above (*P*) the current cursor position. Here are some examples.

"ap Put the contents of buffer *a* below the current

position of the cursor.

"nP Put the contents of buffer *n* above the current position of the cursor.

Moving A Block Of Text From One File To Another

Here is a more complete example that shows how a block of text can be moved by using a named buffer. Suppose that you have two files, named *file1* and *file2*. There is a block of text in file1 that should be moved to file2. The block of text begins with the words *For all* starting at the beginning of the first line of that block, and this sequence, in that file, does not exist anywhere else. The block to be moved ends with the words *anywhere else*.. Here is a sequence of commands, including the command that starts *vi* itself, that accomplishes that move. There are two columns to this example. The left column shows the command that is entered. The right column is a running commentary about the command and its effects.

prompt> *vi* file1	Start the program
/^For all	Find the start of the block
ma	Mark and remember this as position **a**
/anywhere else.	Move to the end of the block
:.,'ade b	From marked position *a* to the current cursor position, delete (*de*) the text into buffer *b*.
:w	Write the current file, with this block missing.
:e file2	Edit file2 without leaving *vi* so that the named buffer is still intact.
/Insert That Text Here	Find the position to put that block
:pu b	Put the block there.

Regular Expressions

Pattern searching is done with what is called *regular expressions*. Certain characters are interpreted as control characters by *vi*. When forming patterns, these characters are not taken to represent the character itself, but rather a condition. For example, you can search for something that occurs only at the beginning of a line, or at the end of a line. You can search for a character that is within a particular group of characters, or substitute any character for a specific character position. This section defines the regular expression pattern characters and the special meanings that each has. Several examples are provided to illustrate the use of these items.

To search for that special character, instead of treating it as part of a pattern match specification, you precede the character by a backslash (\). In other words, *vi* is told to take this character literally. The special control characters are listed below. There are two columns to this example. The left column shows the character that is entered. The right column is a running commentary about the effects that the character has on a regular expression.

^ The pattern should be matched only if it is at the beginning of a line.

$ The pattern should be matched only if it is at the end of a line.

Example: ^lonely$

matches the word *lonely* if it is the only word on a line with no spaces or tabs before it or after it. A line that matches has the first character of the word *lonely* as the first character on the line, at the leftmost position on the line and whose rightmost character is the last character on the line before a line feed.

. Match any single character

Example: f..ther

Matches further, farther, feather.

* Matches zero or more characters. of a specific type.

Example: ba*

Matches b or ba or baa or baaaaaa.

Example: first.*$

Matches the word first, followed by one or more occurences of any character proceding to the end of the current line.

[ ] Match any single character by examining whatever characters are enclosed in the brackets.

Example: [abc]

Matches a, b, or c. in a specific position.

Example: [Ff]ortran

Matches *fortran* or *Fortran.*

[a-d] Matches any single character from a choice of any character in the specified range. Both single characters and character ranges may appear in this specification.

Example: [a-zA-Z0127]

Matches any single lowercase or uppercase character, or any of the digits 0, 1, 2, or 7.

< Match the beginning of a word.
> Match the end of a word.

With either of these, you can mark the start and end of a word so that only a whole word (surrounded by whitespace or punctuation marks of some type) are matched.

Example: <other>

Matches *other,* but does not match an embedded *other,* such as in the word *bothered.*

Reading The Contents Of An External File or Command

You can add the contents of an external file to your edit buffer by reading it into the buffer. You use the read command to insert the complete contents of the file into the edit buffer immediately below the current cursor location.

:read *filename*

The read command can be abbreviated *re* or simply as *r*. The filename may be a complete path name or whatever minimal form of name that is required for *vi* to find the file.

It is sometimes necessary to read data that has been generated by executing a command. The read command is used for this purpose as well as reading text files. The syntax of this command is:

:read *!command*

For example, if you are documenting a Fortran program that has several modules, you might have a sentence in your edit buffer that says: *The program consists of the following major modules:* and the next few lines are to list the **.f** files which are in the current directory whose contents are being documented. You would enter the command:

```
:read !ls *.f
```

The following could appear on the line below the cursor:

```
file1.f     file2.f    file3.f
```

Temporarily Exiting Vi To Use Other Programs

You can exit your editing session by using the command ZZ to suspend operation of *vi* temporarily, returning to the shell, and *fg* to return to *vi*. Or you can begin a shell process from within *vi* by using the *shell* command (the format is *:shell*). When you finish with the shell, issue the command *exit* to terminate the shell and return to *vi*. Both methods keep the editing cursor at exactly the same position and preserve the state of your file. Before you perform an exit of this kind, it is wise to do a write command to save your work. Though the system generally preserves edit files in the event of a system crash, not necessarily all of the latest edits will be available in a save file. Your explicit save, however, will preserve all edits intact.

Editing Multiple Files

In the brief tutorial section at the beginning of this chapter, we described saving the edit buffer to a file. We'll repeat that information here so that all of the file-related commands are together.

:w Write into the file that was specified on the command line when *vi* was started

:w newfilename
Write into a different file name. The original file name remains as the default.

w >>filename
Append the current edit buffer to the end of a specified file name.

:w!
:w! filename
Overwrite an existing file if vi refuses in response to an normal write command (*:w filename*).

:w !*command*
Write the edit buffer as the input to an external command to be executed by the shell from which *vi* was invoked. Notice the distinction between *w!* and *w !command*; the placement of a space between the *w* and the *!* decides the interpretation of the command.

Note that all of the forms of write (*:w*) commands can be accompanied by an optional address parameter, that is,

:*[address]***w ...**

The *[address]* parameter can specify a range of lines to be written to the external file, or as the input for an external command. The default address is 1,$, which means the entire contents of the edit buffer.

:e filename
Edit a file by the name filename. This command works only if the buffer in which you have been working has not been changed since the start of the edit session.

:e! filename

Edit a file but tells *vi* to throw away any editing changes that have been made since the most recent save of the edit buffer and instead begin to edit the new file.

The significance of the *:e* (or *:e!*) command is that while *vi* is running, its environment (all of its internal settings and buffers) remains intact. The contents of all buffers, EXCEPT the unnamed buffer (described below) remain intact when the new file is visited. Thus it is possible to use the numbered buffers and the named buffers as a means of copying text from one file and inserting it into another file.

If you want to use *vi* to edit multiple files, such as *all of the Fortran files in the current directory,* then either specify the files explicitly when starting *vi,* or use a shell wildcard character to cause the shell to select all of the files that you wish to edit. You'll be able to edit them sequentially using the :n command as shown below.

Examples
```
vi test.f mystuff.f more.f
vi *.f
```

vi presents the files to you one at a time for editing. To switch from one to another (in sequence as you specified them individually, or in alphabetical order as the shell presents them when a wildcard is used), when you are finished editing, use the save command (*:w*) to save the edits, then the *:n* command to get the *next* file to be edited. If you have chosen not to save your edits, the *:n!* command throws away current edits and begin to edit the next file. When *vi* has no more files as input, it reports *no more files to edit.*

:n Begin to edit the next file. If you have not saved your work before you issue this command, *vi* reminds you that some edits have not been saved.

:n! Discard edits that have not been saved and go on to edit the next file.

At any time, you can find out how many files are being edited by asking *vi* to show you the list of arguments with which *vi* was called. If you used a shell *wild card character,* it is possible that there are many file names that are in line to be edited. (A pattern that includes a valid pattern matching specification for the shell will expand into possibly several file names to be edited.) The command to view the set of file names being edited is *args.*

:args List all files being edited in this session, enclosing the name of the the current file being edited in square brackets. This allows the user to see which file is to be edited next as well as to determine the name of the current file being edited. (An alternate method to obtain the name of the current file is ^G). Here is an example. Assume that the current directory contains the names file1, file2 and file3, and no other file names in the current directory match the pattern *fil**:

```
prompt> vi fil*
```

Then, when *vi* has begun, if you type the command *:args*, here is the response from *vi* on the bottom line of the display:

```
[file1] file2 file3
```

When you reach the end of the sequence of files (when :n tells you there are no more files to edit), you can ask *vi* to begin again, using the first file name in the list by issuing a *rewind* command.

:rewind
Finish editing the current buffer and restart with the first file in the argument list. This command may be abbreviated *rew*. If you have not saved the current buffer and get the message "No write since last change (:rewind! overrides)", enter *:rewind!* or *:!rew*. This discards the most recent set of changes and begins editing at the start of the argument list.

Repeating Commands

To repeat a command that you just entered, type a period (.). There is an example that uses a period several times to delete successive lines. The example is located in the section titled *Using Numbered Buffers*.

To perform a single command several times, precede the command with the number of times it is to be performed.

Examples:

Delete 10 lines	10dd
Delete 5 words	5dw
Change 3 words	3cw

Repeating works with most of the *vi* commands.

String Substitution

vi provides string substitution commands that take the form:

: *[address]*s/*[regexp_string]*/*[replacement_string]*/*[g]*

The command is **substitute**, and it may be abbreviated *s* as shown in the command template above.

[address] is optional. If there is no address specified, vi assumes that you mean to operate on the current line. Common address expressions are:

1,$ This address means all lines in the entire edit buffer.
% This notation is shorthand for 1,$
.,.+10 This entry means this line and next ten lines
'a,'b This address specifies the range of lines from the line marked *a* to the line marked *b* inclusive.

[regexp_string] is a string that forms a regular expression as described in the topic *Regular Expressions* above. By using a regular expression, you can replace occurences of several different source expressions with a single, corrected expression. For example, the command **:1,$s/[fF]ortran/FORTRAN/g** will replace all occurences of the word "fortran" or "Fortran" with the word "FORTRAN", no matter where they appear on the lines in the edit buffer.

replacement_string is the set of characters that you want to use to replace the text that matched the regular expression. The replacement string is optional, so you can replace the matched character string with nothing if you wish (deleting the strings that match your specification).

[g], if present, means "global". That is, make the substitution anywhere and however many times something occurs on any line within the addressed range in the edit buffer. The default is to replace the first occurence on a line only.

Examples:

1,$s/UNIX/Unix[tm]/g
Replace this anywhere that it occurs in the edit buffer and if it happens several times on a single line, replace them all.

'a,.s/^ [TABKEY]//

From the position marked as line a, to the line on which the cursor is resting, replace any tabs that occur in the first column of the line with nothing (undo a physical indent).

Which Line, and Which File

To determine the line number of the buffer and the name of the file that you are editing, type a CONTROL-G (^G). If you position the cursor on the last line of the file (by using the G command) and then use ^G, *vi* tells you the name of the file, the number of the last line, and the number of characters in the file.

Specifying Literal Characters

Sometimes it is necessary to embed literal characters in a file you are editing. For example, a line feed is a CONTROL-L. While you are typing, certain literal characters are taken by *vi* as commands. If you wish to insert a literal character, you can *quote* the character to tell *vi* to take it literally instead of as a command. The quote character is CONTROL-V. Anything preceded by CONTROL-V (except for a carriage return) is inserted literally in your file while *vi* is in insert mode. Example: **^V^L** inserts a CONTROL-L (a formfeed character). On the screen, it shows up as ^*L*, and if you move the cursor across it, you can see that it takes up only one character position in the file and in the file's edit buffer.

Vi Command Summary

Sample commands

<- | | ->	arrow keys move the cursor
h j k l	same as arrow keys
itextESC	insert text abc
cwnewESC	change word to new
easESC	pluralize word
x	delete a character
dw	delete a word
dd	delete a line
3dd	... 3 lines
u	undo previous change
ZZ	exit vi, saving changes
:q!CR	quit, discarding changes
/textCR	search for text
^U ^D	scroll up or down
:ex cmdCR	any ex or ed command

Counts before vi commands

Numbers may be typed as a prefix to some commands. They are interpreted in one of these ways.

line/column number	z G |
scroll amount	^D ^U
repeat effect	most of the rest

Interrupting, canceling

ESC	end insert or incomplete cmd
DEL	(delete or rubout) interrupts
^L	reprint screen if DEL scrambles it
^R	reprint screen if ^L is -> key

File manipulation

:wCR	write back changes
:qCR	quit
:q!CR	quit, discard changes
:e nameCR	edit file name
:e!CR	reedit, discard changes
:e + nameCR	edit, starting at end
:e +nCR	edit starting at line n
:e #CR	edit alternate file
	synonym for :e #
:w nameCR	write file name
:w! nameCR	overwrite file name
:shCR	run shell, then return
:!cmdCR	run cmd, then return
:nCR	edit next file in arglist
:n argsCR	specify new arglist
^G	show current file and line
:ta tagCR	to tag file entry tag
^]	:ta, following word is tag In general, any ex or ed command (such as substitute or global) may be typed, preceded by a colon and followed by a CR.

Positioning within file

^F	forward screen
^B	backward screen
^D	scroll down half screen
^U	scroll up half screen
G	go to specified line (end default)
/pat	next line matching pat
?pat	prev line matching pat
n	repeat last / or ?
N	reverse last / or ?
/pat/+n	nth line after pat
?pat?-n	nth line before pat
]]	next section/function
[[	previous section/function
(	beginning of sentence
)	end of sentence

{	beginning of paragraph
}	end of paragraph
%	find matching ( ) { or }

Adjusting the screen

^L	clear and redraw
^R	retype, eliminate @ lines
zCR	redraw, current at window top
z-CR	... at bottom
z.CR	... at center
/pat/z-CR	pat line at bottom
zn.CR	use n line window
^E	scroll window down 1 line
^Y	scroll window up 1 line

Marking and returning

‘‘	move cursor to previous context
’’	... at first non-white in line
mx	mark current position with letter x
‘x	move cursor to mark x
’x	... at first non-white in line

Line positioning

H	top line on screen
L	last line on screen
M	middle line on screen
+	next line, at first non-white
-	previous line, at first non-white
CR	return, same as +
ǀ or j	next line, same column
ǀ or k	previous line, same column

Character positioning

^	first non white

<table>
<tr><td>0</td><td>beginning of line</td></tr>
<tr><td>$</td><td>end of line</td></tr>
<tr><td>h or -></td><td>forward</td></tr>
<tr><td>l or <-</td><td>backwards</td></tr>
<tr><td>^H</td><td>same as <-</td></tr>
<tr><td>space</td><td>same as -></td></tr>
<tr><td>fx</td><td>find x forward</td></tr>
<tr><td>Fx</td><td>f backward</td></tr>
<tr><td>tx</td><td>upto x forward</td></tr>
<tr><td>Tx</td><td>back upto x</td></tr>
<tr><td>;</td><td>repeat last f F t or T</td></tr>
<tr><td>,</td><td>inverse of ;</td></tr>
<tr><td>|</td><td>to specified column</td></tr>
<tr><td>%</td><td>find matching ({) or }</td></tr>
</table>

Words, sentences, paragraphs

w	word forward
b	back word
e	end of word
)	to next sentence
}	to next paragraph
(	back sentence
{	back paragraph
W	blank delimited word
B	back W
E	to end of W

Corrections during insert

<table>
<tr><td>^H</td><td>erase last character</td></tr>
<tr><td>^W</td><td>erase last word</td></tr>
<tr><td>erase</td><td>your erase, same as ^H</td></tr>
<tr><td>kill</td><td>your kill, erase input this line</td></tr>
<tr><td></td><td>quotes ^H, your erase and kill</td></tr>
<tr><td>ESC</td><td>ends insertion, back to command</td></tr>
<tr><td>DEL</td><td>interrupt, terminates insert</td></tr>
<tr><td>^D</td><td>backtab over autoindent</td></tr>
<tr><td>| ^D</td><td>kill autoindent, save for next</td></tr>
<tr><td>0^D</td><td>... but at margin next also</td></tr>
</table>

^V quote non-printing character

Insert and replace

a append after cursor
i insert before cursor
A append at end of line
I insert before first non-blank
o open line below
O open above
rx replace single char with x
RtextESC replace characters

Operators

Operators are followed by a cursor motion, and affect all text that would have been moved over. For example, since w moves over a word, dw deletes the word that would be moved over. Double the operator, e.g., dd to affect whole lines.

d delete
c change
y yank lines to buffer
< left shift
> right shift
! filter through command
= indent for LISP

Miscellaneous Operations

C change rest of line (c$)
D delete rest of line (d$)
s substitute chars (cl)
S substitute lines (cc)
J join lines
x delete characters (dl)
X ... before cursor (dh)
Y yank lines (yy)

Yank and Put

Put inserts the text most recently deleted or yanked. However, if a buffer is named, the text in that buffer is put instead.

p	put back text after cursor
P	put before cursor
"xp	put from buffer x
"xy	yank to buffer x
"xd	delete into buffer x

Undo, Redo, Retrieve

u	undo last change
U	restore current line
.	repeat last change
"dp	retrieve d'th last delete

Bibliography

The following books provide additional tutorial and reference material that you might find useful in exploring various programming tools.

Unix System V Release 3.2 Programmer's Guide, Vol I and II; AT&T; Prentice Hall, 1989

Managing Projects With Make; Steve Talbott; O'Reilly & Associates, Inc., 1988

Learning C; Stephan Kochan; Hayden Books, 1988

Unix Text Processing; Dale Dougherty and Tim O'Reilly; Hayden Books, 1988

Word Processing On The Unix System; Morris Krieger; McGraw Hill Books, 1985

Exploring The Unix System; Stephen Kochan and Patrick Wood; Hayden Books, 1989

The AWK Programming Language; Alfred Aho, Brian Kernigan, Peter Weinberger; Addison Wesley Publishing, 1988

USING THE 1500/3000 COMPILERS

CHAPTER TWO

This chapter describes how to compile Stardent 1500/3000 Fortran and Stardent 1500/3000 C source programs into object modules that you can execute. This chapter describes

- The functions of the compilers.
- How to call the compilers.
- How to use the compiler options.
- The formats of the listings produced by the compilers.
- How to locate and correct run-time errors.

This chapter is divided into three sections. The first section contains a general discussion of the functions of the compilers, the options that are common to both languages, preprocessor control, and linking. The last two sections contain the details of calling the compilers, options to the compilers, and locating errors in Fortran and C, respectively.

The Stardent 1500/3000 Compilation System

The Stardent 1500/3000 compilers construct object files from source language files. The Stardent 1500/3000 compilers execute under UNIX. The compilers generate binary object files; these files may be combined by the loader with one another and with system libraries to create executable programs. Ordinarily, specifications of these files are of little interest, but a simplified description of the contents is found in *Chapter 5, Running dbg— The Stardent 1500/3000 Debugger*.

Functions of the Stardent 1500/3000 Compilers

The Stardent 1500/3000 compilers perform the following functions.

- They check that your program is correct. If your program is not correct, they tell you where you have made errors.
- They translate your source program into machine language instructions which the Stardent 1500/3000 can execute.
- They group these executable instructions into object modules which may be linked (by the loader) into a complete executable program.

In addition to the executable instructions, the compilers also generate lists of all subroutines and common blocks defined and used in each module. The loader combines object modules into complete programs, adding additional routines from user and system libraries as needed. During the loading process, a symbol defined in one routine is linked to all the other routines which use the symbol. The compilers and the loader also communicate to pass debugging information to the debugger.

Specifying the Output Files

You use compiler options to specify what kind of output is to be produced. For example, the **-S** option produces an assembly language file. Refer to the option lists for each language later in this chapter for more information about controlling output.

Compilation Control

There are three ways to control the compilation of your program: command line options, compilation control statements, and compiler directives. This section discusses only those options and statements that can be used from either Fortran or C.

Command Line Options

The following three tables list all possible options that can be used in both Fortran and C. Detailed descriptions and explanations of these options are discussed following the table. Defaults are indicated in italics. The negative form of the option only applies to Fortran. Options are case-sensitive; upper and lower case options have different meanings.

If the Fortran command line compiler option **-cpp** is not specified when coding in Fortran, most of the command line preprocessor options are ignored. The options **-P** and **–E** always invoke the preprocessor, however the others do not do this unless the **-cpp** option is specified.

Table 2-1. Command Line Preprocessor Options for Fortran and C

Option	Negative Form	Description
-D*name*		Define *name* to have a value of 1
-D*name=val*		Define *name* to have a value of *val*
-E		Preprocess only; output modified source to the file defined as *stdout* (often the user's terminal)
-I		Do not search for included files in default directory
-I*dir*		Search for included files in *dir*
-i		Do not include information created by the #ident preprocessor option in the output of the preprocessor step
-P		Preprocess only; put the results in *file.i*
-U*name*		Undefine *name*

Table 2-2. Command Line Compiler Options for Fortran and C

Option	Negative Form	Description
-c		Compile only; do not link
-catalog=*name.in*		Create a database of inlined functions
-cpp		Pass a Fortran program through the C language preprocessor before doing the Fortran compile
-full_report		Invoke vector reporting facility
-fullsubcheck		Add code to check each linear array subscript
-g	**-nodebug**	Add debug data to object file
-inline		Inline functions
-Npaths=*name.in*		Use inlined functions from *name.in*
-O0	**-nooptimize**	Turn off optimization
-O1		Perform scalar optimization
-O2		Perform -O1 and vectorization
-O3		Perform -O2 and parallelization
-O		Same as -O1
-o *filename*		Put output into *filename*
-ploop		Profile each loop separately
-S		Generate assembly language source file
-subcheck		Add code to check linear array subscripts
-V		Print version information
-vreport		Invoke vector reporting facility with a particular kind of output
-vsummary		Invoke vector reporting facility with a particular kind of output
-w		Suppress warning messages during compilation
-43		Use BSD capability

Table 2-3. Command Line Loader Options for Fortran and C

Option	Negative Form	Description
-B*hhhhhhhh*		*a.out* has bss address at *hhhhhhhh*
-D*hhhhhhhh*		*a.out* has data address at *hhhhhhhh*
-e*sym*		Set default entry point address
-L		Do not search for libraries in */lib* or */usr/lib*
-L*dir*		Search for libraries in *dir*
-l*tag*		Search library called *tag.a*
-m		Generate a simple load map
-n		Generate NMAGIC file type
-opct		Produce count of FPU ops
-p	**-noprofile**	Generate profiling code
-r		Produce a relocatable output file
-s		Strip line numbers/symbol table information
-T*hhhhhhhh*		*a.out* has text address at *hhhhhhhh*
-t		Turn off certain warnings
-y*name*		Trace *name*

NOTE
The notation *hhhhhh* indicates an 8-digit machine address given in hexadecimal notation. Normally hexadecimal numbers are specified as 0xNNNNNNNN. For use with these options, hexadecimal numbers are expected; thus the "0x" should not be used.

Preprocessor Options

-D*name*

Define *name* to have the value of 1 to the preprocessor. This option may be used in place of explicitly including a *#define* statement within the program itself. It is particularly useful if more than one file is specified on the command line since the definition of the *name* is valid for the compilation of all files specified.

-D*name=val*

Define *name* to have the value of *val* to the preprocessor.

-E

This option stops the compilation process after all macros and conditional compilation expressions have been expanded. Passing source through a preprocessor often causes an increase in the physical bulk of the file, the preprocessed material is often called the expanded source. Thus the option is selected by an E. The expanded source is directed to stdout (the standard output, most commonly the user's

terminal.) A "real" compilation does not take place. Output of the preprocessor appears on standard output.

-I

When the preprocessor encounters a *#include* statement in a C language program, it searches for a file named in that statement in the directory */usr/include*. Specifying the **-I** option suppresses the search of this default directory. This option is useful if the user has a customized version of a particular file in a specific directory other than the current directory and wants this alternate file to be included instead of that which the system would normally find during its search process. The **-I** option is often used with the **-I***directory_name* option to tell the compiler where to look, if not the default directory. This does not affect the Fortran **INCLUDE** statement that searches the default directory.

-I*dir*

Search for preprocessor include-files in *dir*. This does not affect items that might have been specified with the Fortran **INCLUDE** statement.

-i

Suppress the automatic production of *#ident* information.

-P

Preprocess only; the resulting output (from *x.c* or *x.f*) goes to *x.i* rather than standard out.

-U*name*

Undefine *name*.

Compiler Options

This is an abbreviated list of options, describing only those options that are common to both Fortran and C. For a complete list of the options that are unique to Fortran and a detailed explanation of their effects, see *Chapter 7, Fortran Compiler Options* in the *Fortran Reference Manual.*

-c

Compile the source files, but do not invoke the loader. Generate only object code output.

-catalog=_filename.in_

Create a database of functions that are frequently inlined. Be aware that the use of **-catalog** creates two files; the database and a _filename.in_. This can use up a large amount of disk space, as the use of this option adds to an existing catalog.

To create a catalog in Fortran, specify the following on the command line:

```
fc  -catalog=file.in  filename
```

To create a catalog in C, you only need to specify _filename.in_ on the command line, without using the **-catalog** option. The compiler assumes the creation of the catalog from this, so long as **-inline** is not specified.

-full_report

Selecting this option invokes the vector reporting facility. The report that is generated shows you how your program statements have been vectorized, shows you the code the compiler generated, and explains why certain choices (code reorganization, for example) have been made. The **full_report** is the most detailed report that is generated by the vector reporting facility. Refer to the section _Vector Reporting Facility_ later in this chapter for examples and complete descriptions of the output produced by invoking any of the three options **-vsummary**, **-vreport**, or **-full_report**. This output is in a notation that strongly resembles Fortran.

-fullsubcheck

Generate code to check that every subscript in every array reference is within the bounds of the appropriate array dimensions. For example, if the DIMENSION A(10,10) is declared, A(11,5) definitely violates the conditional definition of the **-fullsubcheck** option. The testing that is performed is more stringent than that of option **-subcheck**.

NOTE
Use of this option increases object code size and decreases execution speed.

-g

Generate information for the Stardent 1500/3000 debugger. It is synonymous with the option **-debug**.

-inline

When **-inline** is specified and a catalog is listed in **-Npaths**, the compiler inlines all legal functions found in the catalog. For example

If you specify `fc -inline` without a `-Npaths`,

then the compiler inlines from */usr/lib/libbF77.in* which contains the BLAS subroutines. In addition to inlining BLAS, the compiler will also inline very short subprograms from your source code.

If you specify

`fc -inline -Npaths=`*mycat.in*

then the compiler inlines from *mycat*.in **and** the standard system catalog */usr/lib/libbF77.in*.

If you specify

`fc -inline -Npaths= -Npaths=`*mycat.in*

then the compiler inlines only from *mycat*.in and very short subprograms in your source code, ignoring the standard system catalog.

If you specify

`fc -inline -Npaths=`

then the compiler inlines only very short subprograms from your source code.

The Stardent 1500/3000 compiler can inline virtually any type of function into any language construct. However, there are some exceptions. The compiler does not inline character-valued functions or functions that take character arguments. C functions which appear to be *varargs* (variable number of arguments—the address of the argument is taken) cannot be inlined. Finally, the compiler does not inline any function into the WHILE condition of a loop.

-Npaths=*filename.in*

Instruct the compiler to make use of the database of inlined functions (*filename.in*). Specifying **-Npaths=** avoids getting system functions.

-O0

Turn off all optimizations.

-O1

Perform scalar optimizations, including common subexpression elimination and instruction scheduling.

-O2

Perform **-O1** and vectorization.

-O3

Perform **-O2** and parallelization.

-O

This is the same as **-O1**.

-o *filename*

Place the output into *filename.*

-ploop

Profile each loop in a single routine separately. Refer to *Chapter 8, Tuning and Porting Code* for detailed information on using this option. You may also refer to the *Commands Reference Manual.*

-S

Do not generate object files; instead, generate assembly language source files.

-subcheck

Produce code to check at runtime to ensure that each array element accessed is actually part of the appropriate array. Consider the following fragment

NOTE
Use of this option increases object code size and decreases execution speed.

```
REAL SAM(10,20,30)
...
...
... = SAM(I,J,K) ...
```

If I has any value between 1 and 10, J between 1 and 20, and K between 1 and 30, then the access to an element of array SAM is legal. However, because of the way Fortran arrays are stored in memory, the reference still lies in the storage allocated for SAM when I=25, J=2 and K=4; in fact, the selected element is SAM(5,4,4). In general, there are an infinity of ways to write subscripts that are not strictly legal, but which name an element lying in the array. If used, this option warns only about an access that falls completely

outside of the storage for an array. In other words, this option does **not** check array subscripts individually, but simply checks that the result of the calculation of the array element's location is actually within the bounds of the array.

-V

Print version information.

-vreport

This invokes the vector reporting facility and is used to tell the user what vectorization has been done to the program. A detailed listing is provided of exactly what the compiler did to each loop nest. This option does not include suggestions on how to achieve better performance. Refer to the section later in this chapter titled *Vector Reporting Facility* for detailed information. The output from this option is in a notation that strongly resembles Fortran.

-vsummary

This invokes the vector reporting facility and is used to tell the user what vectorization has been done to the program. This option, **-vsummary**, prints out what statements are and are not vectorized in each loop nest.

-w

This option suppresses warning messages during compilation.

-43

If this option is invoked by a C program, this option supports BSD capabilities. This consists of a BSD header file and a number of BSD libraries. Use this option immediately after the *cc* command. If this option is invoked by a Fortran program, the effect of this option is to add the Berkeley library *libc.a* to the end of the load line.

Loader Options

-B*hhhhhhhh*

Generate an *a.out* file with the bss address at *hhhhhhhh*.

-D*hhhhhhhh*

Generate an *a.out* file with the data address at *hhhhhhhh*.

-e*sym*

Set the default entry point address for the output file to be that of *sym*.

-L

Do not search for libraries specified on the command line with a lower case L in */lib* or */usr/lib*.

-L*dir*

Search for libraries specified on the command line with a lower **-l** libraries in *dir*.

-l*tag*

For libraries to be included in the executable file, search a library called *libtag.a*. The name of the library is formed by prepending *lib* to the word given with the **-l** option. The default is to search first in */lib* and then in */usr/lib*. As noted above, the **-L** options can be used to override the default search path.

-m Generate a simple load map on standard output.

-n Generate NMAGIC file type. This option is not normally selected, and is documented here only for completeness. This is one of the options (**-O** is another) that affects the format of the header of the *a.out* file. The file types are called the following:

- ZMAGIC, the standard form of an a.out file, that is, an executable file. In this form of file, the text and data segments are placed such that the data segment begins at an even page boundary, allowing the loader to page the program in directly from the file.
- NMAGIC, by comparison to ZMAGIC, has the data segment immediately following the text segment, and thus makes the object file smaller. It is primarily used by diagnostics programmers.
- OMAGIC, (the format of the file is to be that of a *file.o*). This is a non-executable file and will contain object code that has unresolved references. In other words, to create a final **a.out** file, a .o file must be passed through the link editor before it will become an executable file.
- SMAGIC is reserved for shared library object files. The method of generating this type of file is not documented here.

-opct

Insert code into the program that dynamically produces a count of all the FPU operations executed by the program.

-p Generate code to profile the source file during execution. Refer to *Chapter 8, Tuning and Porting Code* for detailed information on using this option.

-r Produce a relocatable output file.

-s On most UNIX systems, this option strips debugging information from the output object file. On the Stardent 1500/3000, this option performs no action; it is provided only for compatibility with other systems. To remove debugging information, use the strip command explicitly.

-T*hhhhhhhh*

Generate an *a.out* file with the text address at *hhhhhhhh*.

-t Turn off warnings about multiply defined symbols that are not the same size.

-y*name*

Trace *name* and print out all uses and definitions.

Preprocessor Control Statements

In addition to the compilation control options that you specify on the command line, you can communicate with the Stardent 1500/3000 preprocessor by including certain statements in your source code.

- The *#include* statement specifies that an external source file is to be copied into the source code and to be processed with the file.
- The *#define* statement establishes values for symbols that you use in your source code or that simply act as preprocessor symbols to otherwise control the inclusion of other source code. The *#undefine* statement undefines, or removes the value associated with a symbol.
- The *#ifdef* and *#ifndef* statements simply test whether or not you have defined a particular symbol, that is, have you given it **any** value, numeric or alphabetic.

- The *#endif* statement terminates an *#if, #ifdef* or *#ifndef* statement, allowing nested test conditions to be formed.
- The *#if* statement lets you apply integer arithmetic tests against the integer value of a symbol and choose to include or not include sections of source code based on the results of the test.
- The *#elif* and *#else* statements allow you to include alternative source code if the tested conditions are found to be false.
- The *#line* statement lets you label lines of source code allowing you to have the compiler more accurately report the location at which a runtime error occurs.

The #include Statement

Using an include statement tells the compiler to copy the contents of the specified file into the source code during the preprocessing stage. This material then becomes a part of the source code when it is finally processed by the compiler.

An include file normally contains data structures and data definitions that may be common to many different programs you write. This saves you from a need to copy the source code into each program, allowing the preprocessor to do it for you as the program is compiled. Here is an example of an include statement.

```
#include <definitions.h>
```

The preprocessor normally looks in two places for included files: the current directory (the first choice, where a local copy of some file name is considered more important than that same named file located elsewhere), and */usr/include*. If an included file is in neither of these places, the search fails. You can override the default search places, however, by using the **-I** switch on the compiler command line. This tells the compiler not to search */usr/include* (although it will still search the current directory, even when this switch is specified). Or you can provide additional places to search by specifying **-I**<*new_directory*> where *new_directory* is an explicit path (from the root) to the directory where the include file is located.

The #define Statement

This statement is most commonly used to assign values to symbols, although any kind of text can be substituted for an identifier.

Once the assignment has been made, that value is now available to all statements that follow in the module.

This statement is commonly used to make your source code easier to read by specifying symbols in place of constants. In particular, it makes it unnecessary to generate code for assigning constant values to various symbols, and instead directs the compiler to actually substitute the constant where the symbol occurs as the compilation occurs.

A programmer might write the following code:

```
REAL*8 PI,RADIUS,R
PI = 3.14159
RADIUS = PI * R * R
```

Instead of that format, the programmer could instead write it as follows:

```
#define PI 3.14159
RADIUS = PI * R * R
```

Following the run of the preprocessor (the **-E** option of the compiler), the preprocessed source code would then have substituted 3.14159 for every occurrence of PI, does not save code space, however it does help performance.

Using the *#define* statement is not the only way to establish a string value for a variable named in your source code. The compiler option **-D** can also be used. Typically this option is used simply to establish a nominal value of 1. This defines the variable as a preprocessor constant symbol. Now this symbol can be used in *#ifdef* ("is this constant 'defined'?"), or *#ifndef* ("is this constant 'undefined'?") where there is no explicit *#define* statement in the code, nor an explicit -D*ITEM=1* on the command line defining it. *ITEM* is the name of a string variable to define and what is to the right of the equals sign is the definition for that variable.

The #ifdef Statement

This directive allows conditional inclusion of source statements in your program if the preprocessor identifier is defined. No values are tested with this statement, only a test for a definition of an argument.

There are three methods that a programmer can use to explicitly define a symbol such that the preprocessor, using the *#ifdef*

statement will recognize a symbol as being defined:

- Explicitly define the symbol by using a *#define* statement:

```
#define DEBUG 1
```

- Specify a null defined value on the command line by using the **-D** command line option:

```
-DDEBUG
```

- Specify a symbolic value by using the **-D** option on the command line:

```
-DDEBUG=<integer_value>
```

If the *#ifdef* statement is determined to be true (the item has been defined by one of the three available methods), the source statements included between the *#ifdef* and the *#endif* statement become part of the preprocessed source code presented to the compiler.

Here are examples. Note that the *#endif* statement terminates a block of statements controlled by the *#ifdef* statement. As shown later in this section, *#endif* terminates statement blocks begun with *#ifndef* or *#if* as well.

```
C     Fortran Example
#ifdef DEBUG
      WRITE(6,100)
100   FORMAT('ERROR IN DATA SPECIFICATION, TRY AGAIN');
#endif DEBUG

/* C example */
#ifdef DEBUG
      printf("reached this statement0);
#endif DEBUG
```

The *#ifdef*, *#else*, and *#endif* statements can be used not only to determine whether to use or not use explicit source statements, but can also enclose other compilation control statements, such as *#include* or other conditional conditions, forming nesting levels of tests.

The statements *#ifdef*, *#ifndef*, and *#else* are often used where multiple include files may have interdependencies. If a particular file's parents have not already been included, this type of statement includes them. If one or more of the file's parents have been included, the preprocessor symbol definitions prevent the compiler from including them more than once, thereby preventing multiply defined symbols. The section that describes *#ifndef*

includes such an example.

The #ifndef Statement

This directive allows conditional inclusion of source statements in your program only if the preprocessor identifier is **not** defined. This statement is the reverse of the *#ifdef* statement.

Here are two files, one of which requires that its parent also be loaded to complete the definitions:

```
/* sample file name is links.h */

#ifndef LINKS_H
#define LINKS_H 1

struct header {
   struct header *next;
   unsigned long value;
};
#endif  LINKS_H
/* end of file links.h */

/* sample file name is uses.links.h */

#ifndef LINKS_H
#include "links.h"
#endif  LINKS_H

struct sample {
   struct header sampleHead;
   int otherstuff[20];
};
/* end of uses.links.h */
```

If you are not aware that the file dependencies exist, you might add the following line to your program:

```
#include "uses.links.h"
```

Since the preprocessor symbol *LINKS_H* is undefined, the include file itself automatically adds to the source code the statement *include "links.h"*. The definitions get completed automatically.

However, if you actually remember that one file depends on the definitions in another file, you could specify

```
#include "links.h"
#include "uses.links.h"
```

By adding the *#ifndef* statements in *uses.links.h,* it prevents this file from being included more than once, even though the user has explicitly requested it.

The #endif Statement

This statement terminates the *#if, #ifdef,* and *#ifndef* statements. It is also used to terminate intermediate nesting levels. In the previous examples above, the *#endif* statements are each shown written with the preprocessor symbols that match the corresponding *#ifdef* or *#ifndef* that begins the block of statements. In actuality, the *#endif* statement ignores the symbol, but you may place the symbol on the *#endif* line anyway, even though the symbol will not be used. You can use this to keep track of the nesting levels that you have created.

Here is an example of nested tests, both using *#ifdef* statements. Again, the labels on the *#endif* are not required, but they show the nesting and are therefore recommended. Additionally, you may use white space (spaces or tabs) following the initial '#' sign to indent and therefore more clearly show the nesting level.

```
#ifdef DEBUG1
#include "firstlevel.debug.functions"
#    ifdef DEBUG2
#    include "second.level.debugs"
#    endif DEBUG2
#endif DEBUG1
```

In this example, the second level debug functions depend on functions defined in the first-level group. The second level functions should not be loaded unless the first level functions are available. Notice that the test for the definition of *DEBUG2* does not happen unless *DEBUG1* is already defined.

The #if Statement

This statement allows the conditional inclusion of source statements in your program. These source statements are included only if the integer expression following *#if* is non-zero. The *#if* statement takes the following form:

```
#if <constant_expression>
[these statements are included if
 the integer arithmetic statement
 gives a nonzero result, up to the
 next #elif, #else, or #endif]
#endif
```

The values of all items in the expression must be known at preprocessing time to allow the expression's value to be determined. Thus, the item to be evaluated must be a constant expression because the preprocessor is not as sophisticated as the compiler.

All binary non–assignment C operators can be used in the constant expression, including all of the following: '–', '+', '*', '/', '&', '|', '&&', '||', '!','?:', '~','?:', '>', '>=', '<=', '<>', '=='. Table 2-4 contains a few examples, showing only the expressions, and not the preceding *#if*, or the block of statements that would be included. When an expression has a binary evaluation that is nonzero, the block associated with that *#if* is included. Otherwise it is skipped and not copied as preprocessed output.

The #elif Statement

This statement can be used with the *#if*, *#ifdef*, and *#ifndef* statements to create nesting levels. Here is an example.

Table 2-4. Examples Of Valid Expressions for #if

Expression	Interpretation
A - B	difference between A and B
C + D	sum of C and D
E * F	product of E and F
G / H	quotient of G and H
A & B	bitwise AND of A and B
C	D	bitwise OR of C and D
(E && (A - B))	both E and the result of A - B must be nonzero for this to be nonzero.
(E || (A - B))	either E or the result of A - B must be nonzero for this to be nonzero.
!D	D must be zero... this is the 'NOT' operator
A > B	A must be greater than B
A <= B	A must be less than or equal to B
C <> D	C must not equal D
G ~ H	G exclusive-or H must evaluate to nonzero
E == F	E must equal F
(A > B)?C:D	if A > B, then use C as the result to be tested for nonzero condition, otherwise use D.

```
#if (DEBUG_LEVEL >= 3)
/* debug level 3+ statements */
#elif (DEBUG_LEVEL >= 2)
/* debug level 2 statements */
#endif
```

The #else Statement

This statement can be used with the *#if*, *#ifdef*, and *#ifndef* statements to create nesting levels. The statement *#else* defines what should be done if the preceding test(s) are all found to be false. It must be associated with one of the forms of *#if*. Its block is terminated by an *#endif* statement. Here are some examples.

```
#ifdef TESTING
  [do this if TESTING is defined]
#else
  [do this if TESTING is not defined]
  [in this case, is an alternative to #ifndef]
#endif
```

```
#if (A > B)
  [do one thing]
#elif (A < B)
  [do something different]
#else
  [third alternative; in this case
   a substitute for #elif (A == B)]
#endif
```

The #undef Statement

This statement terminates the *#define* statement and causes the argument to become undefined. That is, any references to the name further down in the file result in an evaluation as a zero-length string.

The #line Statement

This statement provides you with the ability to label lines of source text. It takes the following form. Note that the item designated as *filename* must be enclosed in double-quote marks.

```
#line integer-constant "filename"
```

Generate line control information for the next pass of the C compiler. The *integer-constant* is interpreted as the line number of the next line and *filename* is interpreted as the file from where it comes. If filename is not given, the current filename is unchanged. No additional tokens are permitted on the directive line after the optional filename.

In other words, when the C compiler gets this information, if an error occurs on the line following the *#line* statement, the compiler reports the following:

```
<filename>: line XXX — Error ZZZ
```

When an error occurs, the compiler relates the error to a specific line number in a specific file instead of the line number in the file it actually came from. Normally, this statement is not used explicitly by programmers; instead it is used in programs that write other programs. For example, **yacc** generates a *#line* statement for each line that it includes from a source file into a program that it generates (*yacc.yy.c*). When errors occur, then, the compiler can report exactly which program and which line number caused the problem.

Other Forms Of Define and Include

The Fortran language also has its own individual INCLUDE and DEFINE statements which are covered in detail in the *Fortran Reference Manual.*

Compiler Directives

Compiler directives are used to override the compiler's optimizations and influence all phases of compilation. A complete discussion of these directives is located in *Chapter 7, Efficient Programming Techniques*.

Linking Multiple Files

If you do not specify the **-c** or **-s** option on your command line, then a successful compilation of your program results in a file named *a.out* in the same directory in which your source program was located.

If you wish to compile several program modules individually or you wish to compile other programming language modules to be linked with your program, then use the **-c** option when you compile. Instead of a file named *a.out*, you get a file whose name is the same as your source file name with the *.f* or *.c* removed and a *.o* appended in its place. This is the object file that you later provide to the loader. See *Chapter 4, Using Libraries and the Link Editor* for more details.

Vector Reporting Facility

You can tell what the compiler has done to your code by invoking the vector reporting facility. This can be done by the use of three different compiler options or by using a compiler directive, **VREPORT**. While the examples below are given for Fortran, the same options invoked on the C compiler will produce equivalent output in C-like notation.

-vsummary

This option is designed for programs that have been tuned on other vector machines and are being ported to the Stardent 1500/3000. The user expectation in this case is that all the key loop nests will vectorize and parallelize. The **-vsummary** option permits the user to easily confirm that this is the case or to quickly locate the loops that do not vectorize if this is not the case. Under

this option, the compiler prints out a one-line summary for each loop nesting either stating that all statements vectorized or indicating which statements did not. The **-vsummary** option provides no diagnostics as to why loops did not vectorize, nor does it indicate exactly how a given loop nest vectorizes. The **-vreport** and **-full_report** options can be used to uncover this information about loops pinpointed by **-vsummary**.

Example

The following is an example of a program fragment and the output produced by compiling with **-vsummary**.

```
DOUBLE PRECISION A(100,100), B(100,100), C(100,100)
INTEGER I, J, K
DO I = 1, 100
   DO J = 1, 100
      C(I,J) = 0.0
      DO K = 1, 100
         C(I,J) = C(I,J) + A(I,K) * B(K,J)
      ENDDO
   ENDDO
ENDDO
```

The **-vsummary** report is as follows

```
Vectorization summary for file x.f
***************************************
Line     3:  All statements vectorized.
```

The report indicates the line number at which the loop nest is located and a message indicating one of three cases: all statements vectorized (as above), the loop was not examined by the Stardent 1500/3000 vectorizer (which happens if the loop contains a branch that exits the loop or a function call), or some statements in the loop were not vectorized.

-vreport

Sometimes just knowing that all the statements in a loop vectorized is not enough. For instance, if there are multiple loops in a nest, it may be important that the Stardent 1500/3000 compiler vectorize the correct loop (it only vectorizes one) since there may be short loops or loops with bad strides. The **-vreport** option provides information on how and why a loop vectorizes, plus information on other transformations that the Stardent 1500/3000 compiler may have performed. To illustrate, following is the output from **-vreport** on the previous example.

Example

```
        Vectorized Results from File x.f
                Origin - - Line 3

Line    Stmt    Time    Program

5       6       3       b2 = 100
7       10      3       b3 = 100
5       *       4       DO PARALLEL J=1, 100
4       *       4         DO iv=1, 100, 32
4       *       8            rv = MIN(100, 31 + iv)
*       *       9            vl = rv - iv + 1
*       *       9            vl = rv - iv + 1
*       *       6            DO VECTOR I=iv, rv
6       8       33            C(I, J) = 0.0D0
                             END DO
7       *       4            DO K=1, 100
4       *       6               DO VECTOR I=iv, rv
8       12      119               C(I, J) = C(I, J) +
                                     A(I, K) * B(K, J)
                                END DO
                             END DO
                           END DO
                         END DO
```

At the far left of the report is a listing of the line numbers of each statement in the original source program. In this particular example, the line number information is not of much value, but in some cases, the Stardent 1500/3000 compiler may change the statement order to enhance vectorization. In those cases, the line number information can be of great value. Asterisks in the line number field indicate statements that were generated by the compiler and that do not appear in the original source program.

The "Stmt" field indicates a unique statement number assigned by the compiler. Diagnostics and suggestions printed by the compiler are keyed to this field, since the "Line" field is not always unique (for instance, there are three statements derived from line 4 in the example). Again, asterisks indicate a statement which for some reason did not receive a statement number—such statements are always of little significance in the program execution.

The "Time" column is a static estimation by the compiler of roughly the number of clock ticks the statement will take to execute. Static estimates are never very accurate, so you should not rely on this field to estimate the execution time of your program. It can be used to indicate the statements that should receive your special attention. Thus, the 119 ticks required for each execution of statement 12 (since it is a vector statement, each execution operates on 32 elements) is probably not a good estimate of the

actual time the statement will take, but it does indicate that it is far more important that it vectorize than statement 8, which only takes 33 ticks.

The compiler attempts to print out the program listing itself (the rightmost column) in a form as close as possible to the original code. DO loops that have been parallelized (line 5) are indicated as DO PARALLEL; vector operations are indicated by a DO VECTOR loop (line 4) so that the full subscripts can be presented, rather than one linearized version. Unless the -case_sensitive flag is used, variables that are in the source program are printed in upper case and temporaries generated by the compiler are printed in lower case. Thus, I, J, and C are all variables from the original program, whereas b1 and b2 are temporaries generated by the compiler.

Some compiler temporaries hold information that may be useful to you. In general, variables that end with a number hold some piece of information about the loop at the nesting level indicated by the number—thus, the variable b2 tells something about the loop at nesting level 2 (the original DO J loop) and b3 indicates something about the level 3 loop (originally the DO K loop). In fact, "b" variables indicate the number of times the given loop will iterate: both the DO J and DO K loops iterate 100 times each time they are executed.

There are also "r" variables for each loop (although the compiler has eliminated them in this example), which indicate the upper bound for each loop. When the Stardent 1500/3000 compiler vectorizes a loop, it "strip-mines" it to a vector length of 32. That is, it breaks the loop into two loops—an inner loop, always of length 32 or less (which will become the actual vector operation), and an outer loop (called the strip loop) to sweep the inner loop across the full range of the original loop. By performing this transformation, the compiler guarantees that all vector operations executed in the Stardent 1500/3000 hardware are less than 32 in length, thereby making vector register allocation much easier.

In the vector report, strip loops are always controlled by either the variable "iv" or "ip" and the actual vector loop is controlled by the original user loop variable. Thus, in the example above, the strip loop is the outmost loop from line 4, which corresponds to the original DO I loop, and the actual vector operations are deeper inside.

The variables "rv" and "rp" are used to hold the upper bound for each vector operation during one iteration of the strip loop.

Finally, the variable "vl" is used to hold the length of vector operations for each iteration of the strip loop. In the example, the first three iterations of the strip loop will use a vector length of 32; the last iteration will do a cleanup operation of length 4. "vl" will hold these values on each respective iteration of the strip loop. For various technical reasons, the Stardent 1500/3000 compiler occassionally must generate multiple assignments to "vl" (as in the example above). There is no need to worry about these cases, as they take a very small amount of execution time.

Looking back at the example output, you can tell that the Stardent 1500/3000 Fortran compiler has significantly transformed the source program. The compiler decided that the DO I loop—the outmost loop in the original nest—was the best loop to vectorize, and stripped it to a length of 32. For statement 12, it switched the actual vector loop with the DO K loop, moving the K loop outward, to get better performance. The compiler then decided that the DO J loop (the original level-2 loop) was the best loop to run in parallel. The more computation a parallel loop contains, the more effective it is. So the compiler interchanged the parallel DO J with the strip loop from the I loop. As a result, the parallel loop becomes the outmost loop in the nest. The resulting code makes very effective use of the Stardent 1500/3000 hardware.

There is one other important feature of **-vreport** that is not evident in this example. The Stardent 1500/3000 vectorizer is designed to be very machine independent and vectorizes loops based on the properties of the statements in the loop, independent of the capabilities of the Stardent 1500/3000 processor. Of course, it is very difficult to execute a vector operation for which there is no vector hardware. As a result, the Stardent 1500/3000 compiler contains a later, machine-dependent pass that converts vector operations for which there is no vector hardware into an equivalent sequence of scalar operations. Obviously, the scalar operations do not run as fast as the vector operations, so the compiler also prints a message in the vector report indicating the vector operation it converted. The following is an example of such a message:

```
Following vector ops in line 4 run at scalar speed
                   DIV (integer)
```

This message lets the user know that there is no vector divide instruction for integers on the Stardent 1500/3000, so that it might be worthwhile coding the loop in a different way. Similarly, some parallel loops require more memory bandwidth than the Stardent 1500/3000 hardware can deliver. In those cases, the Stardent 1500/3000 compiler will run the loop sequentially, and will warn

you in the vector report with the following message:

```
Parallel loop at line 3 made scalar due to memory bandwidth.
```

-full_report

Sometimes the Stardent 1500/3000 Fortran compiler will not vectorize loops that you expect to vectorize, or will vectorize different loops than you expect. In those cases, it is valuable to understand why the Stardent 1500/3000 vectorizer made the choices it did. The **-full_report** option provides you with this information. In cases where loops vectorize, the **-full_report** option will tell you why specific loops were chosen as vector and parallel loops. In cases where no loops vectorize, the **-full_report** option will tell you why the compiler was unable to vectorize any loops, and will make suggestions that may speed up your program.

Example

The following example uses the same example fragment. The following is the output produced when **-full_report** is invoked, but with the program listing removed. The program listing for **-vreport** and **-full_report** are identical.

```
          Vector Parallel Statements

Stmt   Parallel Choices   Chosen   Reason   Vector Choices   Chosen   Reason

  8                1, 2      2                        1, 2      1      (2)
 12                1, 2      2                     1, 2, 3      1      (2)


(2)    Stride one access influenced selection of vector loop.
```

In the sample program, everything vectorizes and parallelizes, so the information conveyed by the **-full_report** option relates mainly to its choice of loops to vectorize and parallelize. The **-full_report** option divides statements up into four categories: "Vector Parallel" (as above), "Vector", "Parallel", and "Neither Vector nor Parallel". In the report above, the Stardent 1500/3000 compiler is informing the user that it has detected that the outer two loops (loops 1 and 2) can be run in parallel, and that any of the loops can be run in vector around either of the statements. The Stardent 1500/3000 compiler selected the loop that was originally at nesting level 2 to run in parallel, and the loop that was originally at nesting level 1 to run in vector. It chose the vector

loop because it has stride-one access to the most memory locations—while most strides are fine for accessing memory, the Stardent 1500/3000 hardware has been designed to do extremely well on stride one access. Once loop 1 is chosen as the vector loop, loop 2 is the best candidate for parallel execution.

Other examples of reporting output are used to describe program transformations. Below is a list of all the diagnostics that may be printed by the reporting facilities, as well as a rough description of its meaning.

```
Length of vector operation influenced
selection of vector loop.
```

This diagnostic indicates that the vector loop was chosen because it appeared to have the longest length of any of the vector candidates.

```
Stride one access influenced
selection of vector loop.
```

This diagnostic indicates that the vector loop was chosen because it appeared to have more stride one memory accesses than any of the vector candidates.

```
Scatter-gathers influenced
selection of vector loop.
```

This diagnostic indicates that the vector loop was chosen because it had fewer scatter-gather operations than any of the other vector candidates.

```
Length of loop influenced
selection of parallel loop.
```

This diagnostic indicates that the loop chosen to run in parallel was selected over other loops because it appeared to have the longest length.

```
Outmost possible loop selected
as parallel loop.
```

This diagnostic indicates that the compiler selected the outmost possible parallel loop. While this may appear to be the obvious choice, the Stardent 1500/3000 compiler will often interchange better parallel candidates to the outermost position and parallelize them.

```
Nonlinear subscripts eliminated
some possible vector loops.
```

In early versions of the Stardent 1500/3000 compiler, nonlinear subscripts (for example, A(I*I) where I is a loop variable) prevented vectorization of some loops. This diagnostic should no longer appear, as the Stardent 1500/3000 compiler will now generate scatter-gather code for these cases.

```
A function without a vector
analog prevented vectorization.
```

In early versions of the Stardent 1500/3000 compiler, a function call that did not have a vector analog inhibited vectorization of that statement. That is no longer true; such function calls get devectorized as part of the machine-dependent phase of vectorization. As a result, this diagnostic should no longer appear.

```
A function without a parallel
version inhibited parallelism.
```

A function that cannot be called simultaneously by more than one processors will inhibit parallelization of a loop. For technical reasons, such functions also inhibit some vectorization as well. Since all Stardent 1500/3000 math intrinsics are now written to be called by multiple processors at the same time, this diagnostic should no longer appear.

```
PBEST directive mandated
parallel loop selection.
```

The parallel loop was chosen because of a PBEST directive in the code.

```
VBEST directive mandated
vector loop selection.
```

The vector loop was chosen because of a VBEST directive in the code.

```
Dependencies prevent vectorization and
parallelization of  some loops.
```

Some form of memory overlap involving this statement prevents vectorization and parallelization of some loops. If the **-full_report** option is used, the compiler will print out all dependencies that it thinks may be involved.

```
No vector option inhibits vectorization.
```

The compiler did not vectorize any statements, either because of a NO_VECTOR directive or because of the -no_vector option on the command line.

```
Control branches inhibit vectorization.
```

The Stardent 1500/3000 compiler cannot at present vectorize statements under the influence of conditional branches (goto's, assigned goto's, and so on).

```
Cannot vectorize triangular
loop -- unable to interchange.
```

The compiler detected an outer loop that could be vectorized, but was unable to vectorize it because an inner loop has one of its upper bound, lower bound, or strides that depend on the outer loop. Interchanging those loops is an extremely complicated transformation—one that the Stardent 1500/3000 compiler cannot yet do.

```
Loop was split into two loops to
get vector and parallel execution.
```

The compiler could only find one loop to vectorize and parallelize, so it parallelized the strip loop. This does not always give very effective parallelization.

```
Parallelization inhibited
by optimization level.
```

The optimization level was too low to turn on parallelization.

```
Wrap around scalar value
inhibits parallelization.
```

A scalar that wraps around from one iteration to the next permitted parallelism, but not vectorization. For instance, the Stardent 1500/3000 compiler can vectorize the following:

```
T = A(N)
DO I = 1, N
   B(I) = A(I) + T
   T = A(I)
ENDDO
```

The result of the vectorization is shown here:

```
DO iv=1, N, 32
   tv_t(1) = T
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      tv_t(2 + I - iv) = A(I)
      B(I) = A(I) + tv_t(1 + I - iv)
   END DO
   T = tv_t(2 + rv - iv)
END DO
```

However, it cannot parallelize this loop because the scalar value wraps around from iteration to iteration. Doing some minor rewriting (unrolling one loop iteration) can typically improve the performance of such loops.

```
Reduction operation can
only be done in vector.
```

In early versions of the Stardent 1500/3000 compiler, reduction operations such as sum reduction, dot product, and so on could not be done in parallel. Now, only a few, like count reductions and IDAMAX cannot be done in parallel.

```
Only assignment statements
can be vectorized.
```

CALL statements, GOTO statements, and so on have no vector hardware support.

```
Dependencies prevent vectorization.
```

The compiler detected some memory overlap that inhibited vectorization of this statement. **-full_report** will print out the dependencies the compiler found.

```
Dependencies prevent parallelization.
```

The compiler detected some memory overlap that inhibited parallelization of this statement. **-full_report** will print out the dependencies the compiler found.

```
Loop length too small to
justify parallel execution.
```

The compiler detected that the loop could be run in parallel, but it was so small as to not be worth it.

Vectorizer Strategy

The Stardent 1500/3000 vectorizer, unlike vectorizers for many other machines, considers all loops in a loop nest to be candidates for vectorization. Because of this wider range of choices, the Stardent 1500/3000 vectorizer will usually generate code that does better (relative to peak speed) than would be obtained by other vectorizers. However, in a few cases, the wider range of choices may cause the Stardent 1500/3000 vectorizer to vectorize a less optimal loop, resulting in poor vector code. By understanding the evaluation strategy utilized by the vectorizer, you can predict the loop nests in your code where poor code may be generated, and can focus in on the appropriate vreports to confirm your predictions.

In a given loop nest, the Stardent 1500/3000 vectorizer will vectorize only one loop and will parallelize only one loop. The strategy it uses to choose those loops in each loop nest is roughly as follows:

(1) First, it makes one pass over the nesting finding all the loops that can correctly be performed in vector and in parallel.

(2) Next, it determines the best vector loop. The compiler begins to evaluate vectorization possibilities starting from the innermost loop and moving outward.

a) If this is the first loop that has been found that can be done in vector, then obviously choose it.

b) If this loop can be done in vector and an inner loop has been found that can be done in vector, then choose the one with the fewer scatter-gather operations.

c) If there is a choice between two loops with equal numbers of scatter-gather operations, then choose the loop with a larger number of stride-one memory accesses.

d) If there is a choice between two loops with equal numbers of scatter-gathers and equal numbers of stride-one accesses, and they both have known lengths, choose the one with the longer length. If the length of one is known and the other is unknown, then choose the known length if it is larger than a minimum threshold (length 20 at present) known to give full vector speed. Otherwise choose the unknown length.

e) If all factors listed so far are equal, then choose the inner loop.

(3) Determine the best loop that can be done in parallel.

a) Remove the vector loop from consideration.

b) Starting from the outermost loop, find the first loop that is eligible.

c) If there are no candidates, choose the vector loop.

Vectorizing C Programs

Fortran is a language that was designed from its very inception to allow optimization. C, on the other hand, was designed to provide high-level access to machine features—a design criterion that often requires no optimization. As a result, even though the Stardent 1500/3000 Fortran and C compilers use the same intermediate language, vectorizer, and code generator, a program written in Fortran may run many times faster than the identical program written in C. However, if you pay attention to the following language differences, you can write C programs that will run just as fast as the analagous Fortran programs.

(1) The Fortran language guarantees the compiler that if two parameters are passed by reference, and the subroutine to which these parameters are passed stores a value into one of the parameters, the other parameter is not affected. This restriction is very important for optimizing Fortran programs, and it is not present in C. Consider the following Fortran subroutine:

```
SUBROUTINE VCOPY(A,B,N)
DOUBLE PRECISION A(N), B(N)
DO I = 1, N
    A(I) = B(I)
ENDDO
END
```

It is possible for the Fortran compiler to vectorize this construct because the Fortran standard guarantees that A and B are different arrays. The Stardent 1500/3000 C compiler *cannot* vectorize the equivalent C program because the C standard does not make the same guarantee.

```
void vcopy(double *a, double *b, int n)
{
   while(n--)
     *a++ = *b++;
}
```

It is perfectly legitimate to call the C program via the following:

```
main()
{
    double a[100];
    vcopy(&a[1], &a[0],99);
}
```

In this case, the vectorized vcopy subroutine not work correctly. It is illegal in Fortran to call its VCOPY in the equivalent way:

```
PROGRAM MAIN
DOUBLE PRECISION A(100)
CALL VCOPY(A(2), A(1), 99)
END
```

As you might guess, the lack of this restriction in C prevents a large number of seemingly simple programs from vectorizing. The Stardent 1500/3000 C compiler contains an option which circumvents this problem. If you specify **-safe=parms** on the command line for a C compile, you guarantee to the Stardent 1500/3000 C compiler that all subroutines in the compiled files will obey the Fortran aliasing convention—any reference parameter which is into which a value is stored is not aliased to any other reference parameter or globally known variable. The compiler will then use that information in vectorizing those routines, resulting in much more vector and parallel code. Note that the compiler does not check the assumption you have stated, so that if you incorrectly invoke **-safe=parms**, you may cause the vectorizer to generate incorrect code. Thus, if you compile the C example above with the options **-safe=parms -vreport**, you will see the following:

```
                  Vectorized Results From File sample.c
                         Origin -- Line 3

 Line    Stmt    Time    Program

    3       5       4    $$R1 = $$B1;
    3       *       5    for ($$iv = 1; $$iv != $$R1; $$iv += 32) {
    3       *       9       $$rv = MIN($$R1, 31 + $$iv);
    *       *       9       $$vl = $$rv - $$iv + 1;
```

```
3     *      6       DO VECTOR ($$I1 = $$iv; $$I1 != $$rv; $$I1++) {
4    11     36          *(a + ($$I1*8 - 8)) = *(b + ($$I1*8 - 8));
                     }
                 }
```

If you should call this routine with the C main program above, you will get incorrect results, because you have not maintained the assumption that you made to the compiler.

(1) In C, it is not only possible but also very useful to obtain pointers into dynamic storage; in Fortran, that is not possible (except in extremely limited ways). It is very difficult for the Stardent 1500/3000 C compiler to know that references to such pointers do not overlap in any way. For instance, the following simple C program will not be vectorized by the Stardent 1500/3000 C compiler, even with **-safe=parms** specified.

```
main()
{
    double *a, *b;
    int i;
    a = (double *) malloc(100 * sizeof(double));
    b = (double *) malloc(100 * sizeof(double));
    for(i=0; i<100; i++)
       *a++ = *b++;
}
```

The reason that this program cannot be vectorized is that the Stardent 1500/3000 vectorizer cannot ensure that the references to "a" and "b" do not overlap in some way—since C does not have the notion of an intrinsic function, the compiler cannot even rely on the behavior of *malloc* to derive this information. For subroutines such as that above, the compiler option **-safe=ptrs** will enable the vectorizer to automatically vectorize such references. This option guarantees to the compiler that any location which is stored into through a pointer is not also referenced through another pointer as an input location. Thus, when the program above is compiled **-safe=ptrs -vreport**, the following report is issued:

```
                Vectorized Results From File sample.c
                       Origin -- Line 7

Line   Stmt   Time   Program

   7     *      4    for ($$iv = 0; $$iv != 99; $$iv += 32) {
   7     *      8       $$rv = MIN(99, 31 + $$iv);
   *     *      9       $$vl = $$rv - $$iv + 1;
   7     *      6       DO VECTOR ($$I1 = $$iv; $$I1 != $$rv; $$I1++) {
```

```
        8      12      30           a[$$I1] = b[$$I1];
                                     }
                                   }
```

While **-safe=ptrs** will enable many more files to be vectorized, its assumptions are not always met in programs, and users should be careful in using it. For instance, the following variant of the program above will vectorize with **-safe=ptrs**, but the resulting code will be incorrect, because the routine does not meet the assumptions guaranteed by **-safe=ptrs**.

```
main()
{
    double *a, *b;
    int i;
    a = (double *) malloc(100 * sizeof(double));
    b = a;
    a = a + 1;
    for(i=0; i<100; i++)
       *a++ = *b++;
}
```

(1) The Fortran DO loop operates in a very consistent manner; the lower bounds, upper bounds, and step are all evaluated once at the beginning of the loop, and are not allowed to vary within the loop. C *for* loops operate in a different manner; the stride and the upper bound can be legitimately changed during the execution of the loop. This behavior does not map well to vector hardware. Because of this variance, the C front end converts user *for* loops into *while* loops. A pass of the vectorizer will then convert all *while* loops that it can into DO loops. Since the vectorizer only attempts to vectorize DO loops, any unconverted *while* loop will not be vectorized. A very common occurrence in C routines that have been written to be called from Fortran is that the vectorizer cannot determine that the step or bounds of a *for* loop does not vary within the loop. For instance, varying the vcopy routine slightly so that it can be called for Fortran yields:

```
void vcopy(double *a, double *b, int *n)
{
   int i;

   for(i=0; i<*n; i++)
     *a++ = *b++;

}
```

Running this through the vectorizer, even with **-safe=parms** will not result in vector code. The vreport that is generated is as follows:

```
              Vectorized Results From File sample.c
                      Origin -- Line 5

Line    Stmt    Time          Program

   5      *       8           while (i < *n)
                              {
   5      5       6              $$B1 = $$B1 + 1;
   6     10      16              *a = *b;
   6     16      21              a = a + 8;
   6     18      21              b = b + 8;
   5     20       6              i = i + 1;
                              }
   5      *       4           $$R1 = $$B1;
```

As you can see, the *while* loop has not been converted to a DO loop, so that none of the vector optimizations have been undertaken. The reason that the loop has not been converted is that the vectorizer cannot tell that the upper bounds of the loop (*n) does not vary within the loop (the vectorizer should be able to derive this information from the *-safe=parms* option, but there are some technical subtleties that prevent this). The compiler option *-safe=loops* can be used to guarantee to the compiler that the strides and bounds of a *for* loop do not vary within the loop. This does not always enable the compiler to convert the loop into a DO loop (for instance, the *for* loop may contain a branch into it), but it does capture the majority of commonly arising cases. When the fragment above is recompiled with the command line options **-safe=parms -safe=loops** the following vreport is the result:

```
              Vectorized Results From File sample.c
                      Origin -- Line 5

Line    Stmt    Time    Program

   5      *      10     for ($$iv = 0; $$iv != *n - 1; $$iv += 32) {
   5      *      14        $$rv = MIN(*n - 1, 31 + $$iv);
   *      *       9        $$vl = $$rv - $$iv + 1;
   5      *       6        DO VECTOR ($$I1 = $$iv; $$I1 != $$rv; $$I1++) {
   6     10      30           a[$$I1] = b[$$I1];
                           }
                        }
```

If **-safe=loops** does not cause a while loop to be converted into a DO loop, and if there are no jumps into the loop, then

the most likely problem is that the loop increment is not in a pattern that the vectorizer can handle. Varying the form of the increment should yield a vectorizable loop.

(1) Since **-safe=parms -safe=loops** is commonly used in vectorizing C programs, the Stardent 1500/3000 compiler has provided a single option, **-vector_c**, that is exactly equivalent to these two options. Note that if you wish to specify **-safe=ptrs** rather than **-safe=parms**, you can add it on the command line after **-vector_c** and it will be picked up.

(2) Fortran contains the notion of intrinsic functions; if you make a call to *sin* in Fortran, the language guarantees that you are calling the *sin* function built into the systems math library. C does not contain such a notion; it is legitimate for you to have your own function called *sin* which may return the cosine of the argument, or behave in other random ways. Because the compiler cannot count on the behaviour of mathematical functions in C, it is unable to vectorize any loop containing a call to such a function, even though the same loop would trivially vectorize in Fortran. The Stardent 1500/3000 compiler contains a number of pragmas to get around this inhibition. For the functions contained in *libm*, which are the most common cases, the easiest way to get at vector and parallel versions of these functions is to include the header file *vmath.h* in place of *math.h*. The header file *vmath.h* contains the normal declarations that are present in *math.h*, plus a number of pragmas indicating the available vector versions of routines that are present in *libm*. Note that by using *vmath.h*, you must link *-lm* in order to resolve all references, and you may get unexpected behaviour if you link in your own versions of the functions present there.

(3) Use of operators such as "&&" and "||" do not map well to the Stardent 1500/3000 vector hardware or most other vector hardwares. Both of these operators require that the compiler generate code that does not evaluate the right side of them except in cases where that side may determine the truth of the expression. Such code vectorizes poorly, because it requires that the compiler generated a mask vector while operating under a mask vector. For instance, when evaluating a vector *&&*, the compiler must carefully make out the evaluation of the right side of the expression in all cases where the left side is false, and must make the result of that the mask for later vector operations. The

Stardent 1500/3000 hardware (and most other vector machines) does not have this capability, requiring that part of this operation be done in scalar. In many cases, use of the "&" and "|" operators (which allow evaluation of both sides regardless of the truth of either side) will result in better vector code.

(4) Because of the predominance of pointers in C, it is easy to write loops which translate very poorly to vector hardware. For instance, one way a C programmer might obtain a two-dimensional array is the following:

```
main()
{
      double **a;
      int i;
      extern init(double **);

      a = (double **) malloc(100 * sizeof(double *));
      for(i=0; i<100; i++)
            *(a + i) = malloc(100 * sizeof(double));

      init(a);
}

init (double **a)
{
      int i,j;
      for(i=0; i<100; i++)
         for(j=0; j<100; j++)
              (*(a + i)) [j] = (double) i;
}
```

This type of code might be used to create a two dimensional array with ragged dimensions. The Stardent 1500/3000 compiler will vectorize the "j" loop in the routine "init"; however, the resulting code will run relatively poorly, because it is difficult for the compiler to detect that the base array reference "a[i]" does not vary within the vector loop. As a result, scatter-gather code has to be generated. Furthermore, it is possible to coerce the compiler to generate vector code for the outer loop. Since each element of the outer loop selects a different array to be worked on, such code runs extremely poorly, independent of the compiler used.

As the Stardent 1500/3000 compiler becomes better attuned to compiling C, it will do a better job of handling cases such as the vectorizing the inner loop in the above example. However, vectorizing the outer loop will always give poor code, regardless of the machine on which it is run. The

summary of this is to avoid using constructs such as **p or other more exotic pointers in order to generate good vector code.

The Stardent 1500/3000 Compilation System and Fortran

This section of the chapter discusses things that specifically apply to using the Stardent 1500/3000 compilation system when coding in Fortran. The topics discussed are invoking the compiler, compiler options (not previously discussed), compilation control statements, the format of the compiler listing, and identifying errors.

The discussions in this section assume that you have read the first section of this chapter which talks about concepts that apply to using the compilation system for any language.

Calling The Fortran Compiler

The Stardent 1500/3000 Fortran compiler command syntax allows you to intermix multiple options and file specifications

fc [*options*] [*filespec*] [[*options*] [*filespec*]]

options
: is a set of options, formatted as described below.

filespec
: is a UNIX file path by which a source file may be accessed. The Stardent 1500/3000 Fortran compiler can handle a mix of different programming languages and object files on the same command line. For example, file names can end with the characters *.f*, *.c*, *.o*, *.a*, or *.s*. If the compiler does not know how to create or to access a file you have specified, an error notice is generated.

EXAMPLE

An example *filespec* is:

```
/usr/test/fortran/myfort.f
```

In the example, the pathname part of the *filespec* is */usr/test/fortran* and the filename is *myfort.f*.

NOTE
Filenames or parts of pathnames (between /'s) cannot exceed 14 characters.

Form of the Options

Stardent 1500/3000 compiler options, when used, take one of the forms shown in Table 2-5.

Table 2-5. Form of Compiler Options

Form	Option
1	*-option*
2	*-option symbol_or_file_path*
3	*-option number*
4	*-option symbol*
5	*-option=number*
6	*-option=symbol_or_file_path_list*

A *symbol_or_file_path_list* can be empty; if the list is not empty, the list should be a comma-separated list of symbols and file pathnames. Generally, options have the first, fifth, or sixth forms; the second, third, and fourth forms are loader options which are passed on to the loader.

A *symbol_or_file_path_list* must not include blanks. The symbols generally allow the use of letters, numbers, and special characters that are not otherwise meaningful to the shell or to the option parser.

Options are applied to the compiler in the order written on the command line, from left to right; similarly, suboptions (the items to the right of an equal sign in form 6) are applied to an option in the order written, from left to right. Options for the loader are passed to the loader in the order of the command line from left to right. Options and filenames may be intermingled so that libraries may be searched in the proper order with reference to particular object files; **-ltag** loader options must sometimes precede filenames. However, other options are collected and applied all at once to each of the processes invoked; placement of an option such as **-O1** after a filename still causes the **-O1** option to be applied to all files compiled.

Fortran Compiler Options

Options direct the compiler in its operations. Table 2-6 lists the possible options followed by a detailed explanation. This table does not duplicate the options already mentioned in the first section of this chapter that apply to both Fortran and C programs.

Several options have extensive lists of possible suboptions, including the suboptions **all** and **none** and, generally, a negative option prefixed with the characters *no*. Many options appear in both positive and negative forms (for example, **-list** and **-nolist**). These options are switches; the last appearance in the command line controls the position of the switch. Each option has a default position indicated in italics.

-all_doubles

-all_doubles is useful for converting a single precision code that has been run on a machine such as the Cray (where singles are 8 bytes long) to efficient execution on the the Stardent 1500/3000. When this switch is enabled, all single precision variables, regardless of declaration, are promoted to double-precision and all complex variables are promoted to double complex. Any calls to single precision intrinsic functions are converted to the analagous double precision function. All INTEGER*4 variables are allocated eight bytes of storage, even though arithmetic is still done only on four bytes. All direct memory copies, even of integer variables, are done in eight byte quantities.

In order to correctly parse single precision codes, the Stardent 1500/3000 compiler does not do the conversion until *after* the code has been parsed. This has the effect of correctly converting a true single precision code; however, it may have unexpected side effects with respect to constant conversion and such. For instance, the following fragment, if compiled **-all_doubles**

```
T = AMAX1(1.0e40, 1.0e50)
```

will have the effect of setting "T" to +∞. The reason is that the compiler first parses the code as singles (in order to correctly parse the AMAX1); both of the constants are larger than can be represented in a single precision number, so they get converted to +∞.

Table 2-6. Fortran Compiler Options

Option	Negative Form	Description
-all_doubles		All arithmetic is double precision; 8 byte integers
-blanks72		Pad source file lines with blanks
-continuations=*n*		Specify acceptable number of continuation lines
-cpp		Invoke the C preprocessor
-cross_reference	-nocross_reference	Generate a cross-reference listing in listing file
-debug	-nodebug	Add debug data to object file
-double_precision	-nodouble_precision	Use double precision for all undeclared variables
-d_lines	-nod_lines	Compile debug lines that start with D in column 1
-fast		Enable optimizations which may affect precision
-implicit	-noimplicit	Untype all variables
-include=*pathname*		Specify a directory to search for included files
-include_listing	-noinclude_listing	Add included files to the listing file
-i4	-noi4	Interpret all INTEGER and LOGICAL as though declared *4
-list	-nolist	Generate a listing file
-messages	-nomessages	Allow printing of warning messages
-no_directive		Do not apply directives during compilation.
-object	-noobject	Generate an object file
-onetrip	-noonetrip	Make sure all DO loops execute at least once
-save	-nosave	All variables declared are saved
-standard	-nostandard	Check for standard Fortran 77 usage
-verbose		Use verbose message output

After the parse, the compiler converts this representation to double-precision; however, the original information regarding the value is lost, so that the resulting double-precision numbers are also $+\infty$. Use of the **-double_precision** option in conjunction with this option can have the effect of avoiding problems such as this, for codes that use generic functions rather than specific intrinsics.

-blanks72

Pad all lines in the source file on the right to column 72. Lines longer are untouched. Does not work with **-cpp**, **-P**, or **-E**. Affects only *.f* files specified in the command line; it does not affect files that are included through the include option. This option allows codes that have string constants or Hollerith constants that expect card boundaries to compile as expected on the Stardent 1500/3000.

-continuations=*n*

Set the number of continuation lines the compiler accepts for any statement. The number *n* may range from 0 to 99; the default is 19.

-cpp

Invoke the C preprocessor. If this is not used, all source lines with a pound sign (#) in column 1 are going to be silently ignored. The options **-P** and **-E** always invoke the preprocessor even if it is not used.

NOTE
Line numbers on messages may be confused when **-cpp** is used. You can match up line numbers with the source file by using the **-list** option. Look at the first column of numbers in the *xx.L* file that is produced.

-cross_reference

Generate a cross-reference, if a listing is generated. The default is **-nocross_reference**.

-nocross_reference

Turn off any cross-reference listing.

-debug

Generate information for the Stardent 1500/3000 debugger. The default is **-nodebug**. Options **-debug** and **-g** are synonymous.

-nodebug

Do not generate debugging information.

-d_lines

Compile source file lines with a 'D' or a 'd' in column 1. The default is **-nod_lines**. The 'D' or 'd' is treated as if a blank were substituted for it.

-nod_lines

Treat source file lines with a 'D' or a 'd' in column 1 as comments.

-double_precision

Change all undeclared floating point variables and all variables that are declared in REAL statements to be double precision. It does *not* affect variables that are declared in REAL*4 statements. In a similar fashion, all complex variables are promoted to double complex. Constants and generic intrinsics are adjusted appropriately. Since this conversion is performed at the time that routines are parsed, it can allow for more intuitive conversion of constants than does the **-all_doubles** option. However, it will not convert intrinsic calls that are not generic, so that codes that use specific intrinsics will probably not compile. Integer variables are unaffected by this option. **-nodouble_precision** is the default.

Be aware that use of this option affects functions and their arguments, which may cause them not to work. As an example, *cputim*(3F) does not work with this option unless it is explicitly declared as REAL*4. Be sure to understand the implications of using this option with functions.

-nodouble_precision

Treat floating point variables and constants by the Fortran 77 rules; specifically, compile them as REAL*4 unless otherwise specified in the source.

-fast

This option enables a number of optimizations that may slightly affect the precision of the resulting answer. For instance, when this option is enabled the optimizer will convert expressions such as a/b/c into a/(b*c), which may change answers slightly in the last bit or two. On the Stardent 1500 hardware, this option at link time will also cause a faster, but less accurate, set of math libraries to be loaded. Faster math libraries are not loaded on Stardent 3000, because the Stardent 3000 hardware contains added operators that allow precise answers to be obtained just as quickly as the fast versions would obtain less precise answers. In general, this option will provide extra speedup at some small sacrifice in precision.

-g

Generate information for the Stardent 1500/3000 debugger. The default is **-nodebug**. Options **-debug** and **-g** are synonyms.

-implicit

Make all variables in a program untyped. Using this option, all variables must be declared or an error occurs. This option has the same effect as an **IMPLICIT NONE** statement at the beginning of each routine in a source file. **-noimplicit** is the default.

-noimplicit

Follow the ordinary Fortran 77 typing rules.

-include=*pathname*

Specify a *pathname* to a directory to be searched to locate files specified in Fortran INCLUDE statements.

-include_listing

List included source as well as the original source file when generating a listing. The default is **-noinclude_listing**.

-noinclude_listing

Do not copy included source files to the listing.

-i4

Interpret INTEGER and LOGICAL declarations as if they had been written INTEGER*4 and LOGICAL*4.

-noi4

Interpret INTEGER and LOGICAL declarations as if they had been written INTEGER*2 and LOGICAL*2. The default is **-i4**.

-list

Generate a listing from the compilation. The listing file is named after the source file. The default is **-nolist**.

-nolist

Do not generate a listing file.

-messages

Allow warning messages to be printed. The default is **–nomessages**.

-nomessages

Do not allow warning messages to be printed.

-no_directive

Do not apply compiler directives. The default is to apply directives.

-object

Generate object files from this compilation. This is not quite the same as the **-c** option, which precludes loading; **-noobject** precludes even the generation of object files.

-noobject

Do not generate object files from this compilation; check correctness of the source code only. The default is **-object**.

-onetrip

Generate code to guarantee that all **DO** loops execute at least once. This is a compatibility feature for programs originally written for use with Fortran 66. The default is **-noonetrip.**

-noonetrip

DO loops may execute zero times.

-save

Save all declared variables. The default is **-save.**

-nosave

Do not save all variables. This has no effect because Stardent 1500/3000 Fortran automatically saves all variables.

-standard

Check for standard Fortran 77 usage and flag Stardent 1500/3000 extensions with warnings. This option may appear in forms 1 or 6 with the suboptions shown in Table 2-7.

Table 2-7. Suboptions for -standard option

syntax	nosyntax
source_form	nosource_form
all	none

The default is **-nostandard**. If this option appears in form 1, it is the same as **-standard=all**.

-nostandard

Turn off checking for extensions to Fortran 77. This option has the same structure as **-standard**.

-verbose

Generate more messages tracking the progress of the compilation. As a default, these additional messages are suppressed.

Compilation Control Statements

In addition to the compilation control statements that are discussed in the first section of this chapter, Stardent 1500/3000 Fortran offers one other, the **OPTIONS** statement. An **OPTIONS** statement allows you to specify certain compiler options instead of specifying them on the command line. The **OPTIONS** statement is described in detail in chapter 2, *Fortran Statements*, in the *Fortran Reference Manual.*

Format of the Fortran Listing

The listing of the compiler output contains the following:

- A section that contains your source code with line numbers prefixed to each source line.
- A storage map.
- A summary of the compilation, including the settings of the options and a summary of the number of errors.

Source Code Section

This section contains the program source code as it appears in your source program. The example below shows a sample source code listing from a subroutine.

EXAMPLE

```
(date)    (time)
Source Listing  File: /tmp/ftest.f

   19   *     Function subprogram unit follows.
   20         INTEGER*4 FUNCTION nfunc(k)
   21         nfunc = 0
   22         DO 10 i = 1,k !Loop to compute sum.
```

```
 23          nfunc = nfunc+i
 24       10 CONTINUE
 25          RETURN  !Return value in function name.
 26          END
```

Storage Map

After each program unit, the compiler listing includes a storage map that lists information about the program unit. The storage map includes the following:

- Entry points: all entry points into the program are listed. If an entry point returns a specific data type, that data type is declared along with the entry point.
- Storage blocks: the storage usage of the program is listed.
- Variables: the name and data type of each variable declared for use in this program unit is listed, as well as its offset addresses within the program unit.
- Arrays: the name, size, location, and type of each array is listed.
- Intrinsic functions: all intrinsic functions that are called from within this program unit are listed.
- Statement functions: all statement functions are listed by name and by their declared data type.
- Externals: all externals and their declared data types are listed.

EXAMPLE

```
(date)   (time)
Symbol Storage Map   File: /tmp/ftest.f

ENTRY POINTS
  Offset  Type  Name

       0  I*4   NFUNC

STORAGE BLOCKS
    Size  Block

       4  local
      28  constants
```

```
VARIABLES
  Offset     Size  Type  Block            Name

       0        4  I*4   local            I
     ---        4  I*4   dummy            K

ARRAYS
  Offset     Size  Type  Block            Name

PARAMETERS (Offset is within Storage Block 'constants')
  Offset  Type  Name                              Value

INTRINSIC FUNCTIONS
  Name

STATEMENT FUNCTIONS
  Type  Name

EXTERNALS
  Type  Unit_Kind      Name
```

Cross Reference

If, as part of the command line, you specify the command option **-list** and **-cross_reference**, then a cross-reference listing is generated with the listing file. This cross-reference listing shows all symbols and labels, listing each symbol and the line numbers on which it appears. It also lists labels, showing the line number at which each label is defined and the line number at which a reference to that label occurs.

EXAMPLE

```
(date)  (time)
Symbol Cross Reference  File: /tmp/ftest.f

Symbol                            Line Number(s)

EXONE                                 1
N                                    10     12     13
NFUNC                                 7     12
SUM                                   7     12     13

(date)  (time)
Label Cross Reference  File: /tmp/ftest.f

Label  Defined  References(s)

33         14         13
```

Compilation Summary

The summary section lists the command qualifiers in effect for this compilation as well as statistics about the source files processed. Here is a sample compilation summary.

EXAMPLE

```
(date)   (time)
Source Listing  File: /tmp/ftest.f

COMMAND QUALIFIERS

  -optimize=0 -assemble -list
  -include=/la/cross/include /tmp/ftest.f

-BACKEND=ffe3,ffe1
-CHECK=(NOBOUNDS,OVERFLOW,NOUNDERFLOW)
-STANDARD=(NOSYNTAX,NOSOURCE_FORM)
-INCLUDE=/la/cross/include
-CONTINUATIONS=19  -FE_FISHERBURKE  -FE_VERBOSE
-I4  -LIST  -OPTIMIZE=0  -PADDING=3  -S  -WARNINGS


FRONT END STATISTICS

  Input File:            /tmp/ftest.f
  Base Filename:         ftest.f
  Source Processed:      26 lines
```

Errors and Compiler Diagnostics

The Stardent 1500/3000 Fortran compiler identifies syntax errors and language violations. It outputs various error messages to your terminal, describing the source of the error, usually with enough information about the error to allow you to correct it.

If you have requested a listing file to be produced (**-list** option), any errors in your source file are printed immediately following the statement containing the error.

If you need help in interpreting the error message, you'll find the error messages and their meanings listed in *Appendix A* in the *Fortran Reference Manual.*

The Stardent 1500/3000 Compilation System and C

This section of the chapter discusses things that specifically apply to using the Stardent 1500/3000 compilation system when coding in C. The topics discussed are elements of the compiler and its invocation, compilation control statements, extensions to the language such as a new storage class keyword, and identifying errors.

The discussions in this section assume that you have read the first section of this chapter which talks about concepts that apply to using the compilation system for any language.

A sound knowledge of the C programming language is also assumed. *C: A Reference Manual* by Samuel P. Harbison and Guy L. Steele Jr., Second Edition, Prentice-Hall, 1987 is an excellent modern reference on the details of C, and is mentioned in this section. More information on the proposed ANSI standard for C may be found in the draft standard dated October 10, 1986. Some C language features have been derived from ideas developed in the C++ language; more information about C++ may be found in *The C++ Programming Language* by Bjarne Stroustrup.

Elements of the Stardent 1500/3000 C Compiler

This section describes the elements of the Stardent 1500/3000 C compiler, paying particular attention to the front end. The C language front end is based on the front end distributed with System V Release 3 of AT&T UNIX. This includes the following programs and utilities:

cc	the C language compiler
cpp	the C language preprocessor
ld	the link editor
cb	a C program beautifier
regcmp	a program to compile C program regular expressions

See the *Commands Reference Manual* for full details of these utilities.

Stardent 1500/3000 C supports ANSI function call semantics. For a function declared with a function prototype, the Stardent 1500/3000 C compiler enforces strict type matching rules between the actual arguments in a call of the function and the formal arguments in the definition of the function.

Calling the Stardent 1500/3000 C Compiler

You invoke the Stardent 1500/3000 C compiler with the *cc* command.

cc [*options*] [*filespec*] [*options*] [*filespec...*]

NOTE
Options and file names may be intermingled.

options
: is a set of options, formatted as described below.

filespec
: is a UNIX file path by which your source file may be accessed.

The compiler understands filenames ending with the suffix **.c** to be C source files.

For example, *cc program.c* is the most basic form of the command and performs the following actions on *program.c*:

- invokes the C preprocessor, *cpp*
- invokes the C compiler itself
- invokes the link editor, *ld*

When the whole compilation is finished, the resulting executable file, *a.out*, is left in the current directory. You can rename *a.out* to any legal filename.

C Compiler Options

Options direct the compiler in its operations. Table 2-8 lists the options followed by an explanation. This table does not duplicate the options already mentioned in the first section of this chapter that apply to both Fortran and C programs.

Table 2-8. C Compiler Options

Option	Description
-W	Fully suppress compiler warnings
-n	Suppress the standard C startup routine
-safe=parms	Parameters are not affected
-safe=ptrs	Pointers are not affected
-safe=loops	Upper bounds do not vary
-v	Use verbose message output
-vector_c	Equivalent to **-safe=loops -safe=parms**
-w	Suppress compiler warnings

-W

Fully suppresses all warning messages from the compiler. The difference between this option and **-w** is that **-w** will print out one warning message indicating the number of other warning messages which have been suppressed.

-n

Suppress the standard C startup routine.

-safe=parms

The program being compiled obeys the Fortran standard with regard to parameter passing. Storing a value through any reference parameter does not affect any value read from either a global variable or another reference parameter.

-safe=ptrs

Extend **-safe=parms** to all pointers; a store through any pointer does not affect any value read through any other pointer or through a global variable.

-safe=loops

When you use this option, it guarantees to the compiler that all **for** loops within the program have upper bounds that do not vary within the loop. This is necessary for loops to vectorize when they have array references or (*) values as upper bounds.

-v

Generate more messages tracking the progress of the compilation. As a default, these messages are suppressed.

-vector_c

Equivalent to specifying **-safe=parms -safe=loops** on the command line.

Extensions to the Compiler

This section discusses new features that have been incorporated into the compiler beyond those that are typically found in Kernighan and Ritchie based compilers. These features include new compilation control statements, a new storage class, argument (function) prototypes, reference (address of a variable) arguments in declarations and calls, and two new compiler directives.

Compilation Control Statements

In addition to the compilation control statements that were identified in the first section of this chapter, Stardent 1500/3000 C supports two new control lines. The format of these statements is

#ident "*comment*"
#pragma *identifier*

The **ident** line puts the comment string into the **.ident** section of the *.o* file. It is usually used for version control. The loader *ld* and other tools interpret and list these comments. By default, all ident sections in the *.o* files are concatenated in the *a.out* file.

An illustration of the **#ident** statement can be found within the Stardent 1500/3000 system software itself. The source files to the Stardent 1500/3000 system are kept under SCCS control; at the

top of every file is a control line similar to

```
#ident "%W%" %G%
```

When such a file is checked out of SCCS, SCCS replaces the %W% and %G% with the file version and the date the file was last modified, so that during a build, all Stardent 1500/3000 software is tagged with version control inside **#ident** statements. The Unix command **what** will print this information to the screen, so that you can tell exactly what versions of which files went into creating the software. For instance, type this command

```
what /usr/bin/what
```

The output shows you the versions of the system headers and source files used to create the **what** command itself.

The *#pragma* directive is used as a general mechanism for providing optimization information to the compiler. It may also be used to provide runtime information which can be used to generate more efficient code. Refer to *Chapter 2, Efficient Programming Techniques* for a discussion of how to use this control line.

Argument (Function) Prototypes

Stardent 1500/3000 C includes the ability to declare the types of arguments for functions. When prototypes are used, the compiler will check that functions have been called with the right number and types of arguments, thereby preventing errors resulting from mismatched arguments. Prototypes also allow users to pass **floats** as float values, rather than having them coerced to **doubles** as happens in most C compilers.

Using function prototypes requires changing both the declaration and the definition of a function. To make a declaration into a prototype declaration, the types of all the parameters are listed in order. For instance, consider the following function declaration:

```
extern int astwalk();
```

This function could have a prototype declaration in a header file as follows:

```
extern int astwalk (int *, int, int, int (*) (int *));
```

This declaration states that astwalk takes four parameters: a pointer to an integer, an integer, an integer, and a pointer to a function which returns an integer and which takes a pointer to an

integer as an argument. In general, inserting the parameter types is very straightforward and simple. The trickiest case, which is illustrated above, is passing a function to another function; this case arises rarely, however.

The function definition point is equally as simple to change. Whereas the definition point for astwalk in a non-prototyped C might look like the following:

```
int astwalk(node, type_of_walk, walk_destructive, walk_function)
    int *node;
    int  type_of_walk, walk_destructive;
    int  walk_function();
```

in prototyped form it would appear as the following:

```
int astwalk (int *node, int type_of_walk, int walk_destructive,
             int (*walk_function) (int *))
```

To create a prototype, the argument types and names are moved up into the definition statement itself. Note that the declaration statement will also optionally take the parameter names, so that the same format can be used for both definitions and declarations.

Because parameters are passed differently in prototyped functions than in non-prototyped functions, you should be very careful not to mix the two. To give a specific instance, suppose that the function **foo** is declared as follows:

```
float foo(float x)
```

It is called as shown below in function **bar**. The file that defines function **bar** has no prototype declaration available.

```
void bar()
{
    float x, foo();
    x = 1.0;
    x = foo(x);
}
```

Since there is no prototype declaration in **bar**, the compiler assumes that the user is using old-style argument passing, and so converts x to a double before calling **foo**. However, since **foo** is compiled with a prototype definition, the compiler when compiling **foo** expects to receive a float. This will cause unexpected results.

For a complete discussion of this topic refer to the reference manual by Harbison and Steele and the ANSI draft standard.

Storage Class threadlocal

Stardent 1500/3000 C includes an additional storage class keyword, *threadlocal,* that permits you to allocate storage that is local to each thread cooperating on the task to be performed. The threadlocal keyword allows you to declare an uninitialized variable that is passed to each thread. Refer to thread in the *Commands Reference Manual* for an example of a multithreaded task. You declare such a variable as shown in the following example:

```
threadlocal int i;
```

The loader collects all the threadlocal storage together and puts it in a contiguous block of virtual memory. The operating system then marks these pages so that upon a *thread* call, each thread gets a separate copy of these locations.

Note that the primary place in which threadlocal should be used is at storage definition points. For instance, to create an externally known threadlocal variable, the following definition might be used:

```
threadlocal int proc_no;
main()
{}
```

The position of the declaration establishes the variable **proc_no** as being externally known. Note that this does **not** establish a definition point for *proc_no,* so that the same declaration can be used in several files and the loader will combine them together without indicating any user error. Alternatively, *proc_no* can be declared by using the following declarative statement in other files:

```
extern int proc_no;
```

The loader, when combining attributes for variables, will make an external variable threadlocal if the threadlocal attribute appears on any declaration. This can be particularly important, because the Stardent 1500/3000 compiler will not let you apply both *threadlocal* and *extern* to the same declaration.

Similarly, the attribute *static* can be applied to threadlocal variables, when they are to be known only within the current file. For instance, if *proc_no* were only to be used within one file, then the following declaration would be more proper:

```
static threadlocal int proc_no;
main()
{}
```

When a threadlocal variable is declared within a procedure, it automatically assumes the attribute *static.*

Threadlocal variables cannot be initialized by initialization statements. Also, even though the Stardent 1500/3000 compiler will accept threadlocal on a procedure definition, it should not be used by Stardent 1500/3000 programmers. Threadlocal procedures are used internally by Stardent in several very special cases; in those cases, the programmer is guaranteeing that the procedure obeys some very stringent restrictions. The **pproc** directive is a much more flexible and easy to use method of creating parallel procedures.

Comment Delimiters

The Stardent 1500/3000 C compiler supports "//" as a form of comment delimiter. Outside of strings and comments, everything on a line to the right of a "//" is interpreted by the compiler as a comment. Here is an example:

```
int astwalk (node, type_of_walk)

int   *node;       // node is the address of the location to be walked
int   type_of_walk   // indicates the type of walk to perform
{}
```

Vector Math Functions

The Stardent 1500/3000 C compiler will automatically (at higher optimization levels) replace sequential calls to the math functions in *libm* with calls to analagous vector functions, where appropriate. In order to allow the compiler to perform this transformation, you must guarantee to the compiler at compile time that you really intend to link in *libm* at link time. The easiest way to do this is to include the header file *vmath.h* (from /usr/include). *vmath.h* is an extension of *math.h* that not only includes all the declarations present in *math.h,* but also additional pragmas indicating the appropriate vector analogs for each of the math functions. Simply changing all references to *math.h* to *vmath.h* should allow you to take advantage of this feature.

Compiler Directives

The Stardent C compiler supports one compiler directive that is not directly related to optimization, and which can be very useful on Stardent 1500 hardware. This pragma, **NO_FLOAT**, informs the compiler that a program does not use the floating point unit of the Stardent 1500. On Stardent 1500 hardware, floating point arguments to functions are passed in the vector register file. Because the vector register file is very large, the time to context switch programs that touch it is longer than the time to context switch programs that do not touch it. As a result, the Stardent 1500/3000 operating system and compilers cooperate to carefully keep track of programs that do not use the floating point unit, and hence can use the shorter context switch.

The one place in which this cooperation cannot work is with *vararg* programs. With a *vararg* program, particularly one that just immediately calls another *vararg* program with the same arguments, the compiler must assume that floating point arguments are present, and will make use of the floating point unit in setting up these arguments for the next call. In many cases (such as the Unix kernel code and system commands), no floating point values are ever passed, and this use of the floating point unit serves only to lengthen context switch time. The **NO_FLOAT** pragma exists for users to inform the compiler that a *vararg* procedure does not have any floating point arguments, so that the compiler may avoid touch the floating point unit in argument transmission.

EXAMPLE

The following is an example of when you might want to specify NO_FLOAT.

```
#include varargs
#pragma NO_FLOAT

int printf(char *format, va_del, va_list)
{
        va_list ap;
        va_start(ap);
        count = duprint(format, ap, stdout);
        va_end(ap);
        return(error);
}
```

Address (&) Arguments

Stardent 1500/3000 C will allow the address of constants to be taken as arguments, and supports the use of & arguments in declarations and in function calls. When an & argument is used in a function declaration, that argument is then passed by reference. The compiler will correctly generate code to dereference the parameter in the appropriate locations, without any intervention on the programmer's part. The semantics for using this are the same as in the C++ language.

EXAMPLE

```
main()
{
        f(&5);
}
```

The Stardent 1500/3000 compiler will accept this syntax, and will create a local variable to hold the constant. The address of this local variable will be passed down to the function.

EXAMPLE

The following is an example of coding an & argument in a function declaration in which **x** is passed by reference.

```
void f(int &x)
{
    x++;
}
```

This function will behave exactly like the more conventionally coded function

```
void f(int *x)
{
    (*x)++;
}
```

USING LIBRARIES AND THE LINK EDITOR

CHAPTER THREE

This chapter is divided into two sections:

- The first section discusses libraries, and how to create and manipulate them using the *ar* command. It also provides an informal discussion of the libraries provided on Stardent 1500/3000.

- The second section provides a general introduction to the link editor, *ld*, describes some of the commonly used options for the command, and provides information on using archive libraries with the link editor.

Archive Libraries

A library is a collection of related object files or declarations that simplify programming effort. It is common practice in UNIX system computers to keep modules of compiled code (object files) in archives; by convention, they are designated by a *.a* suffix. The most common use of an archive file, although not the only one, is to hold object modules that make up a library. The library can be named on the link editor command line (or with a link editor option on the *cc* command line). Specifying the link editor option (-l*libraryname*) causes the link editor to search the symbol table of the archive file when attempting to resolve references.

An archive file can also be used to pack together files other than those that end in *.o* (object files). This can be useful when you want to keep blocks of information stored together. Be aware that non-object files in an archive can not be accessed by the loader.

Creating an Archive File

The *ar* command is used to create an archive file, to manipulate its contents and to maintain its symbol table. The structure of the *ar* command is a little different from the normal UNIX system arrangement of command line options.

When giving the *ar* command, you also must include an argument that consists of a single letter that defines the type of action that is to be performed. The argument is one of the following single characters: *drqtpmx*. The actions include adding, deleting, replacing, appending, or extracting files from the archive, moving them within the archive, and listing the table of contents of the archive. This command argument character may be combined with one or more additional characters from that modify the way the requested operation is performed. These optional characters are *vcls*. The secondary options include a verbosity level for the output from the command, redirection of the temporary working file area and the ability to force a regeneration of the symbol table if the archive is composed entirely of *.o* files.

The syntax of the command is as follows:

ar *key* [*posname*] *afile* [*name*]...

key is one or more characters that define the action to be taken

posname is the name of a member of the archive; and may be used with some optional key characters to make sure that the files in your archive are in a particular order.

afile is the name of your archive file. By convention, the suffix *.a* is used to indicate the named file is an archive file. (*libc.a*, for example, is the archive file that contains many of the object files of the standard C subroutines.)

name is the file that is subjected to the action specified in the *key*. One or more *names* may be furnished.

Refer to the *Commands Reference Manual* for specific details on the *key* field and using the *ar* command.

EXAMPLE

The following shows a typical scenario for creating and updating archive libraries. You can create an archive file that contains the modules used in a sample program, **restate**. The command to do this might be as follows:

```
ar -q rste.a restate.o oppty.o pft.o rfe.o
```

If these are the only *.o* files in the current directory, you can use shell special purpose characters as follows:

```
ar -q rste.a *.o
```

The *-q* key creates the archive library. The next step is to reorder the files for access, which creates a table of contents for the archive. This is done as follows, still using the above example:

```
ar -ts rste.a
```

When you want to replace existing source files with updated files, the following can be used.

```
ar -rs rste.a oppty.o
```

Recent Berkeley BSD systems contain a program called *ranlib*,, which is used to speed access to programs stored in a library. The *-ts* option is equivalent to *ranlib*.

Loading Libraries and Their Order

During the link edit phase of the compilation and link edit process, copies of some of the object modules in an archive file are loaded with your executable code. By default the *cc* command that invokes the C compilation system causes the link editor to search *libc.a*. If you need to point the link editor to other libraries that are not searched by default, you do it by naming them explicitly on the command line with the **-l** option. The format of the **-l** option is **-l***tag* where *tag* is the library name, and can be up to nine characters. For example, if your program includes functions from the *curses* screen control package. To use the curses package, specify the following option:

-lcurses

Using this option to the link editor causes it to search for */lib/libcurses.a* or */usr/lib/libcurses.a* and use the first one it finds to resolve references in your program.

When you are programming in Fortran it is **not** necessary to use the options **-lm** and **-lc**, which cause the C math and standard libraries to be loaded. The Fortran compiler already loads these two libraries. In fact, using these options with Fortran can have a negative effect if it causes things to load out of order or causes name conflicts.

In cases where you want to direct the order in which archive libraries are searched, you may use the **-L***dir* option. Assuming the **-L** option appears on the command line ahead of the **-l** option, it directs the link editor to search the named directory for *libtag.a* before looking in */lib* and */usr/lib*. This is particularly useful if you are testing out a new version of a function that already exists in an archive in a standard directory. Its success is due to the fact that once having resolved a reference the link editor stops looking. That's why the **-L** option, if used, should appear on the command line ahead of any **-l** specification.

More information on loading libraries them can be found in later in this chapter and the *Commands Reference Manual.*

Available Libraries

The following table lists the libraries that are available. The link editor recognizes certain abbreviations as an indicator to use specific libraries, and these abbreviations are provided in the table, along with and a short description of the library's contents.

Table 3-1. Available Libraries for C

Library	Abbreviation	Contents
libc.a	-lc	Standard I/O libraries
libcrypt.a	-lcrypt	Encryption subroutines
libcurses.a	-lcurses	Teletype output
libdbm.a	-ldbm	UNIX database management functions
libgen.a	-lgen	General utilities for building commands
libl.a	-ll	Library for *lex*
libm.a	-lm	Math routines
libmalloc.a	-lmalloc	Improved memory allocator functions
libns.a	-lns	Name server library
libnsl.a	-lnsl	Network service library
librpc.a	-lrpc	Remote procedure calls for the client side of NFS
librpcsvc.a	-lrpcsvc	Remote procedures for the server side of NFS
liby.a	-ly	Library for *yacc*
libyp.a	-lyp	Yellow pages for network services
libPW.a	-lPW	Programmer's Workbench functions

Creating a Library Abbreviation

All libraries can be abbreviated when they are placed on the command line. This is done by removing the *lib* prefix and replacing it with **-l**. The *.a* at the end of the filename is also removed. Refer to the previous table of libraries for examples.

When creating your own library, you may abbreviate the name. However, if you have not placed your library in one of the standard areas that the loader searches, you'll have to remember to include the **-L** option, giving the directory to search, and follow that by the normal abbreviation of the library name, as defined above.

The ld command

The basic function of the link editor is to combine object files into an executable program. The link editor combines several object files into one, performs relocation, resolves external symbols, incorporates startup routines, and supports symbol table information used by *dbg*. You may, of course, start with a single object file rather than several. The resulting executable module is left in a file named *a.out*. The typical user, however, seldom invokes *ld* directly. A more common practice is to use a language compilation control command (such as *cc*) that invokes *ld*.

The *ld* command invokes the link editor directly. The *ld* command has 11 options, but this section only describes four of the most commonly used options. These options should be fed to the link editor by specifying them on the *cc* or *fc* command line if you are both compiling and linking with the single command, which is the usual case.

The format of the *ld* command is as follows:

ld [*options*] *filespec*

options — is a set of options; refer to the *Commands Reference Manual* for a complete listing.

filespec — is an object file or an archive library. Any file named on the *ld* command line that is not an object file (typically, a name ending in *.o*) is assumed to be an archive library.

Following are descriptions of two commonly used *options* for the *ld* command.

NOTE
When loading using the *fc* command, it is not necessary to use the **-lm** or the **-lc** options. These two libraries are loaded by *fc*. A library is searched when its name is encountered, so the placement of the option on the command line is important.

-l*tag* directs the link editor to search a library **lib***tag***.a**, where *tag* is up to nine characters. For C programs, *libc.a* is automatically searched if the *cc* command is used. The **-l***tag* option is used to bring in libraries not normally in the search path such as *libm.a*, the math library. The **-l***tag* option can occur more than once on a command line, with different values for the *tag*.

The **-l***tag* option is related to the **-L** option.

-L*dir* changes the **lib***tag***.a** search sequence to search in the specified directory before looking in the default library directories, usually */lib* or */usr/lib*. This is useful if you have different versions of a library and you want to point the link editor to the correct one. It works on the assumption that once a library has been found no further searching for that library is necessary. Because **-L** diverts the search for the libraries specified by **-l***tag* options, it must precede such options on the command line.

Refer to the *Commands Reference Manual* for a thorough explanation and discussion of all of the available *options*.

ld and Archive Libraries

Each member of an archive library (for example *libc.a*) is a complete object file. Archive libraries are created with the *ar* command from object files generated by *cc* or *fc*. An archive library is always processed using selective inclusion. That is, only those members that resolve existing undefined-symbol references are taken from the library for link editing.

The **–l** option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. However, they need not be so. Furthermore, archive libraries can be specified without using the **–l** option by simply giving the (full or relative) UNIX system file path.

The ordering of archive libraries is important because for a member to be extracted from the library it must satisfy a reference that is known to be unresolved at the time the library is searched. Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. *ld* cycles through this symbol table until it has determined that it cannot resolve any

more references from that library.

Consider the following example:

(1) The input files *file1.o* and *file2.o* each contain a reference to the external function **FCN**.

(2) Input *file1.o* contains a reference to symbol **ABC**.

(3) Input *file2.o* contains a reference to symbol **XYZ**.

(4) Library *liba.a*, member 0, contains a definition of **XYZ**.

(5) Library *libc.a*, member 0, contains a definition of **ABC**.

(6) Both libraries have a member 1 that defines **FCN**.

If the *ld* command were entered as

```
ld  file1.o  -la  file2.o  -lc
```

then the **FCN** reference is satisfied by **liba.a**, member 1, **ABC** is obtained from **libc.a**, member 0, and **XYZ** remains undefined (because the library **liba.a** is searched before **file2.o** is specified).

If the **ld** command were entered as

```
ld  file1.o  file2.o  -la  -lc
```

then the **FCN** reference is satisfied by **liba.a**, member 1, **ABC** is obtained from **libc.a**, member 0, and **XYZ** is obtained from **liba.a**, member 0.

If the **ld** command were entered as

```
ld  file1.o  file2.o  -lc  -la
```

then the **FCN** reference is satisfied by **libc.a**, member 1, **ABC** is obtained from **libc.a**, member 0, and **XYZ** is obtained from **liba.a**, member 0.

METHODS FOR DEBUGGING CODE

CHAPTER FOUR

The Stardent 1500/3000 compilers supply debugging information whether or not the debugger option **-g** is specified during compilation of your program. It is necessary to understand what debugging information is available and at what phase of compilation this information is created, in order to select the appropriate debugging tool for your program. This chapter discusses Stardent 1500/3000 versus standard UNIX compilation systems, why and when you should use a debugger and related debugging support tools that are available.

Stardent 1500/3000 Versus Standard UNIX Compilation Systems

The Stardent 1500/3000 compilation system differs from the AT&T System V and BSD 4.3 UNIX compilation systems, and these differences can affect your choice of debugging tools and the specification or lack of a debug option in your program. Each compilation system begins with compiling a source program and producing a *.o* file. Next, the link editor is invoked and it produces an *a.out* file. A symbol table has also been created by the compilation. Although both compilation systems produce the same files, the contents of these files are quite different.

The Stardent 1500/3000 compilation system, **without** the **-g** option specified, includes all of the information that is only available to the standard UNIX compilation systems when the **-g** option is specified. Using the **-g** option with the Stardent 1500/3000 compilation system has different effects than using it with the standard UNIX system. The **-g** option (on the Stardent 1500/3000) removes all optimization. It does not allow register allocation of variables. In addition, the *a.out* file becomes larger. That is, code segments and labels are added so that the debugger can allow debugging at the source level.

The default Stardent 1500/3000 compilation system (that is, without the **-g** option,) should normally provide you with adequate debugging information. The appropriate time to include

the **-g** option is when you have already isolated the problem in the program and need to single step through the execution of the program.

When to use a Debugger

A debugger is a program revealer, used to figure out what happened to the code. When any behavior in your program is different than what you expect, that is the time you should use a debugger. There are generally four circumstances in which you would want to use a debugger.

1) The program does not work and you need to determine what the error in logic is.

2) The program works but the answers are not what you expected. You must isolate the differences between your expectations and the results.

3) The program works and the answers are correct, but the program runs extremely slowly. This situation is usually examined by using a profiler.

4) The program works as designed but you want a post mortem on what happened during execution.

Debugging Tools

In addition to the Stardent 1500/3000 debugger, *dbg*, there are a number of other tools available to aid you in debugging and analyzing programs. This section provides you with a brief description and discussion of some of these debugging tools, as well as an introduction to the Stardent 1500/3000 debugger.

dbg

dbg is the Stardent 1500/3000 debugger and can best be described as a program revealer. It is a tool that allows you to find out what has happened to your code. The Stardent 1500/3000 debugger has some abilities and features which are different from the standard UNIX symbolic debugger. The most significant of these abilities can be summarized as follows:

- The debugger allows you to ask what either processor is doing.

- The debugger can tell you when a process goes parallel and what the *threads* (that is, the individual parallel processes) are doing.
- The debugger can be used to grab any process, and then look at it, stop it, kill it, or take control of it.

dbg can be used for the same tasks as any other debugger, such as controlling the execution of processes, setting breakpoints, examining the contents of memory and registers, and gaining source and object file information. For more information on the specifics of using *dbg*, refer to chapter 5, *Running dbg—The Stardent 1500/3000 Debugger*.

nm

The *nm* command is a tool that can be used to display the symbol table from your *a.out* file. Some of the types of information that are displayed for each symbol in the table include storage class, type, size in bytes, source line number at which the symbol is defined, and the object file section containing the symbol.

Output for this command may be controlled by a large number of options, which can help you organize and sort external and static symbols. For more information on this command, refer to the *Commands Reference Manual.*

od

The octal dump command, *od*, dumps and displays a file in one or more formats. These formats allow interpretation of certain data types in forms that include hexadecimal, octal, and signed and unsigned decimal.

Dumping a file may be a useful tool in debugging certain portions of your code. Refer to the *Commands Reference Manual* for more information on using *od.*

prof

The *prof* command produces a report on the amount of execution time spent in various portions of your program and the number of times each function is called. This can be a useful tool if your program is producing the desired results but is running very slowly.

To use this command, your Fortran or C program may be linked with the **-p** option, and then when the program is run a file called *mon.out* is produced. *mon.out* and *a.out* then become input to the *prof* command.

A program may be profiled by using the *mkprof* command. This requires no compilation of your program. This produces the same kind of results as running the *prof* command.

For detailed information on using the *prof* command, refer to the *Commands Reference Manual* and chapter 9, *Tuning Code* in this manual.

size

The *size* command is another tool which can be used to gain more information about your program and its subsequent execution. This command produces information on the number of bytes occupied by the three sections (text, data, and bss) of a common object file when the program is brought into main memory to be run. Refer to the *Commands Reference Manual* for more information.

RUNNING THE DEBUGGER

CHAPTER FIVE

This chapter describes *dbg*, the Stardent 1500/3000 symbolic debugger. In addition to standard debugging facilities, such as setting breakpoints, examining variables, single stepping code and so on, it also contains Stardent 1500/3000-specific features such as tracing code through parallel process execution and the ability to gain control of a running process for debugging purposes. In this chapter, some reformatting of the actual output that appears onscreen has been performed to fit the text into the displays. However the technical material presented herein is accurate.

A Simple Initial Session

Here is a short example that illustrates a few of the most often used facilities of a debugger. Following this simple initial session, the chapter goes into much more detail about how commands are formed and options for the commands.

This example shows:

- How to start *dbg*.
- How to set a breakpoint.
- How to start a program running under *dbg*.
- How to display a value of a variable.
- How to continue to the next breakpoint
- How to exit *dbg*.

Here is the source code for this initial simple example. Note that in the sample *dbg* session, what the user typed is shown in bold-faced type. Additionally, all *dbg* commands that the user types are shown in upper case letters. This is done only to provide emphasis that certain typed commands are *dbg* keywords. *dbg* accepts either upper or lower case letters as its command input.

```
/* test.c */
void showit(m)
int m;
{
   printf("\nIncoming value of m is: %d\n", m);
   m = 100;
   printf("Changed local copy of m to: %d\n\n",m);
}

main()
{
   int j;
   j = 30;
   printf("\nPassing variable j (%d) to showit\n", j);
   showit(j);
   printf("Passing a variable in C passes a COPY\n");
   printf("of that variable unless address of the\n");
   printf("is explicitly passed.\n");
   printf("Thus, j = %d before & after 'showit'\n\n", j);
}
```

Here is the command that creates an a.out (that is, object) file that *dbg* can run. Notice that the **-g** option is used to ask the compiler to provide debugging label information in the object code, therefore accessible to *dbg*.

```
prompt> cc -g test.c
```

Finally, here is a brief session with *dbg*, including the startup command. Comments are provided after various command entries have been shown.

```
prompt> dbg a.out
dbg new version (as of  date time).

        Initializing FPU symbol table . . done.
        No corefile
        Reading symbol table from "test.o" . . done.
        Currently debugging "a.out"
```

Set a breakpoint in the subroutine showit.

```
dbg_0> BREAK IN showit

    1,0: Breakpoint set at 'showit:#4:test.o'
        (pc=0x40015c)
```

Run the program to that breakpoint.

```
dbg_1> RUN
Running:
        a.out

Passing variable j (30) to showit
        "a.out" stopped at `showit:#4:test.o`
        (pc=0x40015c)

        printf("\nIncoming value of m is: %d\n", m);
```

dbg performs the command sequence preceding the breakpoint and then reports where it stopped. It shows the next source line that would be executed if a STEP command was to be issued.

A WHERE command (TRACEBACK is an alternate command that does the same thing) asks *dbg* to show a stack traceback indicating the nesting level of subroutine calls and the values of the parameters at the time the subroutines are entered.

```
dbg_2> WHERE
   "a.out" stopped
   `showit:#4:test.o`,    showit(m=30),
   `main:#14:test.o`,     main(),
   `_start:crt0.o`,       pc=0x40012c, sp=0x7fdffe38.
```

Typing the name of a variable is equivalent to typing the command PRINT <variable_name>. The current value assigned to that variable name is printed, interpreted according to the data type that the variable assumes in the current SCOPE of execution.

```
dbg_3> m
    m:  30
```

The LIST command lists the source code (when available). The double forward arrow (>>) indicates which source line is about to be executed when a CONTINUE or STEP command is issued.

```
dbg_4> LIST
4>>       printf("\nIncoming value of m is: %d\n", m);
5         m = 100;
6         printf("Changed local copy of m to: %d\n\n",m);
7       }
8
9       void main()
10      {
11        int j;
12        j = 30;
13        printf("\nPassing variable j (%d) to showit\n", j);
14        showit(j);
15        printf("Passing a variable in C passes a COPY\n");
16        printf("of that variable unless address of the\n");
17        printf("is explicitly passed.\n");
```

```
18        printf("Thus, j = %d before & after 'showit'\n\n",j);
19      }
```

Another break is installed at line #6 in the program so that the program can be continued past a library function. A CONTINUE command free-runs to the next breakpoint, if any. A STEP command executes the next source line, stepping into a procedure call if it finds one. This can take quite a long time and should probably be avoided. A NEXT command executes the next source line and does **not** trace into procedure calls. This form of source-line stepping is probably the most desireable. This brief example uses the CONTINUE command, setting a new breakpoint at a line beyond a procedure call.

```
dbg_5> BREAK AT #6

    2,0: Breakpoint set at 'showit:#6:test.o'
    (pc=0x40018c)
dbg_6> CONTINUE

Incoming value of m is: 30
        "a.out" stopped at 'showit:#6:test.o'
        (pc=0x40018c)

        printf("Changed local copy of m to: %d\n\n",m);
```

Now the user examines the value of the local variable **m**, and continues.

```
dbg_7> m
    m:  100
dbg_8> CONTINUE
Changed local copy of m to: 100

Passing a variable in C passes a COPY of that variable
unless the address of the variable is explicitly passed.
Thus, j = 30 before and after 'showit'

Process 0 (a.out) has exited (exit code = 40)
```

Finally, the user quits *dbg*.

```
dbg_9> QUIT
Done with dbg.
```

Preparing For A Debug Session

To prepare for a debug session, you must

- Prepare your executable program file to be usable by *dbg*. That is, use the **-g** option to compile any object modules for

which variables and subroutines are to be examined by name. Note that using the **-g** flag forces the compiler to use optimization level **-O0** (no optimization).

- Gather your executable files, object files, source files and core file if any, into known places, preferably the current directory. Though it is possible to provide *dbg* with a list of directories to search it is much more convenient if all source and object files are in the current directory when you start *dbg*.

If only the core file is located in the current directory, after starting *dbg* you can use the SOURCE and OBJECT commands to tell *dbg* which directories to search. To establish a new executable (a.out) file and core file, use the DEBUG command. The syntax for each of these commands (SOURCE, OBJECT and DEBUG) is given in the next section.

Starting dbg

The syntax for invoking the Stardent 1500/3000 debugger is

dbg [*a.outfile*] [*corefile*]

a.outfile
: defaults to *a.out* produced by compilation or link editing. This may be changed to reflect your own file name.

corefile
: defaults to *core*, and may be changed to another corefile name, to reflect your the name of your own core file.

If you start *dbg* without specifying the *a.outfile* or the *corefile*, you can use the DEBUG command to establish these file names once *dbg* has been started. Here is an example.

```
DEBUG "a.out.1" "core.1"
```

[1] Optimization sometimes causes rearrangement of code. To allow the debugger to match the generated code sequences with specific line numbers in the source code, no optimization can be used.

Specifying A Search Path For Source and Object Files

NOTE
For the most common usage, the source and object files should all be in the current directory from which you start *dbg*.

Before *dbg* can begin a debug session, it must know where to find the source and object files. If these files are not in the current directory, you can provide a searchpath for *dbg* to use to find them by using the SOURCE and OBJECT commands.

The syntax for these commands is:

```
SOURCE "path [:path][:path...]"
OBJECT "path [:path][:path...]"
```

Note that the double-quotes are a required part of the command syntax. Here are some examples.

```
dbg_nn> source
Current source path is "."

Accessible source files:
     junk.c
     junk2.c

dbg_nn> object ".:/usr/lib"
Current object path is ".:/usr/lib"

Accessible object files:
     junk.o
     junk2.o
```

dbg Startup

When *dbg* starts, it reads header information from *a.outfile* and reads registers, data size and location, and stack size and location from *corefile*, if *corefile* exists. *dbg* also reads your *.dbgrc* file from the current directory (if this file exists) or from your home directory. The file *.dbgrc* contains commands that *dbg* will execute before it begins to accept input from the console. You could use commands in this *.dbgrc* file to establish your debug environment or to establish shortcuts (called *aliases*) for commands if you wish. The alias feature is explained later in this chapter. After executing the commands in *.dbgrc*, *dbg* enters interactive mode and prompts for input with

```
dbg_n>
```

n is the *dbg* command number, starting at zero (0), and incrementing once for each command that you issue. This prompt takes a form similar to the prompt presented by *csh*, the C-shell. A command history (explained later in this chapter), is implemented, allowing you to repeat any

command you had issued earlier by using an abbreviated command sequence referring to that prior command within the command history sequence.

Scope

To understand how commands are entered and interpreted by *dbg*, you must understand the concept of *scope*. All symbols from your program have a scope associated with them. A *scope* is the relative location of the program, which, in turn, establishes the type definitions of variables and the value that is currently assigned to them.

A symbol might be local to a function, local to a line range within a function, or external. For example, there might be a symbol named **temp** in function **foo** and a symbol named **temp** in function **bar**. The debugger must know which of these you want to examine. This symbol takes on a different value depending on which function is currently being debugged and the line number within that function. As an example, **temp** can represent an integer in one function, and a floating point value in another function. When *dbg* knows which section of your program that you wish to examine, it can accurately determine and represent the symbols that are used in the source.

Location Counter

dbg retains **two** notions of a location counter. One location counter represents where the actual execution of a program occurs under control of *dbg*, and the other is the scope location counter used **only** for examining the state of the world, so to speak, in a particular part of a program. This scope location counter simply establishes the context for which a variable name is valid.

NOTE
If a subroutine is not currently active, that is, if it does not appear on the display that results from a TRACEBACK or WHERE[2] command—the stack traceback, then any symbols in that subroutine will not have values and cannot be examined.

When issuing commands to *dbg*, the scope of what you are specifying must be taken into consideration and expressed correctly. You may explicitly change the scope (reset the scope location counter) by issuing a **SCOPE** command. The syntax for specifying any scope is

[2] WHERE and TRACEBACK are synonyms.

```
SCOPE `[ func ]:[ line# ]:[ file ]`
SCOPE `[ line# ]:[ func ]:[ file ]`
```

func is the name of a function.

line # is an integer constant optionally preceded by a #.

file is the name of a source or object file.

If any two of *func, line #*, or *file* are missing from a scope specifier, *dbg* makes the most general assumptions about the missing fields. You must use a scope specifier to qualify an ambiguous name, such as *foo*. If no arguments are present, *dbg* echos back the current scope. Scope specifiers must be delimited by back–quotes.

Here are some examples. Remember that scope resets the scope location counter.

SCOPE `bar:foo.o`
Set the scope to the first instrution of function bar in file foo.o.

SCOPE `bar`
Set the location counter and scope to the first executable instruction of the function bar.

SCOPE `55:foo.c`
Set the location counter and scope to line 55 of source code program foo.c.

SCOPE `bar:#55`
Set the location counter and scope to line number 55 of function bar.

When the SCOPE command is entered with no parameters, the output of this command shows which function is being debugged, the line number within the source file containing that function, and which source file is to be listed when a LIST command is given. Here is a sample of that output.

```
List file is "name.c",
   Current scope is `func:#line:name.o`
```

In addition to the SCOPE command, there are other commands that provide the same output. In particular, *dbg* recognizes the purely informational commands FUNC, LINE, and FILE that are aliases for SCOPE with no parameters.

dbg Commands

There are three types of commands that are accepted by *dbg*:

- keyword commands
- simple expression commands
- compound commands

Each command in *dbg* can then be thought of in this general format:

```
[when] COMMAND [where]
```

An optional specifier that specifies *when* to perform the command (whether ONCE or ALWAYS) may be used (the default is ONCE). A command that is to be done only ONCE is called *non-sticky*. A command that is to be performed ALWAYS is called *sticky*. A *sticky* command is performed every time control passes from the executing code or program to *dbg*.

An optional specifier that specifies *where* to perform the command (whether IN a particular function or AT a particular locator) may also be included. The default is to perform the command once and immediately.

Here are examples. The first is a keyword command. The second command includes a *where* clause. The third command includes a *when* clause and a *where* clause.

```
HELP
ONCE BREAK IN foo
ALWAYS BREAK AT 0x4012348
```

A *dbg* keyword command is a **dbg** keyword followed by zero or more arguments. Arguments to keywords may either appear blank-delimited, or as a parenthesize comma-separated list. Keyword commands include all instructions to *dbg* to perform tasks such as grabbing debugee processes, setting breakpoints, and so on.

In addition to keyword commands, you can formulate other commands to *dbg* that take the form of expressions. An expression can be something as simple as a symbol (in which case the value of the symbol is printed), declaring a type and an optional initial value for a symbol (to establish temporary variables), or a simple expression. *dbg Data Declarations And Operators* (later in this chapter) describes how temporary variables are declared and used. Any variable name is also an expression because the

expression **is** the value of the variable.

Compound expression commands are constructed from simple *dbg* commands or by combining keywords and simple commands.

For each class of commands, a separate section is provided that not only explains the command syntax, but also provides examples of the command's use.

dbg Keyword Commands

The following is a quick reference summary that lists most if not all of the available functions and features of *dbg*. For each section, one or more references are shown to make it easier for you to locate the section in this chapter that describes each facility in detail. Keyword commands may be sorted into the following classes, represented by the major titles on the tables that follow.

dbg Help Commands.

Enter Help facility or get help about specific commands	HELP
List all recognized keywords, both internal and external	KEYWORD
Display this quick reference summary of commands	COMMANDS

Debugee Management.

Debugee Assignment	DEBUG, GRAB
Debugee start/continue	RUN, CONT
Debugee release, kill, run, or restart	RELEASE, KILL, RUN, RERUN

Breakpoints.

Set a break point and specify where and when (permanent/once)	BREAK, STOP IN, AT ALWAYS, ONCE
Temporarily deactivate a breakpoint	SUSPEND
Re-enable a suspended breakpoint, making it unsticky (only executed once)	ACTIVATE
Make an existing command (in the STATUS list) sticky	RESUME
Delete a breakpoint entirely	CANCEL

Program Counter Commands.

Allow the debugee to run freely until next break point	CONT
Advance the debugee one source level instruction	STEP, NEXT
Advance the debugee one assembly level instruction	MSTEP, MNEXT

Exit dbg.

Exit using any of these	QUIT, FINISH, EXIT

Setup/Display dbg Environment.

Define or view directory path to source and object files	SOURCE, OBJECT
Define register naming convention	REGNAMES
Set expression parsing style	C, FORTRAN
Log a *dbg* session or view current logfile name	LOG
Suspend a logging session	UNLOG
Set keyword/name precedence	FAVOR_KEYWORDS, FAVOR_NAMES, FAVOR_MIPS, FAVOR_VRF
Set Window size (num lines)	SET
Set the input and output radix	BINARY, OCTAL, DECIMAL, HEX
View the current radix	RADIX

Modify Debugee Environment.

Define a new program to debug	DEBUG, GRAB
Define signals to be sent directly to debugee	CATCH, UNCATCH

Display Debugee Environment.

Display name of current debugee	DEBUG
Display size of the debugee	SIZE
Display signal catch settings	CATCH, UNCATCH

Display the Program.

Display the program in its source language	LIST, WINDOW
Display the program in assembly language	DIS, MWINDOW

Manage User-Defined Actions.

Show environment and breakpoint settings	STATUS
Show list file and scope	FILE, FUNC, LINE
Call a debugee function to perform an action	CALL

Dynamic Informational Commands.

Display the calling tree	WHERE, TRACEBACK
Display the contents of variables	PRINT, DUMP, $
Display variables in procedures upward in calling tree	SCOPE
Call debugee function to print information	CALL

Signals And Interrupts.

Send a signal to the debugee	SEND
Reset signal catching masks	CATCH, UNCATCH
Modify registers and variables	**Use the '=' assignment operator**

dbg Command Language.

<table>
<tr><td>Known types</td><td>CHAR, UCHAR, LOGICAL, SHORT, USHORT, STRING, INT same as INTEGER, UINT, FLOAT, DOUBLE, REAL, COMPLEX, DCOMPLEX</td></tr>
<tr><td>Declare dbg temporary variables</td><td>DECLARE</td></tr>
<tr><td>Assignment operator</td><td>=</td></tr>
<tr><td>Comparison operators</td><td>>, >=, ==, !=, <=, <</td></tr>
<tr><td>Arithmetic operators</td><td>+, -, *, /, %,
** (Fortran only)</td></tr>
<tr><td>Logical Operators</td><td>&, |, ^, !, ~, &&, ||</td></tr>
<tr><td>Command separator</td><td>;</td></tr>
<tr><td>Compound command start/end delimiters</td><td>{, }</td></tr>
<tr><td>Conditional</td><td>IF</td></tr>
<tr><td>Conditional loop</td><td>FOR, WHILE, DO, ENDDO</td></tr>
</table>

Miscellaneous.

Command history dbg command scripts	HISTORY, !<number> INPUT
Define/undefine Aliases	ALIAS, UNALIAS

Simple Expression Commands

A simple expression command is any Fortran or C expression. Any variables named in the expression are assumed to be within the current scope unless they are qualified with a scope specifier.

When a simple expression command is typed in, the value of that expression is printed on the screen by *dbg*. An assignment expression that you enter causes the assignment to actually take place and *dbg* responds accordingly. A function call in an expression you enter causes *dbg* to call the function. Any expression with a null type does not have its value printed by *dbg*. When a simple expression consists only of an integer constant preceded by an exclamation point, *dbg* interprets that as an alias for the previous command whose command number is equal to the value of the integer.

An assignment expression might look like this:

```
dbg_nn> size = 82
  size: 82
```

size has now been assigned the value **82**.

A function expression might look like this:

```
dbg_nn> 10 + func(y)
```

The function **func** is called with an argument of **y**. If the result of the function is equal to **20**, then *dbg* responds as follows:

```
dbg responds
 10 + func(y): 30
```

this shows the function call the results of the addition.

Compound Expression Commands

You may construct *compound expression* commands from simple *dbg* commands as follows. The syntax of the forms of compound expression commands includes the following terminology:

expression is any simple expression command.

command is either a simple expression command, keyword

command, or another compound command.

The syntax of, and examples of the following forms of compound commands are shown here:

```
DO statement
WHILE statement
FOR statement
IF statement
colon-separated compound statement
brace-enclosed colon-separated compound statement
```

dbg expects that even if a command is a compound expression command, it is entered on a single command line. Therefore if a command extends to more than one physical line, or you may wish to make it multiline), a backslash must appear as the last character on each line, indicating to *dbg* that the line continues to the line that follows. *dbg* acknowledges the continuation by inserting a forward arrow (>) as the first character on the next line. This is the prompt to continue to enter the compound expression. An example of this entry format is shown in the explanation of the DO statement below.

DO ... ENDDO Statement

```
DO left-hand-side = expression, expression [, expression ]
simple command
simple command
   .
   .
   .
ENDDO
```

EXAMPLE:

```
dbg_nn DO i=1, 10 \
>   dummy = foo(i) \
>   bar(dummy) \
> ENDDO
```

WHILE statement

```
WHILE (expression) { command(s) }
```

EXAMPLE:

```
WHILE (dummy) { foo(dummy) }
```

FOR statement

```
FOR (expression; expression; expression) {command(s) }
```

EXAMPLE:

```
FOR ($4=0; $4<100; ($4)+=10) { array[$4] }
```

IF Statement

```
IF(expression) { command(s) } [ELSE { command(s) } ]
```

EXAMPLES:

```
IF (counter < 10) { foo(counter) }
IF (flag) { foo1() } ELSE { foo2() }
```

Colon-Separated Statement

```
simple command; ...; simple command
```

EXAMPLE:

```
$4; foo(); $17
```

Brace-Enclosed Colon-Separated Statement

```
{ simple command; simple command; ...; simple command }
```

EXAMPLE:

```
{ *0x7fdc004c; $4; (string)dummy }
```

Use the IF statement to control the actions associated with a conditional test, that is,

```
dbg_nn>  { IF (loop_count >= 100) { WINDOW ; BREAK } } IN walk_tree
```

Note that the outer set of braces is provided to make this compound statement into a single statement that is qualified with the WHEN clause in walk_tree. Also note that the IF statement can also have an ELSE clause associated with it.

The FOR, WHILE, and DO ... ENDDO statements allow conditional looping. The syntax of FOR and WHILE are like C and the syntax of DO is similar to the format and syntax of Fortran.

Specifying Program Locations For dbg

Some commands to *dbg* require locator values to be specified. A *locator* is a representation of a memory location (in an *a.out*, a *core* file, or a running process) that *dbg* understands. You may specify a *locator* in any of the following ways:

- A virtual locator: an integer expression or a name.

```
STOP AT 0x401ee4
STOP IN main
```

- A line number: an integer expression that cannot be a virtual locator or #*n*, where *n* is an integer constant. The example inserts a breakpoint at line 32 of the current listing file.

```
STOP AT #32
```

- A scope specifier: something that tells *dbg* which context it should consider when attempting to resolve variable names to an explicit locator. (See the explanation of SCOPE.)

```
STOP AT `#27:test1.c`
```

- A breakpoint number: @*n*, where *n* is an integer constant specifying the block number of a breakpoint (see the description of the STATUS command).

```
IF ($4) { foo() } AT @2
```

Typical Commands

This section describes the commands most users would be interested in finding immediately. In other words, these are the commands that a user might *expect* to find in a source-level debugger. Because this is a source-level debugger, the explanations concentrate on displaying variables by referencing source code labels and single-stepping by source code lines (instead of by object code steps). The commands likely to be most frequently used are:

- Running the program
- Starting a post-mortem debug session
- Listing the source code
- Setting, listing, and controlling breakpoints; continuing or single-stepping after a break
- Examining values of variables
- Using debuggee functions within *dbg*
- Getting Help
- Exiting *dbg*.

Following this introduction section, the *dbg* commands are grouped by function type and explained in greater detail in individual sections named for the kind of command in that group.

Starting The Program Running

Use the RUN command to get the program running after setting your break points and actionpoints. If a program has already begun running (or has run to a break point), the RUN command starts the process again from the beginning. However, it asks the user for confirmation.

Deliberately restart the program by using the RERUN command. Here too *dbg* asks for confirmation.

Starting A Post-Mortem Debug Session

There are times when a program has crashed and you wish to use the core file (*core*) to determine why the program crashed. Perhaps the crash happened after running the program for several hours and you do not want to take that time again. As *dbg* starts, it reads the core file and can help you determine the source of the crash.

If you wish to use the source-level debugging capabilities of *dbg*, be sure that appropriate modules have already been compiled with the **-g** option (as noted at the beginning of the chapter). *dbg* can, of course, be used for machine-level instruction debugging if you desire. If you need the source-level capabilities and have not been running code compiled with the **-g** option, you will have no choice but to compile the program and run it again under control of *dbg*.

Move the source, object, and core files in to the same directory (for convenience), and start *dbg*:

dbg *executable core*

Once *dbg* starts, one of its messages should be: *reading corefile from 'core'*. This will make it possible to examine the status of the program at the time it crashed.

Now, when *dbg* starts, give the command WHERE. The example shown here is from the C debugging session shown later in the chapter.

```
prompt> dbg junk core

        Initializing FPU symbol table . . done.
        Currently debugging "junk" using core file "core"
        Process 0 (junk) is not currently parallel
```

```
dbg_nn> WHERE

    Reading symbol table from "test3.o" . . done.
    Reading symbol table from "test2.o" . . done.
    Reading symbol table from "test1.o" . . done.
    Reading symbol table from "junk.o" . . done.
    "junk" received a signal 11 SIGSEGV (seg. violation)
    `strlen:libc.a<strlen.o`, pc=0x403e24, sp=0x7fdffa38,
    `_doprnt:libc.a<doprnt.o`,pc=0x4026a4, sp=0x7fdffa38,
    `printf:libc.a<printf.o`, pc=0x4005e0, sp=0x7fdffca8,
    `test3:#7:test3.o`,
        test3(a=0x100040b0,
               b=0x100040b8,
               c=0x100040c0,
               s=0x7fdffd6c, n=0),
    `test2:#13:test2.o`,
         test2(a=1.2300000190734863e+00,
               b=4.6799998283386230e+00,
               c=1.0350000381469726e+01),
    `test1:#10:test1.o`,  test1(a=1, b=2, c=3),
    `main:#10:junk.o`,    main(),
    `_start:crt0.o`,      pc=0x40012c, sp=0x7fdffe90.
```

The WHERE listing shows the *callback stack,* that is, the complete sequence of program subroutine nesting that was in effect at the time the core file was produced. It also shows the parameters that were passed to each subroutine as it was entered. To examine the values of variables within those subroutines, it is only necessary to set the SCOPE to the appropriate subroutine and ask for the value by typing the parameter name. Note that the source code for this example is within the C debugging example at the end of this chapter.

```
dbg_nn> SCOPE `test3`
dbg_nn> PRINT a
    a:  0x100040b0
dbg_nn> SCOPE `test1`
dbg_nn> PRINT a
    a:  1
dbg_nn> PRINT x
    x:  1.23000002e+00
dbg_nn> b
    b:  2
```

Note that the keyword PRINT is optional. The variable name is sufficient to tell *dbg* to print the value.

Listing The Source Code

Using *dbg*, the program text can be displayed as source or assembly. List the source code of the current listing file by using the LIST or the WINDOW command. The syntax for LIST is:

```
LIST[func | func count | #line | #line count | #line #line]
```

func is the name of the function, which could occur in any one of the source files that was used to create your *a.out* file.

count specifies the total number of lines to list either from the current function specified (*#line count*), or from a new function (*func count*).

#line specifies a starting line number within the current listing file, shown by the STATUS command. The current listing file is automatically established as a result of selecting the function to be listed. If the syntax *#line #line* is used, *dbg* lists that range of line numbers from the current listing file.

The LIST and WINDOW commands display the program text as source. The directory containing the source file must be included in the search path defined by the SOURCE command.

Without any arguments, the LIST command displays a window of lines in the list file starting from the current list location. Another LIST command with no arguments continues displaying source lines from the last location. To set the current list file, use the FILE command.

The WINDOW command displays a window of source lines centered around some source line location. The WINDOW command arguments include a starting location and count. Unlike the LIST command, subsequent WINDOW commands window the same location.

```
dbg_nn> HELP WINDOW
 WINDOW [count | scope | scope count | #line count]
 Display a section of source code around a
    particular location. See MWINDOW for
    assembly level window
```

Note that if the program is active, the '>>' indicates the current location of the PC, that is, where the program is stopped.

Breakpoints and Actionpoints

Setting A Breakpoint

To set a breakpoint, use the BREAK command. In its simplest form, the break command is specified with a location specifier:

```
BREAK in test1

BREAK at 0x40001238
```

BREAK IN *subroutine* stops at the first executable user statement in the named subroutine. BREAK AT *locator* stops at the given locator.

Setting Actionpoints.

An *actionpoint,* as compared to a breakpoint, is a place in the code at which an action takes place. It does not halt the code, but the printout reports that a break has occured and performs the action that you request. To set a actionpoint, specify an action to occur somewhere. For example, to print the value of a variable within a routine, you could use the following:

```
dbg_nn> PRINT `test1`a in test1
```

This command tells *dbg* to print the value of the variable **a** within the subroutine test1. And the action is to be taken just as test1 is entered. If you examine the sample C language debugging session at the end of this chapter, you find that the variable *a* occurs several times, once in each subroutine. In each case, the variable has a different value and is actually specified as a different data type. By explicitly specifying the scope of the variable, this actionpoint can be set because *dbg* will now know which of the available *a*'s you wish to examine and when it should print that value.

At actionpoints, any *dbg* command may be performed, not just a PRINT command. See the section titled *dbg Commands* for more information.

Getting A Summary Of Breakpoints and Actionpoints

After a breakpoint is set, you can get a summary of breakpoints and actionpoints by using the STATUS command. For each breakpoint or actionpoint, a major numerical heading is created, and a minor numerical heading indicates a specific action to be

performed when that breakpoint or actionpoint is hit. The example shown here indicates that if several actions are associated with the same locator, several minor headings are created, one for each action.

You refer to the action you wish to affect by these major and minor numbers, separated by a comma. (Only the parts of the STATUS command output that pertain to this example are shown here). Note that the example describes an action point as a *actionpoint—dbg* is watching for something to happen to trigger some action.

```
    1: Break/watch points set at 'test:#14:xxx.o'
        (pc=0x400210)
(sticky)        0: { print 'test'ggg } at #14
(sticky)        1: break at #14
    2: Break/watch points set at 'test:#17:xxx.o'
        (pc=0x400258)
(sticky)        0: break at #17
```

The first command at the first watch point is referred to as (1,0), the second command is (1,1), and the first command at the second watch point is referred to as (2,0). (Parentheses are optional, but convenient for documentation purposes.)

When a breakpoint is set, it is *sticky*, that is, the break happens any time that position in the code is encountered. To set a break or watch point to be *non-sticky* at the outset, use the keyword ONCE in front of the command.

```
dbg_nn> ONCE BREAK IN test
```

When ONCE is specified, the break only occurs the first time *test1* is entered. For such breaks, the STATUS command does not put "sticky" in front of them.

Suspending Actions At Breakpoints

You can temporarily cause one or more actions to be passed over by using the SUSPEND command. SUSPEND without qualifiers suspends all actions.

```
dbg_nn> SUSPEND
dbg_nn> STATUS

    1: Break/watch points set at 'test:#14:xxx.o'
        (pc=0x400210)
    (inactive) (sticky)       0: print 'test'ggg at #14
(inactive)(sticky)       1: break at #14
```

```
   2: Break/watch points set at `test:#17:xxx.o`
      (pc=0x400258)
(inactive)(sticky)      0: break at #17
```

SUSPEND (*m, n*) suspends only the indicated breakpoint or actionpoint. Parentheses and the comma are optional, that is, the command could have been expressed as SUSPEND *m n*.

```
dbg_nn> SUSPEND 2,0
dbg_nn> STATUS

   1: Break/watch points set at `test:#14:xxx.o`
      (pc=0x400210)
      0: print `test`ggg at #14
          (sticky)      1: break at #14

   2: Break/watch points set at `test:#17:xxx.o`
      (pc=0x400258)
(inactive)(sticky)      0: break at #17
```

SUSPEND (m) suspends all actions at breakpoint *m*. Parentheses are optional.

```
dbg_nn> SUSPEND 1
dbg_nn> STATUS

   1: Break/watch points set at `test:#14:xxx.o`
      (pc=0x400210)
(inactive)(sticky)      0: print `test`ggg at #14
(inactive)(sticky)      1: break at #14

   2: Break/watch points set at `test:#17:xxx.o`
      (pc=0x400258)
(sticky)      0: break at #17
```

Actions taken at breakpoints or actionpoints can be simple, single word (also called Keyword) commands, or compound commands. See the sections titled *dbg Keyword Commands, Simple Expression Commands,* and *Compound Expression Commands* for more information. For example, at a watchpoint, a break can be caused if the value of a variable is greater than a certain value.

```
dbg_nn> IF(`test:#19`ggg > 7) { BREAK } at #15
```

Restoring Actions At Breakpoints

You can re-enable a suspended breakpoint by using the ACTIVATE or the RESUME command. RESUME makes the command *sticky,* as well as activating it. ACTIVATE makes the command *non-sticky.* That is, it is only executed once, the first time the breakpoint or actionpoint is encountered.

```
dbg_nn> SUSPEND 1
dbg_nn> SUSPEND 2
dbg_nn> ACTIVATE 1,0
dbg_nn> RESUME 1,1
dbg_nn> STATUS

   1: Break/watch points set at `test:#14:xxx.o'
      (pc=0x400210)
          0: print `test`ggg at #14
   (sticky)      1: break at #14

   2: Break/watch points set at `test:#17:xxx.o'
      (pc=0x400258)
(inactive)(sticky)      0: break at #17
```

Cancelling A Breakpoint

Instead of simply suspending a breakpoint or an associated action, you can cancel a breakpoint, its actions or all breakpoints at once by using the CANCEL command.

CANCEL 1,1
: remove action: breakpoint 1, action 1

CANCEL 1
: remove breakpoint 1 entirely

CANCEL
: remove all breakpoints entirely.

Continuing After A Break

Continue after a break by using the CONT command (or use its abbreviation, a period (.). The code will proceed until it hits another breakpoint or exits. Single step to the next line of source code by using the STEP (or NEXT) command. Note that a single line of source code could very easily be many many machine code instruction steps. Move several steps in source code lines by specifying STEP with an integer step count. Here is an example.

```
dbg_nn> STEP 5
```

This command causes *dbg* to execute 5 lines of source code, then to halt awaiting further instructions.

The STEP and NEXT commands advance the location counter at the source-line level. The STEP command causes the debugee to execute one source-level statement. If that statement is a procedure, then *dbg* breaks within that procedure. That is the STEP

command steps into procedures. The NEXT command also causes the debugee to execute one source-level statement, however, procedure calls are treated as one source-level statement. Control will return to *dbg* command mode when the debugee has completed the procedure call.

Viewing or Modifying Program Variables

View or modify program variables by specifying them by name just as in the original source code. You can use the optional keyword PRINT as part of the command to view the contents of a named variable, but it is not necessary. Depending on the current SCOPE setting, it may be necessary to explicitly specify a scope for a variable to view its value or modify its contents.

dbg reports an error if it cannot find a variable by the name you give in the current SCOPE. *dbg* also reports an error if you try to modify the value of a variable that is not yet active. In other words, only subroutines that are already active can have variables whose memory locations have been defined and from which a value can be determined. If the name of the subroutine does not appear in the listing that the TRACEBACK or WHERE command produces, none of its variables can be printed or assigned a value. Local variables only come into existence when the subroutine in which they are declared have been run, thus assigning a storage for each individual invocation, to the symbol.

```
dbg_nn> PRINT myvar
  myvar is not currently active
dbg_nn> myvar2
  Can't find 'junk:testx'myvar2
  Current scope is 'junk:testx'
dbg_nn> 'testa'myvar3
  'testa'myvar3: 12
```

Using Debuggee Functions Within dbg

Using the CALL command, you may call a function within the debugee, to modify its internal state.

```
dbg_nn> CALL fixup_list()
```

Or, you might wish to test the functioning of a subroutine by calling the function with a known value and checking the result.

```
dbg_nn> x = test(3.14); x
```

Remember that arguments to the debuggee procedure are interpreted in the current radix.

Getting Help

dbg has an extensive online help facility that you can enter either by typing the command keyword HELP or by using its abbreviation, a question mark (?). The prompt changes from *dbg_nn>* (where *nn* is a command sequence number in the command history) to *dbg_HELP>*. Most help pages have three types of information on them:

- help text
- *dbg* command names
- numbered paragraphs

As you read the help text, you may display a short description of a specific command or keyword by typing its name. If necessary, you may redisplay the current menu by typing '0'.

Help on commands and keywords is also available in normal *dbg* command mode using the HELP command, that is:

```
dbg_nn> HELP command_or_keyword
```

You can also list all keywords that *dbg* understands by entering the KEYWORD command. Some of the words that are listed by this command are not explained in this chapter either because they are internal commands or parameters, or because they have been added to the *dbg* parse table (internal data item interpretation table) but are not currently implemented. You can use the HELP <keyword> command to examine any of them for yourself.

Exiting dbg

You can exit *dbg* by any one of three different keywords: EXIT, FINISH, or QUIT. They all have the same effect.

dbg Command Language

The *dbg* command language is very versatile. Temporary variables can be declared. Conditional statements can test the values of temporary or debugee program variables. Control structures can be used to create sophisticated statements.

All keywords and Fortran names are recognized on a case insensitive basis. Keywords are also recognized on an unambiguous prefix basis, which means that *dbg* attempts to understand what keyword you are looking for with as few letters as necessary.

If a word that you type into *dbg* is both a *dbg* keyword as well as a symbol name in your program, *dbg* by default resolves the conflict in favor of the keyword. To change this default, and to tell *dbg* to select symbol names first, use the FAVOR_NAMES command. You can restore the default value by later entering FAVOR_KEYWORDS.

Notice in the example below that only the minimum number of characters has been entered to resolve the difference between the two commands, demonstrating that *dbg* accepts unambiguous prefixes in place of full commands.

```
dbg_nn> FAVOR_N
Resolving name/keyword conflicts
  in favor of names
dbg_nn> FAVOR_K
Resolving name/keyword conflicts
  in favor of keywords
```

In addition to keywords, *dbg* also can set a default method for resolving the naming of the floating point registers. In the Stardent 1500/3000 P3 processor, there are 32 floating point registers associated with the MIPS floating point processor, and 32 scalar registers associated with the Vector Processing Unit. Each of these registers can be referenced as $f[n], $d[n] or $i[n] where *n* is a value from 0 to 31. The keywords FAVOR_MIPS and FAVOR_VRF establish the method used to resolve whether a reference is to the MIPS floating point processor's registers or to those of the VRF (the Vector Register File). The default is FAVOR_MIPS.

Blank input lines are ignored, and a line may be continued by using a backslash as its last character. There is no limit for line continuations. If there are many multi-line commands, it may be useful to execute the commands as a script, using the INPUT command.

dbg Data Declarations And Operators

Declaring a type and a value for a variable for *dbg* causes that name to be declared and recognized. A temporary variable defined in this manner can be assigned a type and an initial value. *dbg* understands the following types: CHAR, UCHAR, SHORT,

USHORT, INT, UINT, STRING (equivalent to char*), FLOAT, and DOUBLE. For Fortran, additional types are: INTEGER, LOGICAL and REAL. An example of a variable declaration is given below.

```
dbg_nn> INT limit = 200
dbg_nn> limit = 200
```

Note that simply declaring a variable without declaring its type is not acceptable to *dbg*.

```
dbg_nn> a = 1
a = 1

^
dbg error:
    'a' is an ambiguous dbg keyword
    Current scope is <extern>
```

Applying Type Casts To Variables To Match Types

If variables of different types are to be used in an expression, occasionally it is necessary to explicitly tell *dbg* that a variable's value is to be handled as though it is of a different type, since *dbg* is only allowed to perform arithmetic or logical operations on variables of compatible types. The types for which type casting is accepted are the same as the types specified at the start of this section (CHAR, UCHAR, SHORT and so on).

Operators Understood By dbg

dbg understands the following operators and can use them to display or to interact between declared variables and your program variables. Note that if you must mix variables of different types within an expression, you must *cast* the variables so that only like types are operated upon.

Type of Operation	Operator	Example	Comments
assignment	=	nnn = 100	Establish a value for a variable
comparison	>	a > b	Test for greater than
comparison	>=	a >= b	Test for greater or equal
comparison	==	a == b	Test for equality
comparison	!=	a != b	Test for inequality
comparison	<=	a <= b	Test for less or equal
comparison	<	a < b	Test for less than
arithmetic	+	a + b	Add two items together

<table>
<tr><td>arithmetic</td><td>–</td><td>a – b</td><td>Subtract</td></tr>
<tr><td>arithmetic</td><td>*</td><td>a * b</td><td>Multiply</td></tr>
<tr><td>arithmetic</td><td>/</td><td>a / b</td><td>Divide</td></tr>
<tr><td>arithmetic</td><td>%</td><td>a % b</td><td>Remainder after division (integer)</td></tr>
<tr><td>arithmetic</td><td>>></td><td>a >> b</td><td>Bit shift right b positions</td></tr>
<tr><td>arithmetic</td><td><<</td><td>a << b</td><td>Bit shift left b positions</td></tr>
<tr><td>arithmetic</td><td>++</td><td>a++</td><td>Increment by one</td></tr>
<tr><td>arithmetic</td><td>—</td><td>a—</td><td>Decrement by one</td></tr>
<tr><td>logical</td><td>&</td><td>a & b</td><td>Bitwise logical AND</td></tr>
<tr><td>logical</td><td>|</td><td>a | b</td><td>Bitwise logical OR</td></tr>
<tr><td>logical</td><td>^</td><td>a ^ b</td><td>Bitwise logical XOR</td></tr>
<tr><td>logical</td><td>&&</td><td>a && b</td><td>Logical AND of logic conditions</td></tr>
<tr><td>logical</td><td>||</td><td>a || b</td><td>Logical OR of logic conditions</td></tr>
<tr><td>logical</td><td>!</td><td>!a</td><td>Logical negation of a logic condition</td></tr>
<tr><td>C only</td><td>-></td><td>pointer->next</td><td>Dereference from a pointer</td></tr>
<tr><td>Fortran only</td><td>**</td><td>a ** b</td><td>Exponentiation</td></tr>
</table>

When forming logical or arithmetic expressions, parentheses may be freely used to establish an order of evaluation different than that of the default.

By using these operators with either your internal program variables or with externally defined global variables, you can easily write test conditions to control your *dbg* session. For example, perhaps you wish to see the value of some variable only on every 50th pass through a particular loop. You might declare an external variable **g** to be your external loop counter. Assume that the function *test1* is called from that loop, so you could set an action point within test1 to increment the external variable **g** and then to print the value of some other variable, lets say **h**, each time 50 loops have been counted.

```
dgb_nn> INT g
dbg_nn> g = 1
   g: 1
dbg_nn> ALWAYS g++ IN test1
dbg_nn> { IF(!(g % 50 ))  { PRINT h } } IN test1
```

The syntax for the IF statement is shown earlier in this chapter, in the section titled *Compound Expression Commands*. Here are some more examples.

```
dbg_nn> FLOAT m = 1.347000E+03
dbg_nn> FLOAT n = 2.100000E+03
dbg_nn> m
   m:  1.347000E+03
dbg_nn> n
   n:  2.100000E+03
dbg_nn> m + n
     m + n:  3.44700000e+03
```

```
dbg_nn> m / n
    m / n:  6.41428590e-01
dbg_nn> INT g
dbg_nn> g = m
    g: 1347
dbg_nn> g / 100
    g / 100: 13
dbg_nn> g % 100
    g % 100: 47
```

For C, if a program declares a pointer variable, and it points to an item of a specific data type, you can examine the items within the data structure to which the pointer points by using C's dereferencing method: **pointer->item**

```
dbg_nn> pointer->item
  pointer->item: 0x7fdffe08
dbg_nn> pointer->item->next->a[7]
  pointer->item->next->a[7]:  5
```

Radix

The radix is the numerical base that *dbg* uses when you enter or display a number that does not have a base value explicitly declared as part of the number itself. For example, if you have a hex value to enter, you could enter it as **0x1234** or specify that the radix is hexadecimal (HEX) and just enter **1234**. Explicitly specifying the base for the entry is accomplished by using a prefix, such as:

```
0b - for binary
0o - for octal
0d - for decimal
0x - for hexadecimal
```

The radix is queried by the RADIX keyword. The radix is set by using any of the following: HEX, DECIMAL, OCTAL, BINARY, for base 16, 10, 8 and 2 respectively. Here are examples.

```
dbg_nn> RADIX
Current radix is decimal
dbg_nn> INT x
dbg_nn> x = 255
     x: 255
dbg_nn> BINARY
dbg_nn> x
     x:  0b00000000000000000000000011111111
dbg_nn> OCTAL
     x:  377
dbg_nn> HEX
     x:  ff
dbg_nn> x = 33
```

```
     x: 33
dbg_nn> DECIMAL
dbg_nn> x
     x: 51
dbg_nn> x = 0x11
     x: 17
```

The Status Command

The STATUS command gives various information about the status of *dbg* itself, as well as about the process being debugged.

The STATUS command accepts no parameters. Here is a sample output of this command. In this example, *dbg* has been entered from a directory that has an *a.out* file compiled from files *junk.c* and *test1.c*, *test2.c*, and *test3.c*. There is no core file present.

```
Interpreting names and expressions as in C
Resolving name/keyword conflicts in favor of keywords
Referring ${f|d:i}n to MIPS (or VRF) floating registers
Process 0 (a.out) is not currently parallel
List file is "xxx.c", Current scope is 'main:#4:xxx.o'
There are no break or watch points currently set
```

The list file is that file that would be listed if a LIST command was given. The scope is set to line 4 of the file containing the entry point 'main'.

If there were any breakpoints or actionpoints set in the program, these break points would also be listed in the STATUS command output.

Controlling Execution Of Debuggee Processes

This section describes how to specify the program or process to be debugged. It also demonstrates that you can grab control of an already running process. *dbg* can debug a program or a running process. The object which is being debugged is known as the "debugee".

If you are debugging a program, use the commands DEBUG, RUN, RERUN, and KILL to control it. You identify the program to be debugged on the shell command line when invoking *dbg* or by using the DEBUG command from within *dbg*. The RUN and RERUN commands, kills the current debugee if it is running, and places the debugee in its initial state and passes it execution control.

If you are debugging a running process, use the commands: GRAB, RELEASE, and any of the commands that advance the program counter, such as CONT or STEP. The GRAB command takes a process ID argument and suspends the process. Note that the process ID is the Unix process ID. This can be obtained by using the *ps* command.

Note that *dbg* must be able to find the executable program in order to grab a process (see the OBJECT command for more information.) When you are finished debugging the process, use the RELEASE command to allow it to continue normal execution and to be freed from *dbg* debugging control. You may use the RUN, RERUN, or KILL commands, however they cause the grabbed process to be killed.

Signals And Interrupts

When the debugee reaches a break point and control is passed to *dbg* command mode, one can execute *dbg* commands to modify the execution environment or the execution of the debugee.

Control may have passed to *dbg* as a result of an attempt to deliver a signal to the debugee. To deliver this or any signal to the debugee, use the SEND command. By default, *dbg* does not deliver this signal to the debugee.

The current signal disposition is displayed in response to the CATCH and UNCATCH commands with no arguments. The CATCH command with a signal number argument, will cause that signal, when it occurs, to be delivered to *dbg* rather than the debugee. To cause a signal to be sent directly to the debugee, use the UNCATCH command.

Miscellaneous Commands

This section contains miscellaneous commands.

Display The Debuggee Environment

The DEBUG command, with no arguments, lists the name of the current debugee.

The SIZE command lists the code, data, and text size of the current debugee.

Command History

View previously executed commands or execute commands that you have entered before just by specifying their position

in the command history listing.

- To view the past set of commands you have entered, type HISTORY

```
dbg_nn> HISTORY
        dbg_nn> DEBUG "junk"
        dbg_nn> SCOPE
        dbg_nn> LIST
        dbg_nn> FUNC
        dbg_nn> LINE
        dbg_nn> HISTORY
```

- To re-execute a command you entered previously without retyping it entirely, enter !<*n*>, where <*n*> is the integer value that appears as the sequence number of the command you wish to execute. Example:

```
dbg_nn> !3
       func
       <output of 'func'>
```

Note that this is the only form of "bang"(!) command syntax that works for the current version of *dbg*. A full C-shell command syntax, (with parameter substitution, for example) has not been implemented.

Sometimes it may become necessary to repeat certain actions, such as setting breakpoints, looking at values of variables, setting values of parameters or dummy temporary variables, single stepping and so on for each time you recompile a program and attempt to find the source of a problem. Thus you might wish to create a command script while performing a debug session and from that script, use the script input capability of *dbg* on a future run to step forward to a particular point in your debug session and continue on from there.

Logging Your dbg Session

You may create a record of a *dbg* session and save what you type as well as the response that the computer provides. For this, you use the LOG command. The syntax for the commands described here is:

```
LOG
LOG "filename"
UNLOG
```

- If **no** file is being used to log your session, *dbg* reports **no log file specified.**

- If a file name for logging has already been specified, *dbg* reports the name of the file **and** re-enables logging of the subsequent commands and output to that file (appends to the open logging file).

WARNING
If a file by this name already exists, its current contents are destroyed and a new file with this name is created.

- If you specify a file name with the LOG command, that file name is used to record subsequent commands. Then *dbg* reports that it is indeed logging commands to this file name.
- If you want to temporarily suspend logging to a file, use UNLOG. The message indicates that logging to a specific file has been temporarily suspended.

Running Commands From A Script

Start *dbg* normally, then give the command:

```
INPUT "myscript"
```

All of those commands that you had written into your script are executed in sequence, just as though you had typed them at the terminal. When the INPUT command hits end–of–file on the external command script, it begins to accept commands from the terminal. Thus your repetitive setup work has been completed.

Creating Command Scripts

When you use the LOG command, everything that you type and everything that the computer types to the terminal is saved to the file. An alternate command, RECORD, saves only what you type. Thus by using the RECORD command, you can create a script that can later be run to debug up to a particular point.

Examples of often-used dbg commands

Setting and deleting break/watch points

Stop on entry to foo:

```
STOP AT foo
BREAK AT foo
STOP IN foo
```

Print $4 at 0x401ffc:

```
$4 AT 0x401ffc
```

Cancel all break/watch points:

```
CANCEL
```

Cancel all break/watch points associated with block 4 (see STATUS)

```
CANCEL 4
```

Suspend the break/watch point at block 3, number 2 (see STATUS)

```
SUSPEND 3 2
```

Resume all suspended break/watch points:

```
RESUME
```

Printing values of variables and looking at memory

Print the value of str.foo[2]:

```
str.foo[2]
```

Print (in hex) the contents of 20 words of memory surrounding address 0x10003224:

```
WINDOW 0x10003224 20
```

Disassemble memory starting at the location counter (number of lines set by WINDOW size):

```
DIS $pc
```

Dump (in hex and ascii) the contents of 16 (the default) words of memory starting at the location of environ:

```
DUMP environ
```

Setting dbg parameters (hex, FORTRAN, etc.)

Print all integral values in hex from now on:

```
HEX
```

(other legal radixes are DECIMAL, OCTAL, and BINARY)

Interpret all subsequent expressions and commands as if the current source were written in Fortran:

```
FORTRAN
```

(you may do the same for C)

Looking at the Floating Point Unit

Print double precision scalar register 5:

```
$d5
```

Print single precision vector register at bank 3, block 4, cell 5:

```
$fv3.4.5
```

Print integer accumulator 3:

```
$ia3
```

Print the value of the vector length control register:

```
$length
```

Print control registers associated with the A pipe:

```
$a
```

In general $<name> gets you the name of the FPU control register named <name> in the fpu_state structure in */usr/include/machine/fpu.h*.

Abbreviating dbg Commands

The following are *dbg* abbreviations:

!!	Repeat the previous command
!*n*	Repeat command *n* by reparsing it (n is an integer constant, corresponding to the command number reflected in the command history file being maintained by dbg)
&*n*	Repeat command *n* by re-executing the internal parse tree that has already been created for it
.	NEXT
,	STEP
..	CONTINUE
?	HELP

A Sample Debugging Session In C

Here is a sample debugging session in C. The formatting of the output from the debugger is not necessarily identical to what you see onscreen; it has been modified to fit in the available space. The program is called *junk.c,* and it seems to have a bug near the end and that bug causes a core dump. (In actual fact, it has **two** bugs, which we see near the end of the session.) Here is the source listing:

```
/* junk.c */
void test1();

void main()
{
    int x, y, z;

    x = 1; y = 2; z = 3;
    test1(x, y, z);
}

/* test1.c */
void test2();

void test1(a, b, c)
int a, b, c;
{
    float x, y, z;

    x = 1.23 * (float)a;
    y = 2.34 * (float)b;
    z = 3.45 * (float)c;
    test2(x, y, z);
}
```

```
/* test2.c */
void test3();

void test2(a, b, c)
float a, b, c;
{
    char s[100];
    char *x, *y, *z;

    x = "hello,";
    y = "world";
    z = "\n";
    strcpy(s, "this goes into the string\n");

    test3(x, y, z, s, 1000);

    /* There's only 100 places in the array, we
     * are going to index to position 1000 to
     * guarantee to kill the process and dump core
     */
}

/* test3.c */
void test3( a, b, c, s, n)
char *a, *b, *c, *s;
int n;
{
     printf("%s %s %s",a,b,c);
     printf("%c",s[n]);
}
```

The program is broken into several segments to show that *dbg* can read several source files for a single debugging session if necessary. You also notice that the same variable names, **a**, **b**, **c**, **x**, **y**, and **z**, are used in each of the segments to demonstrate that *dbg* knows what type of variable each name represents when the *scope* of the program is set to a particular file or line number within a file. In other words, *dbg* knows that **x** is an integer in the function **main**, a floating point value in **test1**, and a pointer to a character string in **test2**.

When this program runs, it prints:

```
hello, world
Segmentation fault (core dumped)
```

Here is the *Makefile* that produced this object code.

```
junk: junk.o test1.o test2.o test3.o
        cc -g junk.o test1.o test2.o test3.o -o junk

.c.o:
        cc -O -c -g $*.c
```

Start *dbg*. The default file to find is *a.out*, but the *Makefile* that produced the program specifies the name of the executable file as *junk*.

```
rap@pubs <1> dbg junk
dbg new version (as of  DATE TIME).

Initializing FPU symbol table . . done.
Reading symbol table from "test3.o" . . done.
Currently debugging "junk" using core file "core"
Process 0 (junk) is not currently parallel
```

Note that the status as *dbg* begins shows Process 0, which means that the program, though loaded into *dbg* and ready to run, is not now running.

```
Interpreting names and expressions as in C
Resolving name/keyword conflicts in favor of keywords
Process 0 (junk) is not currently parallel
List file is "test3.c", Current scope is 'test3:#6-7:test3.o'
There are no break or watch points currently set
```

The STATUS command output shows that the current scope is 'test3:#6-7:test3.o'. This was read from the *core* file and indicates where the location counter was at the time the core file was generated.

The list file is shown as *test3.c* and it shows where in the source code the program was executing when the program crash occured.

```
6>>           printf("%s %s %s",a,b,c);
7             printf("%s",s[n]);
8       }
```

Notice the >> in the above listing, at line 6. It indicates that the line following the double arrow is that line which was to be executed but in the process of trying to execute, the crash occurred. The indication "**#6-7**" in the scope listing indicates that the crash happened between the execution of lines 6 and 7 in that source file. Just to demonstrate why it says **between** lines 6 and 7, look at a disassembly of this section of source code:

```
dbg_2> DIS
(#6)     0x4003d8:       0x3c0f1000      lui     $t7,0x1000
         0x4003dc:       0x25ef40e0      addiu   $t7,$t7,16608
         0x4003e0:       0x1e02021       move    $a0,$t7
         0x4003e4:       0x8fa50060      lw      $a1,96($sp)
         0x4003e8:       0x8fa60064      lw      $a2,100($sp)
         0x4003ec:       0x8fa70068      lw      $a3,104($sp)
(#6)     0x4003f0:       0xc100158       jal     printf
         0x4003f4:       0x0             nop
         0x4003f8:       0x3c0f1000      lui     $t7,0x1000
         0x4003fc:       0x25ef40ec      addiu   $t7,$t7,16620
         0x400400:       0x24180001      li      $t8,1
         0x400404:       0x8fae0070      lw      $t6,112($sp)
         0x400408:       0x0             nop
         0x40040c:       0x1d80018       mult    $t6,$t8
         0x400410:       0x6812          mflo    $t5
         0x400414:       0x0             nop
         0x400418:       0x0             nop
         0x40041c:       0x8fac006c      lw      $t4,108($sp)
         0x400420:       0x0             nop
         0x400424:       0x18d5821       addu    $t3,$t4,$t5
         0x400428:       0x816d0000      lb      $t5,0($t3)
         0x40042c:       0x0             nop
         0x400430:       0x1a05821       move    $t3,$t5
         0x400434:       0x1e02021       move    $a0,$t7
         0x400438:       0x1602821       move    $a1,$t3
(#7)     0x40043c:       0xc100158       jal     printf
```

(Only that part of the disassembly that relates to the current question is shown). Note from this disassembly that the **printf** function is the last function in statement 7 to be executed. It would appear that there is something wrong with the parameters passed to **printf** because there does not seem to be anything above it that could have caused a segment violation. Confirm that by running the program under control of the debugger, and set a breakpoint just above the final call to **printf**.

```
dbg_3> BREAK AT 0x400428

    1,0: Breakpoint set at `test3:#6-7:test3.o'
       (pc=0x400428)
dbg_4>
```

And now run the program.

```
dbg_4> RUN
Running:
        junk

hello, world

"junk" stopped at `test3:#6-7:test3.o'
   (pc=0x400428)
```

```
0x400428:       0x816d0000      lb      $t5,0($t3)
```

The program stops at the breakpoint and displays the next instruction that it is about to execute. Because the breakpoint is set as a locator (rather than as a line number in a source code file), the next instruction is a disassembled machine code instruction instead of showing the next source line to be executed. Take a machine code step, with MSTEP.

```
dbg_5> MSTEP
Reading symbol table from "test2.o" . . done.
Reading symbol table from "test1.o" . . done.
Reading symbol table from "junk.o" . . done.
"junk" received a signal 11 SIGSEGV (segmentation violation)
   `test3:#6-7:test3.o',
      test3(a=0x100040b0, b=0x100040b8, c=0x100040c0,
            s=0x7fdffd4c, n=1000),
   `test2:#13:test2.o',
      test2(a_$$1=1.2300000190734863e+00,
            b_$$2=4.6799998283386230e+00,
            c_$$3=1.0350000381469726e+01),
   `test1:#10:test1.o',
      test1(a=1, b=2, c=3),
   `main:#10:junk.o',      main(),
   `_start:crt0.o',        pc=0x40012c, sp=0x7fdffe70.
dbg_3> $t3
  $t3:  2145386804
```

The crash is a segment violation. The instruction that caused it was loading a byte from location 2145386804, which is outside the range of the program data storage area.

Now recall the programmer's comment in file *test2.c*, that a deliberate crash is to be expected to happen:

```
    /* There's only 100 places in the array, we
     * are going to index to position 10000 to
     * guarantee to kill the process and dump core
     */
```

Go back and recompile the program with this item corrected, changing the line that calls **test3** as follows:

```
test3(x, y, z, s, 0);
```

If you run the program again, it still crashes. At first glance, you might think you are now pointing to the first element in the **s** array. Look now at what happened to the program.

```
dbg_0> LOG "logfile"
Logging session in "logfile"
dbg_1> DIS `test3.c`
(`test3:#2:test3.o`):
    test3:
(#2)     0x4003c0:        0x27bdffc0      addiu    $sp,$sp,-64
         0x4003c4:        0xafbf003c      sw       $ra,60($sp)
         0x4003c8:        0xafa40060      sw       $a0,96($sp)
         0x4003cc:        0xafa50064      sw       $a1,100($sp)
         0x4003d0:        0xafa60068      sw       $a2,104($sp)
         0x4003d4:        0xafa7006c      sw       $a3,108($sp)
(#6)     0x4003d8:        0x3c0f1000      lui      $t7,0x1000
         0x4003dc:        0x25ef40e0      addiu    $t7,$t7,16608
         0x4003e0:        0x1e02021       move     $a0,$t7
         0x4003e4:        0x8fa50060      lw       $a1,96($sp)
         0x4003e8:        0x8fa60064      lw       $a2,100($sp)
         0x4003ec:        0x8fa70068      lw       $a3,104($sp)
(#6)     0x4003f0:        0xc100158       jal      printf
         0x4003f4:        0x0             nop
         0x4003f8:        0x3c0f1000      lui      $t7,0x1000
         0x4003fc:        0x25ef40ec      addiu    $t7,$t7,16620
         0x400400:        0x24180001      li       $t8,1
         0x400404:        0x8fae0070      lw       $t6,112($sp)
         0x400408:        0x0             nop
         0x40040c:        0x1d80018       mult     $t6,$t8
         0x400410:        0x6812          mflo     $t5
         0x400414:        0x0             nop
         0x400418:        0x0             nop
         0x40041c:        0x8fac006c      lw       $t4,108($sp)
         0x400420:        0x0             nop
         0x400424:        0x18d5821       addu     $t3,$t4,$t5
         0x400428:        0x816d0000      lb       $t5,0($t3)
         0x40042c:        0x0             nop
         0x400430:        0x1a05821       move     $t3,$t5
         0x400434:        0x1e02021       move     $a0,$t7
         0x400438:        0x1602821       move     $a1,$t3
(#7)     0x40043c:        0xc100158       jal      printf

... <listing abbreviated> ...

dbg_2> BREAK AT 0x400424

    1,0: Breakpoint set at `test3:#6-7:test3.o`
        (pc=0x400424)
dbg_3> RUN
Running:
         junk

         "junk" stopped at `test3:#6-7:test3.o`
             (pc=0x400424)
         0x400424:        0x18d5821       addu     $t3,$t4,$t5

dbg_4> MSTEP
         "junk" stopped at `test3:#6-7:test3.o`
             (pc=0x400428)
         0x400428:        0x816d0000      lb       $t5,0($t3)
```

```
dbg_5> MSTEP
        "junk" stopped at 'test3:#6-7:test3.o'
           (pc=0x40042c)
        0x40042c:        0x0              nop

dbg_6> MSTEP
        "junk" stopped at 'test3:#6-7:test3.o'
           (pc=0x400430)
        0x400430:        0x1a05821        move     $t3,$t5

dbg_7> MSTEP
        "junk" stopped at 'test3:#6-7:test3.o'
           (pc=0x400434)
        0x400434:        0x1e02021        move     $a0,$t7

dbg_8> MSTEP
        "junk" stopped at 'test3:#6-7:test3.o'
           (pc=0x400438)
        0x400438:        0x1602821        move     $a1,$t3

dbg_9> MSTEP
        "junk" stopped at 'test3:#7:test3.o'
           (pc=0x40043c)
(#7)    0x40043c:        0xc100158        jal      printf
        0x400440:        0x0              nop

dbg_10> $a0
    $a0:  268452076
dbg_11> HEX
dbg_12> $a0
    $a0:  0x100040ec
dbg_13> (STRING)$a0
    (string)$a0:  0x100040ec  "%s"
dbg_14> $a1
    $a1:  116
dbg_15> CONT
Reading symbol table from "test2.o" . . done.
Reading symbol table from "test1.o" . . done.
Reading symbol table from "junk.o" . . done.
"junk" received a signal 11 SIGSEGV (segmentation violation)
 'strlen:libc.a<strlen.o',     pc=0x403e24, sp=0x7fdffa18,
 '_doprnt:libc.a<doprnt.o',    pc=0x4026a4, sp=0x7fdffa18,
 'printf:libc.a<printf.o',     pc=0x4005e0, sp=0x7fdffc88,
 'test3:#7:test3.o',
```

It did not crash in the same place as last time (loading of register $t3). This time it crashed during the **printf** itself. The function **printf** requires one or more *locators* of data. From the contents of **$a0**, it appears that the compiler has positioned the user static data area somewhere near memory location 0x10000000, and notice that **$a1** contains a value of 116, so it does not seem to point to where it should. The **printf** statement in *test3.c* now becomes suspect:

```
printf("%s", s[n]);
             ^^^^
             Here's the problem...
             this construct fetches
             the CHARACTER at that
             position, not its ADDRESS
             as expected by printf.
```

So, fix this to read:

```
printf("%s", &s[n]);
```

Recompile the program and it runs fine to completion.

Now, with or without the recompile, you can use the file *junk* to demonstrate that *dbg* knows about scope. That is, if a label is used local to a function, *dbg* recognizes the type of variable to which that label refers within the function and prints it properly. The next sequence of code shows scope.

```
        Logging session in "logfile"
dbg_2> BREAK IN test1

    1,0: Breakpoint set at 'test1:#7:test1.o'
        (pc=0x4001c4)
dbg_3> BREAK IN test2

    2,0: Breakpoint set at 'test2:#8:test2.o'
        (pc=0x40033c)
dbg_4> BREAK IN test3

    3,0: Breakpoint set at 'test3:#6:test3.o'
        (pc=0x4003d8)
dbg_5> LIST test1
2        void test1(a, b, c)
3        int a, b, c;
4        {
5            float x, y, z;
6
7            x = 1.23 * (float)a;
8            y = 2.34 * (float)b;
9            z = 3.45 * (float)c;
10           test2(x, y, z);
11       }
12
dbg_6> RUN
Running:
        junk

        "junk" stopped at 'test1:#7:test1.o'
            (pc=0x4001c4)

    x = 1.23 * (float)a;
dbg_7> STEP
```

```
        "junk" stopped at `test1:#7:test1.o`
            (pc=0x4001e4)

    x = 1.23 * (float)a;
dbg_8> STEP
        "junk" stopped at `test1:#8:test1.o`
            (pc=0x400224)

    y = 2.34 * (float)b;
dbg_9> STEP
        "junk" stopped at `test1:#9:test1.o`
            (pc=0x400264)

    z = 3.45 * (float)c;
dbg_10> PRINT x
    x:  1.23000002e+00
dbg_11> PRINT y
    y:  4.67999983e+00
```

Note in the file *test1.c,* **x** and **y** are floating point values and *dbg* recognizes the labels as such and prints them as floating point values. The variables **a**, **b**, and **c** are integers here. To continue with the example:

```
dbg_12> PRINT a
    a: 1
dbg_13> LIST `test2`
2       void test2(a, b, c)
3       float a, b, c;
4       {
5           char s[100];
6           char *x, *y, *z;
7
8           x = "hello,";
9>>         y = "world";
10          z = "\n";
11          strcpy(s, "this goes into the string\n");
12
13          test3(x, y, z, s, 0);
14      }
dbg_14> CONT
        "junk" stopped at `test2:#8:test2.o`
            (pc=0x40033c)

    x = "hello,";
dbg_15> STEP
        "junk" stopped at `test2:#9:test2.o`
            (pc=0x400348)

    y = "world";
dbg_16> STEP
        "junk" stopped at `test2:#10:test2.o`
            (pc=0x400354)

    z = "\n";
```

```
dbg_17> STEP
        "junk" stopped at `test2:#11:test2.o'
           (pc=0x400374)

    strcpy(s, "this goes into the string\n");
dbg_18> STEP
        "junk" stopped at `test2:#13:test2.o'
           (pc=0x400398)

    test3(x, y, z, s, 0);
```

The variables **a, b, c** are floating point values here, and **x, y, z** now refer to string variables.

```
dbg_19> PRINT x
    x:  0x100040b0
dbg_20> (STRING)x
    (STRING)x:  0x100040b0  "hello,"

dbg_21> PRINT y
    y:  0x100040b8
dbg_22> PRINT z
    z:  0x100040c0
dbg_23> PRINT a
    a:  1.23000002e+00
dbg_24> PRINT b
    b:  4.67999983e+00
dbg_25> PRINT c
    c:  0.00000000e-01
dbg_26> LIST `test3'
2       void test3( a, b, c, s, n)
3       char *a, *b, *c, *s;
4       int n;
5       {
6            printf("%s %s %s",a,b,c);
7            printf("%s",&s[n]);
8       }
13>>
14
dbg_27> CONT
        "junk" stopped at `test3:#6:test3.o'
           (pc=0x4003d8)

     printf("%s %s %s",a,b,c);
```

And here **a, b, c** are string variables.

```
dbg_28> PRINT a
    a:  0x100040b0
dbg_29> PRINT b
    b:  0x100040b8
dbg_30> QUIT
        Process 786 (junk) killed
```

In this example, you have seen many of the *dbg* facilities used in tracing the flow of the program and in determining the location of a problem. You have also seen that it is possible to use *dbg* as a post mortem examiner of a core file following a program crash to determine the values of variables and the location of the problem in the program. There are many functions that we have not demonstrated in the example. We suggest that you use *dbg* on your own projects and become familiar with by direct use. Remember that there is a help facility available. If you develop some familiarity with *dbg* and have learned many of the commands that you use in normal situations, you can get a quick reference to the commands by using help to get the command list.

```
dbg_32> HELP COMMANDS

   [ lists every command that dbg understands,
     sorted into categories related to the types
     of things you wish to do ]
```

Machine Code Related Commands

The DIS command displays the program as machine instructions. If source line numbering information is available, assembly instructions which mark the start of a source-line instruction are so marked. Like the other display commands, the DIS command takes two optional arguments, a starting location and a count.

```
dbg_42> HELP DIS
DIS address [n]: disassmble the first 'n' words
    starting at 'address' (see WINDOW)

DIS [n]: disassemble the first 'n' words starting
   at the last address disassembled via DIS,
   WINDOW, or DUMP
```

The MWINDOW command displays a window of machine instructions centered around a program location. Like the other display commands, the MWINDOW command takes two optional arguments, a starting location and a count.

```
dbg_43> HELP MWINDOW
MWINDOW [expr n] | [n] : print 2*'n' (default for
   'n' is WINDOW) words of memory starting at
   address 'expr'-'n' (default 'expr' is the
   last address previously printed by a MWINDOW,
   or DIS command or the current pc)
```

Exploring On Your Own

dbg is an evolving program. It is possible that this chapter may not cover all of the commands and keywords that *dbg* can handle. To explore the facilities of *dbg* on your own, begin by typing the command KEYWORDS at any *dbg* prompt. This will list all of the keywords that *dbg* understands. You can then type HELP, followed by any of the listed keywords to find out what that command does. Additionally, there is an online reference for *dbg* that is accessible by typing the HELP command (or simply a question mark) at the *dbg* prompt. Between the two forms of help built into the program, it may be possible for you to find information about any commands that might not be covered here.

VECTOR & PARALLEL OPTIMIZATION

CHAPTER SIX

Stardent 1500/3000 is a vector, multiprocessor computer. Stardent 1500/3000 programs which use vector instructions or which can execute in parallel threads on several processor units can achieve substantial speed increases. The Stardent 1500/3000 Fortran and C compilers aggressively optimize programs, searching for these speed increases. In this chapter, some of the ideas behind these optimizations are explained; these explanations should help you understand the compiler and its effect on your programs. Most of the examples are in Fortran, but the ideas apply as well to C.

Fundamental Concepts

A *scalar* is a single data value in a program. In Fortran, a constant, a simple variable, or a subscripted array element is a scalar. Most computers perform operations combining a few (one to three, usually) scalar values to produce a new scalar value. For example, in the loop

```
DO I = 1, N
     A(I) = A(I) + B(I)
END DO
```

each floating point addition, each load of **A(I)**, each load of **B(I)**, and each store of the sum into **A(I)** is a *scalar operation*. Indeed, the computations needed to control the loop (loading, incrementing, and testing **I**) and to subscript the variables are also scalar operations. A reasonable compiler for a typical computer might be able to generate code that would execute about eight instructions per iteration, with two instructions to set up and one to finish; to add n data elements, the compiler would take about $8n+3$ scalar operations. A *vector unit*, on the other hand, treats a whole sequence of scalar values as one data unit, a *vector*, and can perform a single operation on all the vector's elements at once. A vector is often thought of as a row of values contiguous in memory (elements 1 through n of the one-dimensional Fortran array **A**, for example), but on the Stardent 1500/3000, a vector can

be any evenly spaced sequence in memory, or irregularly spaced with scatter/gather techniques. The spacing between elements is called the *stride*. The previous loop above can be written (in an informal vector notation) as

```
A = A + B
```

where we assume that arrays **A** and **B** are each dimensioned 1 to N. A typical vector unit might execute this single statement with four or five scalar operations to setup the vector unit and then three or four *vector operations*. When a statement is vectorized, the loop control instructions no longer exist in that they have been replaced by instructions that tell the vector unit what to do and on how many elements to perform the operation (as well as where to store the results). The vector hardware can do these operations much more quickly, than these same operations could have been done in scalar mode. A vector operation can also overlap parts of the computation, loading one value while adding two others and storing yet another. This overlapping makes much more efficient use of hardware components than scalar operations, where each operation completes before the next is allowed to begin. For these reasons, loops that can be *vectorized* might run as much as ten times faster than their scalar counterparts.

Similarly, multiple processor units may allow a program to execute along multiple *threads*, each thread assigned to a different processing unit. This scheme works best on and is easiest to apply to program loops with two characteristics:

- There is a great deal of computation within the scope of the loop (this guarantees low average cost for setting up the multiprocessing, that is, the overhead is amortized).

- Each iteration of the loop is independent of the other iterations of the loop; it neither generates data for nor uses data from other iterations (this means there is no need for interprocessor communication).

If both properties hold for a loop, then the individual iterations can be executed in *parallel*, each on its own processor. With proper operating system support, the individual iterations need not execute the same instructions nor take the same length of time. Also, the number of processors available need not be same as the number of loop iterations (typically, there are far fewer processors than iterations). In theory, n processors could run a program in parallel in $1/n$th the time one processor would take; in practice, this goal is never reached because of the overhead

necessary to manage the threads and to keep track of the processors. Still, a multiprocessor like the Stardent 1500/3000, running on scientific programs that are well suited to parallelism, might well see speed increases of 70% to 95% of the theoretical maximum.

The Stardent 1500/3000 compilers exploit the Stardent 1500/3000's vector units and multiple processors by transforming source programs written in terms of scalar operations into ones in which vector and parallel operations appear (the jargon is that the compilers *vectorize* and *parallelize* the programs or, in brief, *optimize* them). But these transformations must be *safe*. What is a safe transformation? Consider running the original scalar program and the optimized program on any possible input values; if the results are always the same, then the transformations were safe. The notion of safety is usually limited to programs which used appropriate data transformations in the first place; if the scalar program could fail for some input, the optimized program may also fail and may do so in a different way or on different inputs.

How can a compiler (or a programmer, for that matter) tell if a transformation is safe? The notion of *data dependence* provides the key. Consider these statements:

```
10        X = 2.0*Y
20        Z = X*W + 1.0
30        P = X + A(3)
40        A(4) = P + Z
```

Each assignment depends on the values of variables used in the expressions on the right hand sides. This means that statements **20** and **30** cannot be executed until statement **10** is complete and statement **40** must wait for both statements **20** and **30**. However, statements **20** and **30** do not depend on each other in any way; either could be executed first without changing the effect of the program. We say that a *data dependence* exists between two statements if, during some execution of the program, the first statement and the second statement both access the same memory location in such a way that the order of the two statements must be preserved. Most of the time this occurs because the first statement *produces* a value that the second statement *uses*. In general, a transformation is safe if it preserves all data dependencies in the program; in the example, exchanging statements **20** and **30** would be safe, but exchanging statements **30** and **40** would not.

Now consider vector operations. The vector units operate on all the elements of a vector effectively **simultaneously**. Look again at

the first example:

```
DO I = 1, N
       A(I)  = A(I)  + B(I)
END DO
```

When converted to vector form, it is as if the **A** array were lined up over the **B** vector, the adds performed all at once, and the results stored together.

$$\begin{array}{cccc} A_1 & A_2 & \cdots & A_N \\ + & + & \cdots & + \\ B_1 & B_2 & \cdots & B_N \\ | & | & \cdots & | \\ A_1 & A_2 & \cdots & A_N \end{array}$$

This is acceptable because no result depends on any other; each add and store combination is independent of the others.

Here is a slightly modified loop.

```
DO I = 1, N
     A(I)  = A(I-1)  + B(I)
END DO
```

Now there is a data dependency hidden in the loop; the assignment statement **depends on itself**. If the loop is eliminated by writing the assignments from each iteration in order (a process called *loop unrolling*), the dependency stands out.

```
A(1)  = A(0)  + B(1)
A(2)  = A(1)  + B(2)
A(3)  = A(2)  + B(3)
         .
         .
A(N)  = A(N-1)  + B(N)
```

In fact, the value of each element of **A** is given by the equation:

$$A(I) = A(0) + \sum_{J=1}^{I} B(J)$$

This dependency of the assignment on itself is known as a *recurrence*. The recurrence stops the compiler from using the vector hardware because there is no way to assign the values simultaneously, as required by the *semantics* of vector operations. (Semantics describes the meaning of something). The vector operation equation demands that the operations on all elements of the vector be operated upon at the same time. The vector unit is able to overlay certain operations, however there is still an element of sequential operation to the vector unit calculation.

Just as some loops can be executed on the vector unit and others cannot, some loops can be executed by multiple processors in parallel and others cannot. To use multiple processors effectively, a system must balance the work load across the processors. Individual iterations of a loop normally take about the same amount of processing time. So performing different iterations of the same loop on different processors should keep the processors working evenly. Processing a loop in parallel can often speed up the loop by a factor slightly less than the number of processors utilized.

The compiler does not assign iterations to processors because processor availability is not known until runtime; rather, it marks code for parallel execution. Each processor, as it becomes available during the program's run, requests an iteration to perform. Suppose that the first processor receives the first iteration and begins working. The second processor becomes available, receives the second iteration, and begins working. Meanwhile, the first processor finishes the first iteration and requests another; it receives the third iteration and begins working. Iterations are parcelled out to processors as they become available until all iterations have been assigned and are completed.

Under this scheduling scheme, the loop iterations may be performed in random order. Suppose that the second processor finishes the second iteration before the first processor finishes. Recall the first loop to sum two arrays.

```
DO I = 1, N
        A(I) = A(I) + B(I)
END DO
```

Each iteration of the loop works on different elements of the arrays, independent of all the other elements. Parallel processing is possible (although the loop is so short that it may not be efficient). Introducing a recurrence, as in the following loop, keeps the loop from being processed in parallel.

```
DO I = 1, N
        A(I) = A(I+1) + B(I)
END DO
```

Now the first iteration uses the existing value of **A(2)**. Suppose that the second processor computes the second iteration before the first processor fetches the existing value of **A(2)**. Then the first processor computes the first iteration with the wrong value and produces the wrong result. This loop forces the processors to *synchronize* their accesses to data, that is, to wait for one another, thus canceling the benefit of parallel processing. Loops are good

candidates for parallel processing when they have no data dependencies and when they contain as much computation as possible to maximize the distance between synchronization points. The Stardent 1500/3000 compiler finds loops that produce the same results when processed in parallel as when processed on a single processor. This process is called the *parallelization* or *concurrentization* of a program.

The Stardent 1500/3000 compilers automatically vectorize and parallelize programs at compile-time. In addition to detecting and scheduling operations for vector processing, for parallel processing or for vector and parallel processing, the compiler optimizes loops in other ways regardless of whether they vectorize or parallelize.

Compiler Techniques

A Stardent 1500/3000 compiler optimizes a program by performing a complete dependency analysis on it. Once the analysis is done, the compiler transforms the program by replacing scalar operations by vector and parallel operations, always observing the constraints defined by the dependencies. Auxiliary (but still safe) transformations may sometimes be applied to rearrange or eliminate dependencies, thus allowing still more transformations. Also, the full battery of optimizations usually applied to scalar operations are applied during vector and parallel optimization; these commonly uncover yet more transformations.

The Stardent 1500/3000 vector units offer a variety of vector operations; all of the obvious ones are included. In addition, there are many operations that combine scalars with vectors, transform scalars into vectors, or transform vectors into scalars.

Here is an example of that combines a scalar with a vector:

```
A = B + 1.0
```

where **1.0** is added to each element of **B** and the sum stored in the corresponding element of **A**.

Here is an example that transforms a scalar into a vector:

```
A = X
```

where each element of **A** is given the scalar value **x**. And here is an example that transforms a vector into a scalar:

```
X = SUM(A)
```

That is, sum the elements of **A**. If a statement in a loop can be rewritten so that all the operations are vector operations and the statement no longer needs to be surrounded by the loop, we say that the statement is *vectorized*. If only some or none of the operations can be transformed, and the loop is still necessary, the statement **did not** vectorize (although the use of vector operations within it may still create a significant speed increase). In the rest of this chapter, some important vector transformations are described.

When a vreport is generated, the notation that it uses is similar to the format of a Fortran 8X program. Note that the compiler itself does not accept Fortran 8X, but accepts only Fortran 77 statements. Fortran 8X statements are not accepted as source input to the compiler. The compiler transforms Fortran 77 constructs into more efficient vector operations wherever possible and expresses the modifications that it made in the program in a Fortran 8X form.

Program Transformations

The Stardent 1500/3000 compiler is extremely sophisticated, and performs a number of program transformations to uncover and enhance the parallelism within a program. This section details many of those transformations, and gives examples of the resulting code in **vreport** format. With these examples, you should be able to recognize the changes in your program effected by the vectorizer.

Induction Variable Elimination

The *loop induction variable* is the variable named in the loop control statement that steps through the iterations of the loop. The variable **I** in the loop below is the induction variable. A common programming practice, particularly in older Fortran programs, is to use an *auxiliary induction variable* to control some part of the loops execution. For instance, an auxiliary variable may be used to step through an array with strides of two, or to start at an offset deeper in an array, or (as in the following example with variable **IX**) to step through an array backwards.

```
IX = N
DO I = 1, N
     A(IX) = B(I) + C(I)
     IX = IX - 1
END DO
```

The loop adds arrays **B** and **C** and assigns the result to **A** but in **reverse** order. In this form, the loop cannot be directly vectorized, because **IX** hides the variance of array **A** on the loop. If you compile the above program fragment to obtain the vreport, you see the following:

```
DO iv=1, N, 32
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
       A(1 - I + IX) = B(I) + C(I)
   END DO
END DO
```

The Stardent 1500/3000 vectorizer has replaced the variable **IX** within the loop with an expression that varies directly with the loop variable, and as a result, the loop gets vectorized. The Stardent 1500/3000 compiler is able to recognize and replace a broad range of auxiliary induction variables, so normally you need not worry about this phase. If, however, you have an auxiliary induction variable which the Stardent 1500/3000 compiler does not eliminate (due to EQUIVALENCE or COMMON, perhaps), then you want to eliminate it by hand since no use of an auxiliary induction variable (that is not eliminated by induction variable substitution) will vectorize.

Constant Propagation

One of the key hindrances to automatic code vectorization is a lack of information for the compiler regarding the values of variables. When a variable whose value is only known at runtime is used in a subscript of an array, the Stardent 1500/3000 compiler must assume the worst case with regard to dependencies, which often means that code does not vectorize. To alleviate this problem, the Stardent 1500/3000 compiler performs global constant propagation. When the compiler can prove that a particular use of a variable always has the same constant value, it replaces that use with the constant value. For example, consider a slight variation of the previous example:

```
N = 100
IX = N
DO I = 1, N
     A(IX) = B(I) + C(I)
     IX = IX - 1
END DO
```

The **-vreport** output for this fragment is as follows:

```
DO iv=1, 100, 32
   rv = MIN(100, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      A(101 - I) = B(I) + C(I)
   END DO
END DO
```

The Stardent 1500/3000 Fortran compiler has eliminated the auxiliary induction variable, and has recognized that the use of N is constant both in the bound of the loop and in the replacement expression generated for **IX**. In general, the constant propagation algorithm employed by the Stardent 1500/3000 compiler is very effective, so that if you see variables that you believe are constant valued that the compiler has not recognized, there is probably a control path or equivalence that you have missed.

Dead Code Elimination

Dead code is a program segment that calculates a result that is never used during the program execution. Users rarely create dead code, but many compiler transformations can. For instance, in the previous example, the assignment to *N* is dead once all uses of *N* have been replaced by 100. The Stardent 1500/3000 compiler eliminates such code automatically.

Similarly, some parts of a program may be unreachable at run-time. Again, users rarely create such code, but transformations such as procedure inlining can. The Stardent 1500/3000 compiler eliminates unreachable code to save space in the object file (since the code is never executed, eliminating it obviously does not affect the execution time). When this elimination occurs in user code, a warning message is issued by the compiler since this condition usually reflects a user oversight. When the elimination occurs in inlined code, the compiler silently removes it.

Loop Distribution

Sometimes the compiler can vectorize parts of a loop but not all of the loop. Consider the following loop.

```
      DO 80 I = 1, N
         A(I) = A(I-1) + A(I)
         B(I) = C(I) + D(I)
80    CONTINUE
```

NOTE
A **recurrence** is a cycle in a dependence graph; it occurs when a statement **depends** upon itself, reusing its own results.

The assignment to **B** can be vectorized, but the assignment to **A** can not be vectorized because it contains a recurrence. The Stardent 1500/3000 compiler partially vectorizes this loop by *distributing* the loop around the two statements.

```
      DO 80 I = 1, N
         A(I) = A(I-1) + A(I)
80    CONTINUE
      DO 85 I = 1, N
         B(I) = C(I) + D(I)
85    CONTINUE
```

It then vectorizes the second loop, producing the following vreport:

```
DO iv=1, N, 32
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      B(I) = C(I) + D(I)
   END DO
END DO
DO I=1, N
   A(I) = A(I - 1) + A(I)
END DO
```

Note that in the process of vectorizing the loop, the Stardent 1500/3000 vectorizer changed the order in which the statements are executed.

Loop Interchange

The Stardent 1500/3000 Fortran compiler, unlike many other vectorizing compilers, considers any loop in a *nest* to be a viable candidate for vectorization. The transformation which enables this is loop interchange. The Stardent 1500/3000 compiler is able to determine when the order in which loops are executed can be safely and profitably changed, and may often execute loops in a different order than that specified by the programmer. Matrix multiplication is a common example that illustrates the value of

this transformation:

```
DO I = 1, N
    DO J = 1, N
        C(J,I) = 0.0
        DO K = 1, N
            C(J,I) = C(J,I) + A(J,K)*B(K,I)
        END DO
    END DO
END DO
```

Any of the loops can be vectorized in this example, but the best loop to vectorize around both statements is the J loop. The K loop is a dot product reduction, which is slightly less efficient than a regular vector operation, and the I loop accesses many arrays with non-unit stride. The Stardent 1500/3000 compiler recognizes the J loop as being the best vector loop, producing the following vreport:

```
b2 = MAX(N, 0)
b3 = MAX(N, 0)
DO I=1, N
   DO iv=1, N, 32
      rv = MIN(N, 31 + iv)
      vl = rv - iv + 1
      DO VECTOR J=iv, rv
         C(J, I) = 0.0D0
      END DO
      DO K=1, N
         DO VECTOR J=iv, rv
            C(J, I) = C(J, I) + A(J, K) *
         END DO
      END DO
   END DO
END DO
```

Note that the DO VECTOR J loop has been interchanged inside the K loop. This vectorization generates the fastest possible code for this formulation of matrix multiplication.

In general, the ability to interchange loops permits the Stardent 1500/3000 Fortran compiler to generate extremely efficient vector and parallel code. However, this ability may occassionally cause the compiler to generate worse code, particularly for programs that have been tuned for compilers that can only vectorize inner-most loops. For instance, in the following code fragment:

```
DO I = 1, M
   DO J = 1, N
       A(I,J) = A(I,J) * A(I,J)
   ENDDO
ENDDO
```

the Stardent 1500/3000 compiler evidently vectorizes the outer loop as

```
b2 = MAX(N, 0)
DO iv=1, M, 32
   rv = MIN(M, 31 + iv)
   vl = rv - iv + 1
   DO J=1, N
      DO VECTOR I=iv, rv
         A(I, J) = A(I, J) * A(I, J)
      END DO
   END DO
END DO
```

since it does not know the loop lengths, and the outer loop provides unit stride access to memory. If at run time *M* happens to be very small (say 3) and *N* happens to be very large (say 1000), then this vectorization runs much slower than vectorizing the inner loop. This situation tends to happen most often on codes that have been tuned for compilers that only vectorize inner loops (since programmers tend to ignore outer loops on those compilers.) A simple VBEST directive inserted on the inner loop lets the Stardent 1500/3000 compiler know that it is the best vector loop, causing it to generate the best vector code.

Scalar Expansion

The compiler cannot vectorize loops that reuse computations. Scalar temporaries provide one way for a loop to reuse its computations and, therefore to prevent the compiler from generating vector or parallel instructions. Suppose that the previous matrix multiplication had been coded to take optimal advantage of a scalar machine by using a scalar temporary to accumulate the results.

```
      DO 100 I=1, N
          DO 101 J=1, N
              T = 0.0
              DO 102 K=1, N
                  T = T + A(J,K)*B(K,I)
102           CONTINUE
                  C(J,I) = T
101       CONTINUE
100   CONTINUE
```

This loop cannot be directly executed in vector as it stands, because the scalar temporary blocks correct use of the vector unit. The Stardent 1500/3000 compiler recognizes this blockage, and overcomes it by expanding the scalar temporary into a vector temporary:

```
b2 = MAX(N, 0)
b3 = MAX(N, 0)
DO I=1, N
   DO iv=1, N, 32
      rv = MIN(N, 31 + iv)
      vl = rv - iv + 1
      DO VECTOR J=iv, rv
         tv_t(2 + J - iv) = 0.0D0
      END DO
      DO K=1, N
        DO VECTOR J=iv, rv
          tv_t(2 + J - iv) = tv_t(2 + J,K) + A(J,K) * B(K,I)
        END DO
      END DO
      DO VECTOR J=iv, rv
         C(J, I) = tv_t(2 + J - iv)
      END DO
   END DO
END DO
```

When the Stardent 1500/3000 compiler expands a scalar, it creates a new scalar vector name by adding the prefix "tv_" to the scalar name (for example, "tv_t" for the variable T above). As a rough rule, the Stardent 1500/3000 compiler is able to expand any scalar where there is a clear definition before any use in the loop (as above) or where all uses come before the first definition. If uses and definitions are mixed haphazardly, then you make it difficult for the Stardent 1500/3000 compiler to determine how to create a correct expansion. The Stardent 1500/3000 compiler does not expand scalars that appear in EQUIVALENCE statements.

NOTE
Release 3 and prior versions of the compiler do not expand EQUIVALENCEd scalars.

Reduction Recognition

Many important computations require a loop to use results that it has previously computed. The most common of these operations reduces a vector into a single element as in the following loop.

```
      DO 110 I=1, N
         T = T + A(I)
110   CONTINUE
```

The operation's use of the result of the previous iteration prohibits execution on the vector unit. Because of this operation's importance, Stardent 1500/3000 contains special hardware, called

reduction hardware, to speed up its computation. This is represented in the vreport by a reduction operation:

```
DO iv=1, N, 32
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      T = T + SUM_REDUCTION(A(I))
   END DO
END DO
```

The Stardent 1500/3000 compiler recognizes the following reduction recognitions for which there is special purpose hardware:

- SUM_REDUCTION — taking the sum of all the elements of a vector.
- PROD_REDUCTION — taking the product of all the elements of a vector.
- DOT_PRODUCT — taking the inner or dot product of two vectors.
- MAX — finding the maximum element of a vector.
- MIN — finding the minimum element of a vector.
- ANY_REDUCTION — taking the OR reduction of a Boolean vector.
- ALL_REDUCTION — taking the AND reduction of a Boolean vector.
- PARITY_REDUCTION — taking the exclusive OR reduction of a Boolean vector.
- COUNT_REDUCTION — counting up the nonzero entries in a vector.

Additionally, the Stardent 1500/3000 compiler also recognize special index reductions for which there are vectorized subroutine calls. Consider the following program segment:

```
DO I = 1, N
    IF (ABS(A(I)) .GT. MAX) THEN
        INDEX = I
        MAX = ABS(A(I))
    ENDIF
ENDDO
```

If you request a vreport for this code segment, it would show that the compiler has converted the bulk of the work into a vectorized library call:

```
r1 = N
cs6 =  IDAFMAX(b1, A, 1)
IF (ABS(A(1 + cs6)) .GT. MAX .AND. b1.NE. 0) INDEX = 1 + cs6
IF (ABS(A(1 + cs6)) .GT. MAX .AND. b1.NE. 0) MAX = ABS(A(1 + cs6))
```

The routine *IDAFMAX* (this name is an artifact of vreport conversions; the actual internal name is _idafmax) returns the index of the first occurrence in the vector A of the element which has the greatest absolute value. The rest of the code is cleanup code that resets the scalar values MAX and INDEX only when the indexed value is greater than the original value of MAX.

The following special index routines are in the runtime library, and patterns similar to that above are recognized by the Stardent 1500/3000 vectorizer and converted to calls to these functions:

```
_idafmax  _idfmax  _iiafmax  _iifmax  _isafmax  _isfmax
_idafmin  _idfmin  _iiafmin  _iifmin  _isafmin  _isfmin
_idalmax  _idlmax  _iialmax  _iilmax  _isalmax  _islmax
_idalmin  _idlmin  _iialmin  _iilmin  _isalmin  _islmin
```

These routines have been named using a convention similar to that in the Linpack group of subroutines:

- The first letter is always *i*, indicating that the routines return integer values.
- The second letter indicates the type of array searched—"d" for a double precision array, *s* for a single precision array, and *i* for an integer array.
- If the third letter is *a*, the routine takes the absolute value of all the array elements before searching.
- If the next letter is *f*, the routine returns the index of the first element that satisfies the request (this arises from a greater than operation in the Fortran source code for maximum). If the letter is *l*, the routine returns the index of the last element that satisfies this request (the Fortran source code had a greater than equal for maximum).
- If the last three letters are *max*, the routine hunts for a maximum. If they are *min*, it hunts for a minimum.

Thus, the routine *_idfmax* searches a double precision array for the index of the element with the maximum value, and returns the index of the first such element it finds. The routine *_islmin* searches a single precision array for the index of the element with the minimum value, and returns the index of the last such element it finds.

EFFICIENT PROGRAMMING TECHNIQUES

CHAPTER SEVEN

The Stardent 1500/3000 compilers automatically optimize programs to run on Stardent 1500/3000's vector and parallel hardware. Stardent 1500/3000's compilers do the following things well:

- They examine all candidates in a nest of loops for vectorization and parallelization opportunities.
- They select the best loops for vectorization and parallelization based on the information available at compile-time.
- They run sophisticated dependence tests to accurately determine the data usage order in arrays.
- They remove induction variables where possible to aid vectorization.
- They accurately analyze equivalenced variables to permit vectorization.

As a result, most of the time you need do nothing more than write standard code to have your programs run in a quick and efficient manner on Stardent 1500/3000.

While the Stardent 1500/3000 compiler is very sophisticated in optimizing your programs, it is not omniscient, and does not generate perfect code for every program. In most of the cases where it does not generate nearly optimal code, the reason is usually that some knowledge about the program structure that it needs to safely optimize the program is not available. In such cases, a small amount of effort on your part in restructuring code or in inserting directives can produce dramatic improvements in program performance. You should also be aware that this chapter describes the Stardent 1500/3000 vectorizer in the 2.2/3.0 release; in future releases many of the restrictions described will be relaxed.

The first section of this chapter explains programming techniques that take the best advantage of the automatic optimizations of

Stardent 1500/3000's Fortran and C compilers. Most examples in this chapter are illustrated using the Fortran language; however, the same principles apply to C. Differences between Fortran and C are noted.

Write Vectorizable Loops

Stardent 1500/3000's compilers focus most of their effort on looping constructs—the Fortran **DO** loop in particular. The more computation you code in structured loops, the more efficiently your programs will run.

Write Convertable Alternate Loops

The Stardent 1500/3000 vectorizer only attempts to vectorize Fortran **DO** loops, so if you have any other looping construct (Fortran **WHILE** loops or C **for**, **while**, and **repeat** loops)in your program, either you or the compiler must convert it to a Fortran **DO** loop before it will be vectorized. For the majority of common cases, the Stardent 1500/3000 compiler will perform this conversion automatically, thereby allowing you to somewhat freely use alternative looping constructs. If the Stardent 1500/3000 compiler does not convert a loop, you will need to modify the loop so that the compiler will recognize it as a **DO** loop.

In Fortran, **WHILE** loops that have been converted to **DO** loops will appear as **DO** loops in the vector report. In C, the situation is slightly more subtle. Inside the compiler, **for** loops are actually represented as **while** loops. As a result, the compiler uses **for** notation in the C vector report to list loops that have been converted to **DO** loops; it uses **while** notation for loops that have not. For instance, if you feed the following fragment into the Stardent 1500/3000 C compiler at **-O2 -vreport**

```
main()
{
    int  i;
    double a[100];

    for(i=0; i<100; i++)
         a[i] = (double) i;
}
```

you will see the following in the vreport file

```
for ($$iv = 0; $$iv != 99; $$iv += 32) {
    $$rv = MIN(99, 31 + $$iv);
    $$vl = $$rv - $$iv + 1;
    DO VECTOR ($$I1 = $$iv; $$I1 != $$rv; $$I1++) {
       a[$$I1] = double($$I1);
    }
}
```

The compiler has successfully converted the **for** loop in the source into a Fortran **DO** loop, and then vectorized it. The **for** loop in the vector report is evidence of that; had the compiler not converted the source loop into a **DO** loop, the vector report would contain a **while** loop instead. To see this, consider the following similar example which the compiler cannot convert to a **DO** loop.

```
main()
{
   int  i,*n;
   double a[100];

   for(i=0; i<*n; i++)
        a[i] = (double) i;
}
```

The vector report for this fragment contains a **while** loop, not a **for** loop.

```
while (i < *n)
{
   $B1 = $$B1 + 1;
   a[i] = double(i);
   i = i + 1;
}
```

The compiler is unable to convert this **while** loop into a **DO** loop, because it does not know that the pointer *n* does not overlap some location in *a*. If it does, then the upper bound of the loop changes within the loop; the semantics of C require that the change be reflected in the controlling condition of the loop, whereas the semantics of the Fortran **DO** loop require that it does not. Any time you see a **while** loop in a vector report, the compiler has been unable to convert it into a **DO** loop. You must get that to occur before any vectorization will occur, no matter how many directives you insert.

If you have a **while** or **for** loop that the vectorizer does not convert to a **DO** loop, you should examine it to see if it violates one of the following conditions:

(1) The loop must be simple enough that the Stardent 1500/3000 compiler can easily recognize the induction variable, the step, and the upper bound of the loop. The compiler examines the condition controlling the **while** loop; if it is not one of the C operators <, <=, >, >=, or != and it is not a single variable, the compiler immediately gives up the conversion process. The single variable case gets treated as " variable != 0 ", thereby converting it into one of the operator cases. One side of the operator must be a variable (the loop induction variable); the other can be a general expression (the upper bound for the loop). Finally, the compiler must be able to find one (and only one) statement in the loop that increments the loop induction variable; it must be able to determine that that statement is executed once on every iteration of the loop. The easiest way to guarantee that the compiler will recognize that the increment comes on every loop iteration is to ensure that the increment is either the first or last statement in the loop. If any of these conditions are not met, the compiler will not be able to recognize the various parts of the Fortran DO loop. To give a few examples:

```
while (*i < n)
   *i++;
```

Presently, the compiler will not recognize the use of a dereferenced pointer as being the loop induction variable. As a result, it will not convert this loop.

```
while(i > n) {
   if (i > 1) found = 1;
   i--;
   if (i < m) done = 1;
}
```

The compiler will not convert this loop, because it cannot determine that the decrement occurs only once within the loop.

```
while(i > n) {
   if (i > 1) found = 1;
   if (i < m) done = 1;
   i--;
}
```

This example, which is a slight permutation of the previous example, will be converted into a **DO** loop. When the decrement is at the end of the loop, the compiler can recognize that the statement is executed on every iteration of the loop.

```
static int i;
while(i < n)
   i++;
```

The compiler uses optimization information in order to locate the increment statement. So long as the compiler prints out no warning messages about suppressing optimization for any class of variables, it will locate the increment and transform the loop. If, however, the compiler suppresses optimization of static variables, indicated by a message

```
Use of many calls inhibits static variable optimizations.
```

then the compiler will not be able to locate the increment for *i*, and the loop will not be converted.

(2) The upper bound expression and the step expression for the **DO** loop to be created must not vary within the loop. Fortran (and also most vector units) fix the value for the upper bound and the stride upon entry to the loop; any variation within the loop will be ignored. As a result, it is incorrect to convert any loop in which these expressions vary. This is the most typical cause of C loops not vectorizing: there are many cases where it may appear obvious to you that the bounds of a loop do not vary, but it is not obvious to the compiler. To reexamine a previous example, if you write a subroutine to be called from Fortran (and thereby has call-by-reference parameters), you will likely pass a loop bound as a parameter. The second example in this chapter does just that:

```
main(double *a, int *n)
{
   int  i;

   for(i=0; i<*n; i++)
        a[i] = (double) i;
}
```

Because *n* appears to be a random pointer to the compiler, it is possible that it overlaps some storage location in *a*, so that the assignment to *a* in the loop could cause *n* to change. As a result, the compiler cannot convert this loop. Since the compiler uses optimization information to determine whether the stride or bound varies within the loop, any inhibition of optimization information will also restrict loop conversion. Note that the compiler option **-safe=loops** has been supplied for just this case; if you use it, the compiler will bypass this check and assume that the expressions do not vary within the loop.

(3) If the controlling logical operation is < or >, then the step expression must be a constant known at compile-time. When the compiler is given a loop of the form

```
i = b;
while (i < n)
   i = i + m;
```

the analogous Fortran **DO** loop is

```
DO i = b, xxx, m
ENDDO
```

where *xxx* is *n-m* if *m* is positive and *n+m* if *m* is negative. The compiler cannot know which form to generate without knowing the sign of *m*.

(4) There can be no branches from outside the loop to inside the loop. This is legal for C looping constructs; it is not (except for a very restricted case) for the Fortran **DO** loop.

While these conditions may sound somewhat restrictive, in practice you should find that most of the loops you write will be automatically converted with some occasional help from **-safe=loops**.

Write Vectorizable DO Loops

The Stardent 1500/3000 vectorizer considers all loops in a loop nest to be viable candidates for vectorization, and quite often vectorizes outer loops. However, there are a few types of **DO** loops which the vectorizer will not examine for vectorization. Most of these are loops which would require large amounts of compile-time to examine, and which have a very low probability of vectorizing.

(1) Loops that are controlled by an ASIS directive are completely untouched by the Stardent 1500/3000 vectorizer.

(2) Loops that contain calls to user functions that have not been declared in either a PPROC or VPROC directive are not examined.

(3) Loops that contain a branch that exits the loop, e.g.

```
          DO I = 1, N
             IF (A(I) .EQ. 0.0) GOTO 100
          ENDDO
      100 CONTINUE
```

are not analyzed. Any loop surrounding this loop is also not analyzed by the vectorizer, although "sibling" loops will be.

(4) Loops that are controlled by a variable of any type other than integer will not be examined by the vectorizer. Floating point loop variables generally mean that nothing would vectorize even if the vectorizer examined the loop; short and byte loop variables (if enabled) would cause the 2.2/3.0 release of the vectorizer to generate inefficient vector code, and also have potential overflow conditions that are hard to check.

(5) If a loop contains some form of input-output statement (e.g. WRITE) as well as a vectorizable inner loop, it will not be analyzed (but the inner loop will be). For example

```
DO I = 1, M
   DO J = 1, N
      A(I,J) = 0.0
   ENDDO
   WRITE(6,*)
ENDDO
```

the *I* loop will not be analyzed in this example, but the *J* loop will. This is a fairly common programming paradigm, and it is very rare that the outer loop can be successfully vectorized.

All of the cases listed above will produce appropriate informatory messages, either as warning error messages or in the vector report facility.

Use COMMON and EQUIVALENCE Carefully

The Stardent 1500/3000 compiler uses very sophisticated dependence tests to determine when different references may access the same memory location. For most cases, these tests have no problems dealing with COMMON statements and EQUIVALENCE statements. However, there are two exceptions that can inhibit code vectorization that users should avoid.

Avoid EQUIVALENCES into COMMON

When arrays are **EQUIVALENCED** into **COMMON** and used in loops that have symbolic bounds, the Stardent 1500/3000 will often assume that vectorization-preventing dependences exist that you did not intend. The following is an example.

```
COMMON A(10), B(100)
REAL C(10)
EQUIVALENCE (A,C)
DO I = 1, N
   B(I) = A(I)
ENDDO
```

If this fragment is run through the Stardent 1500/3000 Fortran compiler, it will not vectorize. The reason is the equivalence of A and C, and the unknown bound N on the **DO** loop. If N is greater than 10, then a dependence will exist that will make vectorization incorrect. For instance, on iteration 11 of the loop, the statement is fetching from A(11) which overlaps B(1); that memory location was stored into on the very first iteration of the loop. The user program is behaving incorrectly in this example by reading past the end of A. However, this type of behavior is not uncommon in older Fortran programs. Normally the Stardent 1500/3000 vectorizer assumes that a user program never violates a bound on a subscript. This example illustrates the one exception—an access into a **COMMON** block which also contains an **EQUIVALENCE**. If the **EQUIVALENCE** of C into the **COMMON** block is removed, or if the bound on the loop is replaced with a constant that is less than or equal to 10, the Stardent 1500/3000 compiler vectorizes the loop.

Avoid EQUIVALENCED Scalars

The Stardent 1500/3000 vectorizer current will not expand an **EQUIVALENCED** scalar that is defined within a loop. Thus, the following loop will not be vectorized.

```
REAL X(100), Y(100)
EQUIVALENCE (S,T)
DO I = 1, N
   T = X(I) + Y(I)
   X(I) = T / 2.0
ENDDO
```

For this loop to be vectorized, the scalar *T* must be expanded into a temporary array. The vectorizer uses scalar optimization information in order to determine exactly how the expansion should

be done. Since scalar optimization information is not accurate for **EQUIVALENCED** variables, the compiler is unable to determine how the scalar should be expanded. Later releases of the compiler will be able to vectorize cases such as this. For now, you can avoid the problem by either eliminating the **EQUIVALENCE** or by replacing the equivalenced variable with a temporary in the loop, as in the following

```
REAL X(100), Y(100)
EQUIVALENCE (S,T)
DO I = 1, N
   TEMP_T = X(I) + Y(I)
   X(I) = TEMP_T / 2.0
ENDDO
T = TEMP_T
```

Use Recognized Patterns

The Stardent 1500/3000 vectorize employs a very general theory of dependence to vectorize statements. However, in one special case, it must do pattern matching in order to vectorize statements. Additionally, there are certain patterns which can be used to achieve faster performance in C.

A very common operation in linear algebra is finding the index of the element of a vector that has the maximum absolute value. This operation is often called "idamax" after the BLAS routine that performs this function. The following loop illustrates the functional definition.

```
IDAMAX = 1
MAXVAL = ABS(A(1))
DO I = 2, N
   IF (ABS(A(I)) .GT. MAXVAL) THEN
        IDAMAX = I
        MAXVAL = ABS(A(I))
   ENDIF
ENDDO
```

This loop cannot be vectorized directly, but it does occur commonly enough in important codes that the Stardent 1500/3000 compilers provide special assembler routines to provide this functionality at vector speeds. The Stardent 1500/3000 compiler uses a fairly general pattern matching mechanism to recognize loops such as the one above and replace them with calls to the appropriate vector routines. Chapter 8 of the Fortran Reference Manual contains a complete listing of all available vector idamax functions.

While the Stardent 1500/3000 compilers recognize very general variations of idamax operations, there is one form that they cannot convert—one in which the vector being examined is a temporary expression, rather than a single array. For instance, the Stardent 1500/3000 compiler will not convert the following

```
IDAMAX = 1
MAXVAL = ABS(A(1) + B(1))
DO I = 2, N
   IF (ABS(A(I) + B(I)) .GT. MAXVAL) THEN
        IDAMAX = I
        MAXVAL = ABS(A(I) + B(I))
   ENDIF
ENDDO
```

The Stardent 1500/3000 compiler calls the idamax functions with the address of the vector to be examined. With an expression such as *A(I) + B(I)*, there is no location in memory that holds the entire vector, so the compiler cannot perform the conversion. Later releases of the compiler will automatically allocate a temporary vector in this instance. For now, it is usually very simple and much faster to create your own temporary

```
DO I = 1, N
   T(I) = A(I) + B(I)
ENDDO
IDAMAX = 1
MAXVAL = ABS(T(1))
DO I = 2, N
   IF (ABS(T(I)) .GT. MAXVAL) THEN
        IDAMAX = I
        MAXVAL = ABS(T(I))
   ENDIF
ENDDO
```

Since C does not contain an absolute value operator, writing a C loop that converts into an idamax call is a little more difficult. However, it is not impossible. Because the Stardent 1500/3000 hardware has the capability of doing absolute values, minimums, and maximums directly at a speed that is much faster than doing the equivalent set of compares, the Stardent 1500/3000 compiler will convert IF-THEN-ELSE patterns into appropriate ABS, MIN, and MAX operations. For instance, if you compile the following C loop at **-O2 -vreport**

```
main()
{
   double a[100], b[100], c[100];
   int i;

   for(i=0; i<100; i++) {
```

```
            if (b[i] < c[i])
               a[i] = b[i];
            else
               a[i] = c[i];
        }
    }
```

you will see the following vector report

```
    for ($$iv = 0; $$iv != 99; $$iv += 32) {
         $$rv = MIN(99, 31 + $$iv);
         $$vl = $$rv - $$iv + 1;
         DO VECTOR ($$I1 = $$iv; $$I1 != $$rv;$$I1++) {
            a[$$I1] = MIN(b[$$I1], c[$$I1]);
         }
    }
```

The compiler has converted the if-then-else sequence directly into a min operation, which executes much faster on the Stardent 1500/3000 hardware. In places where you have abs, max, and min operations coded as if tests, you should make sure that the compiler is recognizing and converting them. If it is not, you can speed your code up by changing the code so that it does.

Coding Conditional Vectors

Vector machines are not well designed for conditional operations. As a result, it is often the case that conditional codes will run slower in vector than in scalar. In order to avoid this situation, you need to understand the properties that lead to good conditional vector code.

Understand Conditional Vector Hardware

One of the key elements of obtaining good performance on vector operations is understanding how the Stardent 1500/3000's conditional vector hardware works. When you write a conditional loop that vectorizes, such as the following

```
DO I = 1, N
      IF (A(I) .GT. 0.0) THEN
         B(I) = A(I) * 2.0
      ENDIF
ENDDO
```

the compiler will generate the following code:

(1) First it will do a vector comparison of the elements of A against 0.0. It will set the results of the comparison into a mask register.

(2) Next it will do a multiply of all the elements of A by 2.0. *All* the elements are multiplied, not just the ones where A is greater than 0.0.

(3) Finally, it will store the results into B under the control of the logical mask from step 1. The time required for this masked vector store is essentially equal to the time for the unmasked store.

This behavior is very different from the scalar case, in which the logical condition is evaluated, and if it is false, nothing else is done. If the logical condition is going to be false a fair amount of the time, then the conditional scalar code will execute faster than the unconditional scalar analog. The vector version is different; even if A is negative at every location, the conditional vector version will still take *longer* than the unconditional vector version.

Because of this difference in behavior, conditional codes that execute very well in scalar may execute very poorly in vector. For example,

```
DO I = 1, N
     IF (COLOR(2,I) .LE. BLUE_THRESHOLD) THEN
          COLOR(3,I) = COLOR(3,I) + DIFF * BLUE(1)
     ELSE IF (COLOR(2,I) .LE. GREEN_THRESHOLD) THEN
          COLOR(3,I) = COLOR(3,I) + DIFF * GREEN(1)
     ELSE
          COLOR(3,I) = COLOR(3,I) + DIFF * WHITE(1)
     ENDIF
ENDDO
```

Since the conditionals are mutually exclusive in scalar, this code will execute one addition and one multiply for each iteration of the loop. If vectorized, however, the fragment will execute 3 additions and 3 multiplies for each iteration of the loop, and will do 3 times as many stores as well. Even though the vector unit is much faster than the scalar unit, it has to do a lot more work. As a result, the vector version may well run slower than the scalar version.

The way to avoid anomalous performance like this is to recode loops slightly. If the above loop is structured so that the additions and multiplies are done only once per iteration, regardless of the branch taken, then the vector unit's speed will become much more apparent. One way of doing this is as follows:

```
DO I = 1, N
     FACTOR = WHITE(1)
     IF (COLOR(2,I) .LE. BLUE_THRESHOLD) THEN
        FACTOR = BLUE(1)
     ENDIF
     IF (COLOR(2,I) .GT. BLUE_THRESHOLD .AND
         COLOR(2,I) .LE. GREEN_THRESHOLD) THEN
        FACTOR = GREEN(1)
     ENDIF
     COLOR(3,I) = COLOR(3,I) + DIFF * FACTOR
ENDDO
```

Here, only the selection of the multiplier is done under mask. The compiler is intelligent enough to keep this in a vector register, so that stores to memory are eliminated. That and the elimination of the redundant arithmetic provides an excellent vector speedup.

Use Structured IF-THEN-ELSE Statements

The compiler is much less effective on loops that contain random **GO TO**'s than it is on codes containing structured **IF-THEN-ELSE** statements. The reason is that it must convert the **GO TO**'s into structured **IF-THEN-ELSE**'s before it can vectorize any statement that may be influenced by any **GO TO**. If these statements vectorize, they will use masked operations in the vector unit. The efficiency of masked operations depends heavily upon the density of true values in the logical condition controlling the mask; when **GO TO**'s are present, it is hard for the compiler to determine whether the masked vector operation will be efficient. As a result, current releases of the Stardent 1500/3000 compilers will not vectorize statements under the control of a branch, although it will parallelize them. Later releases of the compiler will be able to vectorize such statements. However, you will always get slightly better performance if you use structured **IF-THEN-ELSE**'s rather than **GO TO**'s.

The following example will illustrate current limitations of the Stardent 1500/3000 compilers.

```
    DO N = 1, 100
      IF (N .GT. 50) GO TO 95
         M = 10
         A(N) = B(M*N)
      GO TO 98
95  M = 5
      A(N) = B(M*N)
98  CONTINUE
    C(N) = 2.0 * A(N)
    ENDDO
```

The vector report for this example compiled at **O3** is as follows

```
DO PARALLEL ip=1, 100, ds*1
   rp = MIN(100, ip - 1 + ds)
   vl = 1 + rp - ip
   DO N=ip, rp
      IF (N .GT. 50) THEN
         GO TO 95
      END IF
      A(N) = B(N*10)
      GO TO 98
95    A(N) = B(N*5)
98    CONTINUE
   END DO
   vl = 1 + rp - ip
   DO VECTOR N=ip, rp
      C(N) = 2.0 * A(N)
   END DO
END DO
```

The last statement in the loop is run both in parallel and in vector, since it is not under the control of any branch. The remaining statements, which are controlled by branches, are only run in parallel. If the section of code involving the **GO TO**'s is rewritten as a structured IF-THEN-ELSE, as below

```
DO N = 1, 100
  IF (N .LE. 50) THEN
     M = 10
     A(N) = B(M*N)
  ELSE
     M = 5
     A(N) = B(M*N)
  ENDIF
   C(N) = 2.0 * A(N)
ENDDO
```

then the compiler will do a much better job on it. Following is the vector report for this fragment at **O3**. The compiler is able to vectorize and parallelize all the statements in the loop this time.

```
DO PARALLEL ip=1, 100, ds*1
   rp = MIN(100, ip - 1 + ds)
   vl = 1 + rp - ip
   DO VECTOR N=ip, rp
      tv_br1(2 + N - ip) = N .LE. 50
      IF (tv_br1(2 + N - ip)) A(N) = B(N*10)
      IF (.NOT. tv_br1(2 + N - ip)) A(N) = B(N*5)
   END DO
END DO
DO PARALLEL ip=1, 100, ds*1
   rp = MIN(100, ip - 1 + ds)
   vl = 1 + rp - ip
```

```
        DO VECTOR N=ip, rp
           C(N) = 2.0 * A(N)
        END DO
     END DO
```

Avoid && and ||

The C operators && and || require that their right hand sides be evaluated only when the left hand side alone is not enough to determine the truth of the boolean. Thus, when the left side of an && operator is false, the C compiler must guarantee that the right side will not be evaluated. In order to enforce these semantics, the Stardent 1500/3000 C compiler expands && and || expressions into a sequence of branches. For instance, the following condition

```
if (x != 0 && y != 0)
   z();
```

gets converted by the C front end into

```
   if (x == 0) goto 10;
   if (y == 0) goto 10;
   z();
10:
```

There are two important points to note. First, since the Stardent 1500/3000 vectorizer does not currently vectorize code under the control of branches, it will not vectorize code under the control of an && or || operator. Second, even if it did vectorize this code, the generated code would run extremely poorly on the vector unit—slower than scalar execution. To execute correctly, the Stardent 1500/3000 vector unit must evaluate the second logical condition under a mask set by the first logical condition, and must set the mask for the resulting statements to be either the conjunction or disjunction of those. This operation is not possible in the Stardent 1500/3000 vector unit, so that part of the work must be done in the scalar unit. The transfer of information between the two units causes the evaluation to be slower than the scalar evaluation alone.

In loops that you expect to vectorize where the semantics of the condition permit, you will get far more efficient code by replacing the && operator with an & operator and the || operator with an |.

Avoid Unnecessary Error Checks

Vector units achieve fast execution speeds by continuously streaming in operands on which to work. Anything that breaks up the stream of operands will cause the vector unit to operate at less than peak efficiency. There are two common programming paradigms involving checks for error conditions and checks for unnecessary computation that give rise to slower execution speeds.

Many programmers check for error conditions while performing an expensive computation. For instance, in the following code, it is obviously an error when the input matrix A has any value less than or equal to EPSILON.

```
      DO 80 I = 1,100
         IF (A(I) .LE. EPSILON) GO TO 81
         A(I) = A(I) + B(I)*C(I)
80    CONTINUE
      RETURN
81    CONTINUE
      CALL ERROR()
```

The programmer obviously does not expect the error branch to be taken very often, since it appears to be an error. By including the check in with the computation, he has caused the computation to run much slower. In fact, since the error check exits the loop, the Stardent 1500/3000 vectorizer will not attempt to vectorize this loop. If the error check is split off into a separate loop

```
DO I = 1, 100
  IF (A(I) .LE. EPSILON) CALL ERROR()
END DO
DO I = 1, 100
  A(I) = A(I) + B(I)*C(I)
END DO
RETURN
```

then the bulk of the computation will vectorize and run at much faster speeds.

A second cause of slowdowns is checks for unnecessary computations. The following fragment produces very good code on most scalar machines; it produces very poor code on most vector machines.

```
DO I = 1, 100
      IF (C(I) .NE. 0.0 .AND. B(I) .NE. 0.0) THEN
          A(I) = A(I) + B(I) * C(I)
      ENDIF
ENDDO
```

The code avoids doing the multiply and addition whenever the product would be zero. On a scalar machine, the floating point multiply and add are expensive enough to make the checked code run faster. However, on a vector machine, it is much faster to remove the check. The Stardent 1500/3000 vector unit executes both the multiply and the add even when the check is present; the check just causes the results of that iteration to be ignored. As a result, the following code will execute faster in vector on the Stardent 1500/3000.

```
DO I = 1, 100
     A(I) = A(I) + B(I) * C(I)
ENDDO
```

Don't Single Out Specific Iterations

Anything that interrupts the continuity of computation will slow a vector unit down. One common programming practice that does just such an interruption is singling out one loop iteration for special work. For instance, one way of coding a MIN operation is

```
DO I = 1, 100
     IF (I .EQ. 1) T = A(I)
     T = MIN(A(I), T)
ENDDO
```

This code will not execute well on a vector unit, because the computation must be checked on each iteration. The check is mostly wasted, because it will only be true on one iteration. It is far better to roll that check off outside the loop.

```
T = A(1)
DO I = 2, 100
     T = MIN(A(I), T)
ENDDO
```

Another example of this kind of programming is singling out diagonals of arrays.

```
DO I = 1, N
      DO J = 1, N
         IF (I .NE. J) THEN
              A(I,J) = A(I,J) / A(I,I)
         ENDIF
```

```
            ENDDO
        ENDDO
```

In this case, it is more efficient to split the inner loop into two loops.

```
        DO I = 1, N
            DO J = 1, I-1
                A(I,J) = A(I,J) / A(I,I)
            ENDDO
            DO J = I+1, N
                A(I,J) = A(I,J) / A(I,I)
            ENDDO
        ENDDO
```

Boundary conditions in differential equation codes are the most common occurence of this type of construct. The Stardent 1500/3000 compilers perform some of these transformations now; later releases will do more.

Choose Appropriate Conditional Hardware

The Stardent 1500/3000 vector unit has two modes for doing conditional vector operations. One mode involves using mask registers and has been described previously. In this mode, the hardware operates on every element and a bit in the mask register controls whether the result is stored. The other mode involves compression of vectors. That is, the logical condition controlling the computation is evaluated, and those elements corresponding to true locations are compressed into a small vector. That vector is operated on unconditionally, then the results are scattered back out to memory according to the truth value of the logical expression.

The most efficient mode for a particular computation depends upon the number of true elements in the controlling logical expression. If most of the elements are true, then mask registers are more efficient, since it is faster to just skip a few elements than to compress them out. On the other hand, if only a few elements are true, then compression is faster—the time spent compressing the elements is recovered by the short vector operations that follow. For the moment, the Stardent 1500/3000 compilers only generate mask register conditional code.

If you have a computation on a vector that you know is very sparse, it may be worthwhile to pack and unpack it yourself. Following is an example of how you can do so.

```
      J = 1
C     COPY THOSE ELEMENTS TO TO TEMPORARY ARRAY.
      DO I = 1, N
       IF (B(I) .NE. 0)  THEN
C T(J) IS THE TEMPORARY ARRAY.
         T(J) = B(I)
C S(J) REMEMBERS THE ORIGINAL POSITIONS OF THE T ELEMENTS.
         S(J) = I
         J = J +1
       ENDIF
      END DO
C OPERATE ON THE NON-ZERO ELEMENTS.
      DO I = 1, J-1
       T(I) = T(I) + FUNCTION(T(I))
      END DO
C PUT THE RESULTS IN THE ORIGINAL ARRAY.
      DO I = 1, J-1
       B(S(I)) = T(I)
      END DO
```

Use Double Precision

The Stardent 1500/3000 hardware is somewhat unique in that double precision operations take exactly the same amount of time as single precision operations. In fact, there is one specific instance where single precision is *slower* double precision. The memory unit in the Stardent 1500/3000 is banked at 8 bytes; that is, whenever you do a fetch or a store from memory, you fetch or store 8 bytes, regardless of the size you request. This size is (deliberately) the size of a double precision operand; it is the size of two single precision operands. As a result, when you do a vector load or store of a set of contiguous single precision operands, you will access the same bank twice in a row. The bank will not respond to the second request until the first request has completed. This causes vector loads and stores of contiguous single precision operands to run significantly slower than vector loads and stores of contiguous double precision operands. The net effect is that well-written vector codes will often execute much faster in double precision than in single precision.

Unfortunately, it is not possible to say definitively that double precision always executes faster than single precision. The single precision versions of most math libraries execute faster than the analogous double precision routines, because less precision is necessary. Also, converting to double precision doubles the amount of space allocated for data, which can cause paging slowdowns. The net effect is that double precision is faster for some codes and slower for others. However, the Stardent 1500/3000 Fortran compiler contains two options (**-all_doubles** and

-double_precision) which make it very easy to evaluate the effects of using double precision. For many well-written vectorizable single-precision codes, these options can provide a significant speedup.

Compiler releases subsequent to 2.2/3.0 will automatically transform contiguous single-precision vector memory accesses to eliminate memory bank conflicts. At that point, using double precision will provide less benefit.

Use Integers; Avoid Shorts

The Stardent 1500/3000 hardware has facilities for working on integers; it does not have any facilities for working on shorts or bytes. While these operations can be done by combinations of the appropriate integer hardware, there is a fair amount of overhead involved, and the resulting code does not utilize the full capabilities of the Stardent 1500/3000. Unless memory is an absolute constraint, it is almost always beneficial to convert codes using shorts and bytes to integers.

Loop Unrolling

Loop unrolling is a process used to simulate vector processing on some scalar machines. It allows the scalar machine to start an operation before another has finished. Loop unrolling duplicates sections of code within a loop so that each section operates with a different increment of the loop variable. You may have unrolled loops to enhance performance on scalar machines. Stardent 1500/3000's compiler usually vectorizes loops whether they are unrolled or not, but re-rolled loops execute a little faster on Stardent 1500/3000 hardware. The compiler automatically unrolls a loop if unrolling is necessary for optimization.

In the following examples, the first **DO** routine executes faster than the **DO** routine that increments in steps of 5.

```
DO I = 1, N
  X(I) = X(I) + Y(I)
END DO

DO I = 1, N, 5
  X(I) = X(I) + Y(I)
  X(I + 1) = X(I + 1) + Y(I + 1)
  .
  .
  .
  X(I + 4) = X(I + 4) + Y(I + 4)
```

```
      END DO
```

The primary place in which you might want to unroll loops is when you have contiguous single precision vectors. In that case, unrolling the loop once will eliminate memory bank conflicts.

Data Storage

Fortran stores the elements of a multi-dimensional array by columns, a procedure called *column-major format*. That is, the left-most subscript varies fastest as elements are accessed in storage order. Then the 3 x 3 matrix

$$\begin{matrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{matrix}$$

appears in memory with each column's elements stored together.

memory(1) = $A_{1,1}$
memory(2) = $A_{2,1}$
memory(3) = $A_{3,1}$
memory(4) = $A_{1,2}$
memory(5) = $A_{2,2}$
memory(6) = $A_{3,2}$
memory(7) = $A_{1,3}$
memory(8) = $A_{2,3}$
memory(9) = $A_{3,3}$

Suppose your program includes an array of **Pressure, Volume,** and **Temperature** for each of 100 grid points. You could arrange the data in one array as **PVT(3,100)** with each column representing the values for a single grid point. Or you could arrange the data in the array **PVT(100,3)**. Then calculations of **Pressure**, for example, involve elements in a single column, and the values of these elements are in adjacent locations in memory. The hardware operates more efficiently on long contiguous vectors.

While contiguity is not required to achieve optimal performance, it is one stride which is guaranteed to give good performance. Since Stardent 1500/3000's have either 8 or 16 memory banks, strides that are multiples are 8 are always bad strides. Also, strides that are very large can cause paging effects. However, with those exceptions, all other strides should produce roughly equivalent performance.

Note that the semantics of C (and all other programming languages) is backwards from Fortran. In C, the rightmost subscript varies the fastest. As a result, you should reverse this logic

when writing C programs.

Scalar Temporary Variables

Whenever a scalar variable is defined in a loop, the compiler must be able to expand that scalar into a vector temporary before the loop can be vectorized. The Stardent 1500/3000 compiler is able to expand most common uses of scalars, but there are a few points to be aware of.

(1) In order to avoid useless expansion of scalars with no increase in execution speed, the Stardent 1500/3000 compiler will only expand a scalar when *all* the statements that scalar is used in can be vectorized. Thus, if you use a scalar in a statement that cannot be vectorized (a WRITE, for instance), it may inhibit vectorization of an entire loop. To give a specific example, the Stardent 1500/3000 compiler will not vectorize the following loop:

```
DO I = 1, N
   T = A(I) + B(I)
   C(I) = T
   PRINT *, 'Current iteration value ', T
ENDDO
```

If the PRINT statement is removed, the compiler will expand T into a vector temporary and vectorize both statements.

(2) The Stardent 1500/3000 compiler is able to expand many patterns that other compilers will not. For instance, the compiler will expand "wrapped scalars" such as

```
DO I = 1, N
   A(I) = T
   T = B(I) + C(I)
ENDDO
```

giving the following vector report:

```
DO iv=1, N, 32
   tv_t(1) = T
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      tv_t(2 + I - iv) = B(I) + C(I)
      A(I) = tv_t(1 + I - iv)
   END DO
   T = tv_t(2 + rv - iv)
END DO
```

The one thing the compiler must be able to determine to expand a scalar is the fact that the first definition is executed unconditionally on every iteration of the loop. One that is executed conditionally, say

```
DO I = 1, N
    IF (X(I) .NE. 0.0) T = X(I)
    Y(I) = T ** 2
ENDDO
```

will not be expanded; to do so, the compiler would have to create code similar to the following

```
DO I = 1, N
    IF (X(I) .NE. 0.0) THEN
       T(I) = X(I)
    ELSE
       T(I) = X(I-1)
    ENDIF
    Y(I) = T(I) ** 2
ENDDO
```

This transformation is beyond the capabilities of the compiler at present. Most of the cases in which the compiler does not expand a scalar result from the compiler's inability to determine clearly that the scalar is defined on every iteration of the loop. For instance, the compiler will not vectorize the following.

```
DO I = 1, 100
    IF (A(I) .LT. 0.0) THEN
       T = A(I)
    ENDIF
    IF (A(I) .GE. 0.0) THEN
       T = B(I)
    ENDIF
    B(I) = T
ENDDO
```

The compiler is intelligent, but not intelligent enough to realize that the conditions on the IF statements encompass all possibilities, so that T is defined on every iteration. Changing the program slightly

```
DO I = 1, 100
    IF (A(I) .LT. 0.0) THEN
       T = A(I)
    ELSE
       T = B(I)
    ENDIF
    B(I) = T
ENDDO
```

makes that fact clear to the compiler, allowing it to vectorize the loop.

(3) When the last value of a scalar is used outside the loop, the compiler must generate slightly worse code to expand the scalar, because it must save the last value. You will get faster code if you ensure that the scalar temporaries used in loops are not used elsewhere. Note that programs can reuse scalars in subtle ways that compilers must detect, but that users do not always realize. For instance, the following fragment

```
SUBROUTINE LAST_USE(X,Y,T)
REAL X(100), Y(100)
DO I = 1, N
   T = X(I) + Y(I)
   Y(I) = T / 2.0
ENDDO
END
```

will generate the following vector report

```
DO iv=1, N, 32
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      tv_t(2 + I - iv) = X(I) + Y(I)
      Y(I) = tv_t(2 + I - iv) * 5.0E-01
   END DO
   IF (rv .EQ. N) THEN
      T = tv_t(2 + rv - iv)
   END IF
END DO
```

Because Fortran supports call-by-reference, the compiler must save the last value of T to send back to the calling routine. This forces the extra IF test at the end, which slows down the code. Whenever you see an assignment like this, you can improve the performance of the vectorized code by using a new temporary or by changing the code so that the compiler can easily determine that the last value is not used.

Ensure The Right Loop Vectorizes

A key element to achieving good performance on the Stardent 1500/3000 is ensuring that the right loops vectorize and parallelize. In particular, it is almost always better if you vectorize and parallelize different loops in a nest; when the strip loop is parallelized, the resulting performance is usually less than optimal. Normally the Stardent 1500/3000 compiler works very hard to get the

right loops optimized, but in many cases, it does not have enough knowledge to do the job right. In those cases, you should use directives to help the compiler generate better code. Some of the most common cases include:

(1) Unknown loop bounds. When the compiler has no knowledge of the bounds on a loop, it assumes that the loop is long enough to execute profitably in vector and in parallel. As a result, it may make bad choices. You can tell whether the compiler knows loop bounds by looking in the vector report; if it has been able to determine the loop bound, it will be a constant there.

(2) Expanding arrays. A reasonably common programming paradigm in many codes is the use of a vector temporary:

```
        DO I = 1, M
           DO J = 1, N
              X(I,J) = 2 * Z(I,J)
              T(J) = X(I,J) + Y(I,J)
           ENDDO
           IF (.NOT. DOIT) GOTO 20
           DO J = 1, N
              A(I,J) = T(J) * 2.0
           ENDDO
20         CONTINUE
        ENDDO
```

Here the vector T is used to carry values from the first J loop to the second J loop. If this example is compiled with the Stardent 1500/3000 compiler, each of the J loops will vectorize and parallelize; the I loop will not parallelize. This type of loop construct generally gives much better performance if the I loop is parallelized. The Stardent 1500/3000 compiler does not parallelize the I loop, because it would have to allocate storage on each processor for the whole T array. If T is very large, this can result in a significant increase in memory requirements. Furthermore, it is difficult for the compiler to determine whether T is used again outside the loop; if it is, the compiler must save the last values of "temporary" T computed in the "real" T. Because of these difficulties, the compiler deliberately does not expand vector temporaries such as this. If you have many examples like this, it could be worth some time rewriting the code to either expand T to have an extra dimension for the I loop, or to fuse the J loops.

(3) Beware unusual cases. In general, the Stardent 1500/3000 compiler tries to vectorize and parallelize separate loops in order to hedge its bets against short loops. For most cases,

this is exactly the right strategy; in a few cases, it is the wrong strategy. Consider the following code for doing matrix vector multiplication:

```
DO I = 1, M
   C(I) = 0.0
   DO J = 1, N
     C(I) = C(I) + A(I,J) * B(J)
   ENDDO
ENDDO
```

Compiling **-O3** yields the following vector report.

```
DO PARALLEL ip=1, M, ds*1
   rp = MIN(M, ip - 1 + ds)
   vl = 1 + rp - ip
   DO VECTOR I=ip, rp
      C(I) = 0.0
   END DO
   DO I=ip, rp
      DO iv=1, N, 32
         rv = MIN(N, 31 + iv)
         vl = rv - iv + 1
         DO VECTOR J=iv, rv
            C(I) = C(I) + DOT_PRODUCT(A(I, J), B(J))
         END DO
      END DO
   END DO
END DO
```

The compiler has parallelized the outer loop, vectorized the outer loop around the first statement, and vectorized the inner loop around the second statement. This is exactly the right strategy if M happens to be relatively small (say 32 or less). However, if M is very large, it is not the right strategy. In general, when you vectorize different loops in the same loop nest, you are going to lose some performance, because register reuse will not be as good and some extra synchronization is required. Thus, vectorizing the I loop around the first statement and the J loop around the second statement involves some loss of performance; if M is big enough, this loss is enough to offset the protection provided by optimizing separate loops. Thus, for large M, better performance is provided by the following vectorization:

```
DO PARALLEL ip=1, M, ds*1
   rp = MIN(M, ip - 1 + ds)
   vl = 1 + rp - ip
   DO VECTOR I=ip, rp
      C(I) = 0.0
   END DO
   DO J=1, N
      DO VECTOR I=ip, rp
```

```
            C(I) = C(I) + A(I, J) * B(J)
         END DO
      END DO
   END DO
```

This vectorization is easily obtained by inserting a PBEST and VBEST directive around the outer loop.

(4) Vectorize outer loops for register reuse. One of the reasons that the latter version of matrix-vector multiply runs faster is that the compiler is able to keep the section of array C that it is computing in a vector register throughout the computation. That is, the first statement zeros out a vector register; results are continually fed into that register throughout the J loop; and only after termination of the J loop is the result stored out to memory. The compiler utilizes a very general strategy to utilize vector registers that speeds up code in a number of situations. The form which is most easily recognizable (and also that most prevalent) is illustrated in the vector report above: the result operand inside the DO VECTOR loop (C(I) in this case) is invariant with respect to the loop immediately outside the DO VECTOR loop (the J loop). The most frequent way of obtaining this situation is by vectorizing an outer loop, and actually moving the vector loop inside of the invariant loop.

Beware Coupled Reductions

The Stardent 1500/3000 compilers recognize reductions by means of special case pattern matching. The patterns recognized are very general, so most of the time you will have no problems coding reductions. However, there is one very special case of coupled reductions which the compiler does not recognize. Following is an example.

```
DO I = 1, N
    IF (A(I) .GT. 0.0) THEN
        T = T + A(I)
    ELSE
        T = T - A(I)
    ENDIF
ENDDO
```

Here, the reduction is actually spread across two statements. The Stardent 1500/3000 compiler will not recognize the reduction, because it only examines one statement at a time for reductions (with the single exception of count reductions). Vectorizing these statements would also change the execution order somewhat

drastically, passibly causing the results to change due to variations in associativity. Future releases of the Stardent 1500/3000 compiler will recognize coupled reductions so long as floating point associativity is assumed.

Avoid Vectorized Structures

Because Fortran is by far the language of choice for most Stardent 1500/3000 programmers writing numerical programs, the Stardent 1500/3000 Fortran compiler has been far more extensively used than the C compiler at higher optimization levels. As a result, some features that are C specific have not yet been implemented or extensively tested. The most common of these are structure references. For instance, the Stardent 1500/3000 C compiler will not run the following program on the vector hardware.

```
typedef struct
{ double real;
  double imag;
} complex;

main()
{
   complex a[100], b[100], c[100];
   int i;

   for(i=0; i<100; i++)
      a[i].real = b[i].real + c[i].real;
}
```

Currently, the Stardent 1500/3000 compiler is unable to compute the stride for operations involving structures, and as a result, will run these in scalar even though the vector report reports vector execution. Later releases of the compiler will be able to handle such references. For now, critical sections of code that rely on structure references can be rewritten similar to the following:

```
typedef struct
{ double real;
  double imag;
} complex;

main()
{
   complex a[100], b[100], c[100];
   double  *ap,    *bp,     *cp;
   int i;

   ap = &(a[0].real);
   bp = &(b[0].real);
   cp = &(c[0].real);
```

```
    for(i=0; i<100; i++)
        ap[2*i] = bp[2*i] + cp[2*i];
}
```

Use –fast

The Stardent 1500/3000 compilers focus intently on precision of results, and avoid many optimizations which may affect the last bit or two. For instance, the expression a/(b/c) involves two divides, which are very expensive. Algebraically, this expression is exactly equivalent to the expression (a*c) / b, which will evaluate much more quickly, because a multiply is much faster than a divide. Unfortunately, on floating point hardware, this equivalence is not exact—there can be some differences in the last few bits. As a result, the Stardent 1500/3000 compilers will not perform this optimization. The **-fast** option on both compilers enable optimizations such as this, which may lose a little precision in the last few bits. For most codes, this loss of precision is perfectly acceptable. As a result, you may want to try this option.

Avoid Slow Operations

Even when vectorized, there are some operations which are slow. Divide and square root are the chief examples. The compiler performs common subexpression elimination, so it automatically removes many redundant divides and square roots, but it is not perfect. Any recoding you can do to reduce the number of these operations will enhance the performance of your code.

Summary

The following briefly summarizes the programming techniques that take the best advantage of the automatic optimizations of Stardent 1500/3000's Fortran and C compilers.

- Write vectorizable loops.
    - If you use **DO WHILE** or **for, while,** or **repeat while** loops, ensure that they can be converted into **DO** loops.
    - Avoid exit branches and function calls in loops.
    - Use integer loop control variables.
- Use **COMMON** and **EQUIVALENCE** carefully.
- Use the patterns that the compiler recognizes.

- Be careful coding conditional operations.
  - Understand the limitations of vector conditional hardware—the more conditionals involved, the more likely the code will run faster in scalar.
  - Use structured if-then-else statements.
  - Avoid && and ||.
  - Avoid unnecessary error checks.
  - Do not single out specific iterations.
  - Compress vectors for very sparse operations.
- Use double precision and integer operands.
- Do not unroll loops to enhance performance.
- Be aware of data storage considerations. Long contiguous vectors are best.
  - In Fortran, columns are contiguous. In C, rows are.
- Be aware of the restrictions required to expand scalars.
  - Ensure that all uses of the scalar can be vectorized.
  - Ensure that there is a clear definition.
  - Avoid **EQUIVALENCED** scalars.
  - Avoid last use scalars.
- Check the vector report to ensure that the right loop vectorizes.
- Avoid coupled reductions.
- Avoid structure references.
- Use **-fast**
- Avoid slow operations.

Unformatted I/O In Fortran

There are two programming techniques available that can improve program performance when using unformatted I/O. The first consideration is the alignment of data and the second is using arrays versus **DO** loops.

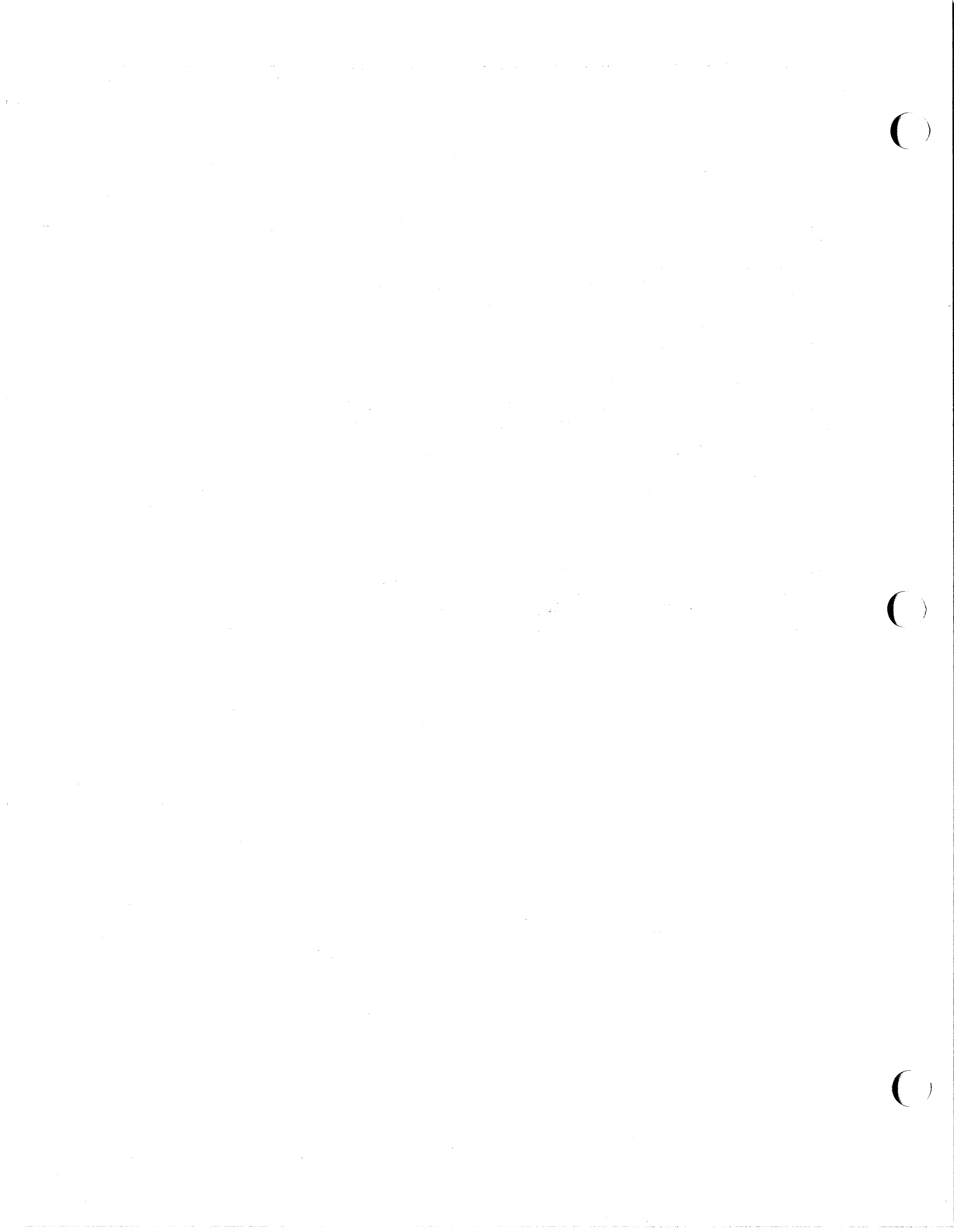
Data Alignment

The file system block size is 4096 bytes, and if data can be arranged to stay on block boundaries there is a small speed improvement. When coding any I/O that requires the statement specifier **RECL** to be set in the **OPEN** statement, use multiples of 4096 bytes or 1024 REAL's, and so on. When using sequential unformatted I/O without specifying a record length, try to align the data using the formula 4096 * n - 8, where n is the number of 4K byte blocks you need.

If aligning data on block boundaries is not possible, try to at least avoid mixing odd size data elements with elements that are multiples of words. Moves are faster if they are word to word or double word to double word.

Arrays Versus DO Loops

In general it is faster to move data in large chunks as opposed to moving data elements one item at a time. It is better to use long arrays or vectors than to use implied **DO** loops. The Titan attempts to remove **DO** loops which can use block data moves, but this is not always possible. Any help that the programmer can provide in removing **DO** loops is useful.



EXPLICIT PARALLEL PROGRAMMING

CHAPTER EIGHT

Preparing a program for use on multiple processors is called *parallelization*. Automatic parallelization of program code is performed by the Stardent 1500/3000 compilers and, for many programs, compiler generated parallelism is sufficient. Compiler generated parallelization is normally applied to nests of loops. However, there may be programs that could be parallelized at a much higher level so that several blocks of program code are included in a single parallelized chunk. Only user-directed parallelization can create this effect because compilers simply are not sophisticated enough to do this on their own. This chapter describes mechanisms that you can employ to direct the compilers to parallelize code explicitly at a high level.

The most important issue in programming for parallel execution is the correct handling of memory that might be shared among parallel processes. The chapter discusses setting up separate memory spaces for each process. Setting up separate spaces addresses the issue of private memory—accessible only to an individual process, as compared to public memory—accessible to all processes. The chapter also discusses static and dynamic memory. Finally, there are times when memory **must** be shared among processors, but where certain compiler optimizations might prevent variables from being shared. This chapter describes how to prevent the compiler from doing optimizations that would prevent a planned sharing of variables between processes.

The chapter also shows how communications can be set up among processes, through the use of *semaphores*, a mechanism by which only a single process at a time can own and use an item that must be shared among processes. Generally, when more than one process is used to accomplish a given task, the processes must communicate. To ensure accuracy of the results, explicit arrangements must be made and communications protocols must be established and followed. To support the interprocess communication and synchronization, this chapter defines parallel communication functions.

In the chapter, Fortran and C programming issues are broken into separate sections to allow someone interested in only one of the languages to locate items of interest and to skip the parts that do not apply to a particular problem.

Serial and Parallel Programs

A *serial section* of a program is a section which only one processor is executing. This single path through the program is called a single *thread* of execution. Given the same set of data inputs to a serial program, it produces the same set of outputs each time it is run. If one ignores such things as interrupt servicing, step-trace of the execution of a serial program is exactly the same, step for step, from one run to another. Most conventional programs are serial.

In some programs, there are sections which could operate on multiple sets of data simultaneously. Such sections are called *parallel* because multiple processors could be effectively used to speed up execution by assigning each processor to a different block of input data and by letting the processors run simultaneously. Each processor is executing one of the *parallel* or *multiple threads of control* through the program. When all processors have completed running their assigned portions of the program code, all but one processor become idle (waiting to be reassigned to some other thread) and one processor continues on to execute the next serial portion of the program. Generally, the several threads must communicate among themselves to allocate input data, to summarize results, and to detect termination of the parallel section; communication requires well-understood protocols.

A properly written program which uses parallel threads of execution usually obtains the same set of outputs, given the same inputs as a program that runs only one serial thread all the way through. However, the operating system has to schedule multiple processors to handle the parallel threads; thus, it is virtually assured that the sequence in which the threads are executed varies from run to run. Because of this variability, programs which accumulate partial results from parallel threads into one final result may produce different answers when the accumulation process is sensitive to the order of its input data. For example, floating point summation is sensitive to the order of the input data. Such numeric differences are usually very small. Any other output differences are almost invariably a symptom of a programming error.

A program intended for parallel execution must take special care of its data. In particular, a program might have to ensure that separate processors receive private copies of certain data elements or that data elements that may be modified or used by more than one processor are protected against simultaneous access that might create an incorrect result. These techniques are discussed later in this chapter.

Opportunities to use the vector processing capabilities and the parallel processing capabilities can be automatically detected by the Stardent 1500/3000 compilers by their own careful analysis of the loop structures of a program's code. In fact, the Stardent 1500/3000 compilers can rearrange the program code to make the most of such opportunities. However, when only the application programmer is explicitly aware of the relationships between the data that is being manipulated and the functions that are being used to work on that data, only you can tell for sure that a parallel sequence of calculations does not affect the correctness of the result. Thus, only you can parallelize the code. These cases arise because real world problems are richer than programming languages can express and what a language cannot express, a compiler cannot detect.

Determining Whether Explicit Parallelization Is Appropriate

A program may benefit from explicit parallization if it has these characteristics:

- There may be large chunks of executable code which you know do not have much to do with one another.
- The results from those chunks are independent of one another.
- Those chunks of code do not compete to generate results into the same output variables.

A program with these characteristics clearly has independent sections and it should be obvious that parallel execution could be profitable. **If the program does not have these characteristics, then what follows in this chapter may not be of much interest to you.**

Levels of Parallelism

Programs can execute in parallel at three different levels of *granularity*: *fine-grain*, *medium-grain*, or *coarse-grain*. Fine-grained parallelism occurs when several instructions overlap execution or when vector operations flow through the vector execution pipeline; generally, creation of fine-grained parallelism is the province of the compiler and only indirectly under your control. Medium-grained parallelism occurs when statements, loop iterations, or subroutines execute simultaneously. Setting up for medium-grained parallelism is generally a duty shared with the compiler, where the compiler concentrates on finding opportunities to parallelize loops and where the programmer might have a larger interest in locating subroutines which may run in parallel. Finally, coarse-grained parallelism is provided by processes and pipes (under Unix), and is outside of the scope of this chapter.

To make medium-grained parallelism operate correctly some modificiations to a program might be necessary. Here are the kinds of problems you might have to consider:

- Processors might have to be assigned correctly to individual subroutines.
- If threads share public memory, the program must synchronize access among processors to ensure correct results.
- If threads have private memory, they can use that private memory to accumulate results before copying it to the public memory. Once again, access to the public memory must be controlled.
- If the processors interact with the operating system, they must do so without collision.
- If errors occur, the process must recover appropriately.

Neither C nor Fortran have a good way of expressing these concepts, so functions and macro calls that implement the parallel processing techniques properly have been provided.

Memory Conflicts

Explicit parallel programming requires the explicit management of three kinds of memory:

- Private and public memory

  Private memory is allocated to one specific execution thread; *public* memory is accessible from more than one execution thread.

- Static and dynamic memory

  Static memory is allocated at the beginning of execution; any value stored in static memory remains until changed or until program termination. *Dynamic* memory is allocated to a subroutine at the moment of subroutine execution and is deallocated when the subroutine terminates. Values stored in dynamic memory are retained only as long as the owning subroutine is active; the values are lost when the memory is deallocated.

CAUTION
It is a common and serious error to assume that dynamic memory retains its values; such errors cause bizarre and unpredictable runtime errors, particularly in parallel programs.

- Volatile memory versus register intermediate variables

  This topic is of interest only to C programmers. It deals with how the compiler optimizes program code.

Private Memory

If each processor has its own private memory, and uses that private memory for both writes and reads during the execution of one thread of a parallel process, writes with subsequent reads of this memory always obtain the expected data. The results obtained by that processor will be the same as the results obtained by a single processor running a serial process. You might expect that declaring variables local to a subroutine would provide memory that is protected from simultaneous access by multiple processors; however, this is **not** the case. Though declaring local variables provides memory that is local to the subroutine itself, with names that are unknown to other functions, it does not actually provide memory that is protected from multiple processor access. Here are the reasons.

- In Fortran, local variables act as if they reside in an unnamed local COMMON block. Unless otherwise instructed, the Fortran compiler allocates this block in static, public memory

and all executions of the subroutine access this same block.

- In C, local variables are normally allocated dynamically from the stack. Unless a special library route, *MT_INIT*, is called all the parallel invocations of a particular subroutine from a particular call share the same stack storage.

The Stardent 1500/3000 C and Fortran compilers contain extensions for assigning private memory to processors.

Public Memory

Public memory is accessible to all processors. If write-access to that memory location by multiple processors can cause incorrect results, then that memory location must be protected. That is, access must be moderated by some memory operation that allows access to only one processor at a time.

Imagine a case in which several processors are accumulating a sum in a single memory location. Without a lock to control access, one thread could read the sum value, add its own contribution, and put the new sum back into the public location. This could be a mistake, though, if another thread had done exactly the same thing at the same time.. Depending on which thread finished last, the public sum would include the contribution of one thread or the other, but not both. The Stardent 1500/3000 compilers provide locking mechanisms to ensure correct processor synchronization.

NOTE
Such lock operations have the technical name *indivisible*. This means that opening or closing the lock can not be interrupted by any other operation which would leave the lock in some strange "half-open" state.

Static Memory

Static memory has its location fixed at load time. A value stored in static memory remains unchanged until explicitly rewritten.

Fortran Static Memory:

A COMMON block is normally allocated as static, public memory, accessible to all routines which name the block. The block can be made private to a particular routine by use of the STATIC directive. The static directive causes the COMMON block to behave like a C static variable; it is allocated in static storage, but its name is not externally known. For example, consider this fragment from subroutine *Sub1*:

```
        COMMON /X/ A,B,C
C$DOIT STATIC X
```

and this fragment from subroutine *Sub2*:

```
        COMMON /X/ A,B,C
```

If the STATIC directive were not present, any change to a variable in COMMON block *X* in either routine is immediately reflected in the other routine because the two COMMON declarations actually create references to the same storage. The STATIC directive makes the COMMON block *X* in subroutine *Sub1* completely private. Now when Sub1 references anything in *X*, it is referencing completely separate storage from the COMMON block in *Sub2*; there is no communication between the two. However if *Sub1* is run in parallel, all of the copies of *Sub1* can still communicate through COMMON block *X* because it is static storage.

NOTE
Both the STATIC and the THREADLOCAL directives can be applied to a single COMMON block, making it possible to create threadlocal COMMON blocks not visible outside of the procedure.

C Static Memory:

In C, static memory is that memory assigned to variables which have been declared as global or which have the attribute *static* specified. The distinction between global variables and local variables which are declared as *static* within a subroutine is that static local variables have names that are not externally known. Access to variables in static memory must be appropriately protected to assure correct results.

Dynamic Memory

In Fortran, local variables are normally in static memory so that their values may be implicitly saved from one subroutine invocation to another. To ensure that Fortran local variables are allocated dynamically and are thus safe for parallel execution, compile a parallel routine with the **-safe=procs** option. This option **does not** affect local variables which appear in EQUIVALENCE or NAMELIST statements or which are initialized by any variant of a DATA statement. These variables remain statically allocated.

In C, all local variables not given the *static* attribute are always dynamic.

NOTE
Some Fortran programmers are accustomed to using a SAVE statement to ensure that variables retain their value from one call to the next. The Titan Fortran compiler ignores the SAVE statement; it allocates all variables in static storage as though there is a SAVE in effect. Under appropriate circumstances, such as compiling for parallel execution, variable assignments are moved out of static storage.

Volatile Memory

When programming for multiple processors in C, you may need to explicitly declare variables as *volatile*. A declaration looks like the following:

```
volatile int n;
```

NOTE
Fortran programmers need not concern themselves about this concept.

Once a variable is declared *volatile,* the compiler guarantees that every change to the variable's value is immediately written to memory; the optimizer is not allowed to generate code which holds the changed value in a processor register. The compiler also causes a new load of the variable's value at each reference to the variable. In this way, the value of the variable is always fresh. This freshness is important when several processors are using the variable to communicate, since one processor may change the value of the variable invisibly to another process.

Summary Of Memory Conflict Situations

To ensure the accuracy of the results when multiple processors access public or static memory, memory accesses must be protected to prevent one processor from reading the memory while another is attempting to write it. The mechanism for this protection is described in the next section. Additionally, if more than one processor must access a variable, then that variable's value must be current and in memory, to ensure an accurate result after each processor has used it. For C programmers, using the *volatile* declaration for such variables solves the problem.

NOTE
There is no *volatile* declaration available to Fortran programmers because the compiler ensures that accesses are correct.

Critical Sections Of Code

There may sometimes be critical sections of code for which only one processor at a time should be allowed to run a thread of execution. This occurs when a single global variable is being used by multiple processes where a data structure such as a file descriptor should be accessed by only one process at a time. This is the problem of simultaneous access of public memory locations.

An *atomic operation* can be used to synchronize multiple processors handling public memory. An atomic operation is one that cannot be interrupted. In other words, if one processor begins an operation and another processor subsequently attempts the same operation, the second processor is forced to wait while the first processor finishes. Then and only then can the second processor perform the operation.

One mechanism for gaining exclusive access to a section of code or a data object is the *semaphore,* so-called because it is like a little flag signalling whether it is safe to go ahead. A semaphore is given some initial value (on Stardent 1500/3000s, the value is conventionally 1) which signals that the protected item is free for use. The semaphore is said to be *unlocked*. When a processor wishes to

use the protected item, it executes a *lock* operation for the semaphore; Stardent 1500/3000 has an atomic instruction, *load-and-sync*, which is used to implement the locking operation. Once the process has set the lock, it does whatever it wishes to the item. When processing is finished, the process uses an *unlock* operation to reset the semaphore to its original state.

What happens if a second process comes along and also wants to use the same item? The newcomer also tries to lock the semaphore. However, it finds that the semaphore is already locked (on Stardent 1500/3000s, the semaphore has the conventional value 0 when locked). The process then *spins* on the semaphore, waiting for it to be unlocked. The new process must wait until the original process unlocks the semaphore and releases the protected item.

This scheme works only if each resource is protected by its own semaphore and if each process is careful to check semaphore status before using a protected resource.

On Stardent 1500/3000s, any 32-bit (or one word) memory location may be used as a semaphore. In Fortran, an INTEGER variable applies, and in C, an *int* variable should be used. A program may employ as many semaphores as it needs.

The use of semaphores is discussed in the sections that describe solutions to the public memory problem in Fortran and C.

Allocating Private Memory In Fortran

There are two ways in Fortran to allocate memory that are private to each processor:

- Allocate separate dynamic variables by including them in a named COMMON block for which the THREADLOCAL directive has been applied. This is described more fully in the section *Using THREADLOCAL COMMON In Fortran.*

- Allocate a separate dynamic memory space (a separate stack) for each processor by calling the subroutine *MT_INIT*. Subroutines keep temporaries and parameters on a stack. MT_INIT causes each processor to have its own individual stack; if this procedure is not called, each processor uses the same stack. This results in chaos.

Using THREADLOCAL COMMON In Fortran

Fortran users can define COMMON blocks as *THREADLOCAL*. This directs the compiler to create a separate copy of all variables in that COMMON block for each processor. Because Fortran itself provides no method for declaring the type of a Fortran variable as threadlocal, a compiler directive is used.

Here is a sequence that uses the THREADLOCAL directive.

```
      COMMON /X/ A, B, C
C$DOIT THREADLOCAL X
```

This sequence causes the COMMON block containing variable *A*, *B*, and *C* to be placed in threadlocal storage and causes the names to be externally known. Each processor has its own copy of the COMMON block *X*. If two different procedures which are running on the same processor both contain this declaration, then those two procedures may communicate through the COMMON block X.

There are two important points to note about threadlocal variables:

- Threadlocal variables cannot be initialized in block data statements. Threadlocal storage is not created until the program goes parallel; thus static initializations are useless. You are guaranteed that a threadlocal variable has an initial value of zero. To provide an initial value for a variable initialize it explicitly in an assignment statement at the start of the parallel region.
- It is not possible to assign values to threadlocal variables outside of parallel regions because the variables do not come into existence until the program goes parallel.

Creating Local Stacks (Processor Private Memory) In Fortran

Processor private memory can also be provided by explicitly telling the compiler to allocate a separate workspace for each processor. Then when a processor is assigned to run a particular thread of execution, it uses that private workspace (here called a *stack*) to allocate its local variables.

Fortran users tell the compiler to provide this separate workspace, by calling the following subroutine

```
CALL MT_INIT(stack_size)
```

The *stack_size* can be any value that is large enough to account for the maximum memory space taken up by all local variables and arrays. When in doubt, set *stack_size* to a value of 100,000. This should handle most cases unless very large local arrays are declared. The subroutine *MT_INIT* provides a local storage space for each processor. It *must* be called before any parallel procedure.

NOTE
These functions are defined in a header file: */usr/include/micro.h*

Here is a corresponding function.

```
CALL MT_FINI()
```

This function frees the memory allocated for each processor's private use, however it need not be called. The cleanup code that is executed when a program ends automatically deallocates any storage that a program may have allocated.

Allocating Private Memory In C

There are three ways in C to allocate memory that are private to each individual processor:

- Allocate separate dynamic variables by declaring them as *threadlocal.*
- Allocate variables and arrays by using the memory allocation function *thread_new.*
- Allocate a separate dynamic memory space for each processor (by calling procedure *MT_INIT*). When a procedure is called by an individual processor, the working variables that the procedure uses are assigned to addresses within this dynamic memory that is owned by the individual processor. Variables created in dynamic memory do not retain their values between calls. If you wish to pass values between procedures that are performed by separate processors, you have to pass those values through global variables.

Using Threadlocal Variables in C

C language users can define individual variables as *threadlocal.* This directs the compiler to create a separate copy of these variables for each processor.

Here is an example that uses the *threadlocal* directive:

```
void foo()
{
   threadlocal i;
   for(i=0; i<1000; i++) {
      dosomething(i);
   }
}
```

In this example, *foo* is a function that is to be run by more than one processor. Thus the working variable *i* is declared as threadlocal so that each processor is guaranteed to have a separate copy of that working variable. In the absence of a declaration of *threadlocal,* the variable *i* would be assigned as an offset from the stack pointer. If a separate stack is not assigned (using *MT_INIT,* explained later in this section), separate processors using the same stack would collide on the variable *i*, each trying to assign it a different value on separate iterations of the individual processor's loop.

There are two important points to note about threadlocal variables.

- Threadlocal storage is not created until the program goes parallel; thus static initializations are useless. You are guaranteed that a threadlocal variable has an initial value of zero. To provide initial values for the variables (other than zero), initialize each explicitly with an assignment at the start of the parallel region.
- It is not possible to assign values to threadlocal variables outside of parallel regions because the variables don't come into existence until the program goes parallel.

Allocating Dynamic Threadlocal Storage In C

At times, in addition to using threadlocal variables, it might be necessary to allocate dynamic memory for use by an individual processor. The threadlocal equivalent to the memory allocation functions *malloc* and *free* are *thread_new* and *thread_free* respectively. These are macros (not functions) and are defined in */usr/include/thread.h*. The syntax is:

```
char *thread_new(size)
unsigned size;

thread_free(ptr);
char *ptr;
```

The programmer might expect to store the return value from thread_new into a threadlocal variable. However, this is not required.

Creating Local Stacks (Processor Private Memory) In C

Processor private memory can also be provide by explicitly telling the compiler to allocate a separate workspace for each processor. Then when a processor is assigned to run a particular thread of execution, it uses that private workspace (here called a *stack*) to allocate its local variables. You must, of course, be aware that each of the processors does indeed have local copies of the named variables, and use public memory at some point to merge together intermediate results if appropriate.

C users tell the compiler to provide this separate workspace by calling the following function:

```
MT_INIT(&stack_size);
```

The address of the stack_size variable is needed because this is a macro calling the Fortran equivalent function, for which an address value is required.

The *stack_size* can be any value that is large enough to account for the maximum memory space taken up by all local variables and arrays. When in doubt, set *stack_size* to a value of 100,000. This should handle most cases unless very large local arrays are declared. The function *MT_INIT* provides a local storage space for each processor.

Here is a corresponding function.

```
MT_FINI();
```

This function frees the memory allocated for each processor's private use, however it need not be called. The cleanup code that is executed when a program ends automatically deallocates any storage that a program may have allocated.

Synchronizing Through Public Memory in Fortran

To synchronize access to public memory from different processors, you can implement a semaphore for any memory location. A *semaphore* is a data item which, when accessed through system functions, provides exclusive ownership to the accessor. If a semaphore is used to protect access to a data item or structure, each accessor first attempts ownership by locking the semaphore. If another processor has already locked the semaphore, all other processors attempting the same lock are stalled. On release (unlocking of the semaphore), another processor can gain access to the protected data.

The subroutines used to implement semaphores are:

```
CALL MT_LOCK(semaphore)
CALL MT_UNLOCK(semaphore)
```

MT_LOCK causes the calling processor to wait until the value of semaphore is greater than zero (typically set to a value of 1). When this occurs, the system sets the semaphore to a value of 0 and returns, all in one uninterruptible operation. *MT_UNLOCK* unlocks the semaphore by setting it to a value of 1, also using an uninterruptible operation.

To use these functions within code designed to run in parallel, semaphore must be set to a value of 1 during the serial process sometime before the parallel threads are begin to run. Here's an example:

```
SUBROUTINE PROTOTYPE
DATA PROT_SEMA/1/
COMMON /PROT/ X, Y, Z

REAL*8 X,Y,Z
```

This procedure is about to access at least one of the variables in COMMON block *PROT* and needs to insure that no change can occur during the access period. Thus it attempts to achieve exclusive access by locking the protective semaphore *PROT_SEMA*. Either the lock will succeed or it will fail; if it fails, the procedure will hang on the call to *MT_LOCK*, waiting for success.

```
CALL MT_LOCK(PROT_SEMA)
```

PROT_SEMA has been successfully locked by this procedure. Access and modification of variables in *PROT* may be done freely.

```
...
CALL MT_UNLOCK(SEMA)
```

Upon completion of the need for exclusive access, the procedure unlocks *PROT_SEMA* so that the data in *PROT* can be accessed by other threads.

There are several points to note about this example. First, *PROT* is an ordinary COMMON block, accessible to any procedure in the entire program. Second, the semaphore *PROT_SEMA* is not connected in any way **by the rules of Fortran** to the COMMON block *PROT*; this connection is entirely a convention which must be observed by the programmer. Failure to observe such conventions is a common source of error. Third, *PROT_SEMA* has been forced into static storage because it appears in a DATA statement. Had this not been done, each thread would have had its own copy of *PROT_SEMA* and it would not have functioned as a semaphore between threads. Failure to force semaphores into static memory is another common error.

Finally, *PROT_SEMA* is local to this subroutine while the data it protects (the COMMON block PROT) is accessible by any procedure in the program. If the only possible accessing conflicts on *PROT* appear in this routine, then this setup is suitable. But if other routines might try to access *PROT* during parallel execution, an error may occur. To solve the problem, then, *PROT_SEMA* should be global (that is, in a COMMON block).

Synchronizing Through Public Memory In C

To synchronize access to public memory from different processors, you can implement a semaphore for any memory location. A *semaphore* is a data item which, when accessed through system functions, provides exclusive ownership to the accessor. If a semaphore is used to protect access to a data item or structure, each accessor first attempts ownership by locking the semaphore. If another processor has already locked the semaphore, all other processors attempting the same lock are stalled. On release (unlocking of the semaphore), another processor can gain access to the protected data.

The subroutines used to implement semaphores are:

```
MT_LOCK(semaphore)
MT_UNLOCK(semaphore)
```

MT_LOCK causes the calling processor to wait until the value of semaphore is greater than zero (typically set to a value of 1). When this occurs, the system sets the semaphore to a value of 0 and returns, all in one uninterruptible operation. *MT_UNLOCK* unlocks the semaphore by setting it to a value of 1, also using an uninterruptible operation.

To use these functions within code designed to run in parallel, semaphore must be set to a value of 1 during the serial process sometime before the parallel threads begin to run. Here is an example:

```
struct PROT = {
   double X,Y,Z;
};

prototype()
{
   int PROT_SEMA = 1;   /* should probably be a global
                         * as described in the text below
                         */

   ...
   MT_INIT(100000)
   ...
```

This procedure is about to access at least one of the variables in global data structure *PROT* and needs to insure that no change can occur during the access period. Thus it attempts to achieve exclusive access by locking the protective semaphore *PROT_SEMA*. Either the lock will succeed or it will fail; if it fails, the procedure will hang on the call to MT_LOCK, waiting for success.

```
MT_LOCK(PROT_SEMA)
```

PROT_SEMA has been successfully locked by this procedure. Access and modification of variables in *PROT* may be done freely.

```
...
MT_UNLOCK(SEMA)
```

Upon completion of the need for exclusive access, the procedure unlocks *PROT_SEMA* so that the data in *PROT* can be accessed by other threads.

There are several points to note about this example. First, *PROT* is an ordinary global variable, accessible to any procedure in the entire program. Second, the semaphore *PROT_SEMA* is not connected in any way **by the rules of C** to the global variable block *PROT*; this connection is entirely a convention which must be you must observe. Failure to observe such conventions is a common source of error. Third, *PROT_SEMA* has been forced into static storage because it appears in a DATA statement. Had this not been done, each thread would have had its own copy of *PROT_SEMA* and it would not have functioned as a semaphore between threads. Failure to force semaphores into static memory is another common error.

Finally, *PROT_SEMA* is local to this subroutine while the data it protects (the global variable *PROT*) is accessible by any procedure in the program. If the only possible accessing conflicts on *PROT* appear in this routine, then this setup is suitable. But if other routines might try to access *PROT* during parallel execution, an error may occur. To solve the problem, then, either *PROT_SEMA* should be global (that is, as a global variable) or *PROT* should be threadlocal.

Restriction On Library Access (Fortran or C)

Because of internal state, library functions which do string operations, system calls, or Input/Output (I/O) can not be called simultaneously by multiple threads. If parallel threads are going to call such functions, all such calls should be protected by a semaphore. Remember that in Fortran, I/O statements and operations like // which manipulate CHARACTER*N* variables call such library functions implicitly on behalf of the program.

NOTE
The general term *resource* is often used to cover both critical data items and critical procedures; both may need to be protected from parallel access.

Creating A Fortran Parallel Procedure

Once a parallel Fortran program has been written, there are two ways to compile it.

- The directive **C$DOIT PPROC** *Name* may appear in each routine *Name* which should be compiled for parallel execution.

- Specify **–safe=procs** on the compiler command line when compiling procedures that are to run in parallel. This tells the compiler that you have taken care of all parallel processing issues. If you should still have reservations about whether certain procedures have been compiled for parallel

operation, you can specify **C$DOIT ASIS** ahead of any procedure or loop which you don't wish to run in parallel.

Using either of these two methods compiles the procedure for parallel execution, regardless of optimization level or other options. This means:

1. The compiler moves all local variables to dynamic storage (with exception of variables named in EQUIVALENCE statements, NAMELISTS, and variables initialized by DATA statements). The compiler warns about local variables kept in static storage.

2. The compiler does not generate any parallel loops within the procedure, regardless of optimization level. In other words, optimization level O3 behaves just as level O2 within the procedure. (Something that is already running in parallel cannot be further parallelized.)

If the compiler is told that a procedure may be called from a thread of a parallel program, it assumes the following:

1. Any procedures called within the parallel subroutine have also been compiled for parallel execution. The compiler enforces no runtime check on this; it is entirely your responsibility. If you forget to compile a procedure for parallel execution, you will get unreproducible results.

2. You have provided semaphore protection around I/O routines and system calls. In particular, string manipulation and assignment in Fortran involve library calls to obtain storage; these are not protected by the system. If you do any string manipulation within a parallel procedure, the protection is your responsibility. Intrinsic math routines have all been compiled for parallel execution so you do not need to worry about them.

3. All entry points are intended to be compiled for parallel execution. Separate entry points into parallel code are acceptable; however if different entries in the same function return different types of values (that is, one returns a REAL and another a REAL*8), the Fortran compiler must create static variables to handle this correctly. As a result, such entries may have unpredictable results. The compiler prints warning messages in such cases; the variables named have the form *return_xxx* where *xxx* is some name obviously related to the function or entry name.

4. The compiler assumes that no %VAL parameters are used. The setup code for parallel procedures assumes that all parameters are passed by reference.

5. Access to global locations is either read-only or semaphore protected.

While the restrictions sound fairly difficult, they arise infrequently in practice. Converting a code for parallel execution typically requires you to make only a few code modifications.

As a final warning, once the program has entered a section of parallel code, the program inherits the state of the system at that time. This is particularly important, for example, if files must be opened and the file descriptor must be accessible to all processors. If it should become necessary to open a file sometime after the first parallel procedure is executed, the function **MT_SERIAL()** can be called to force a return to serial execution, the file(s) can be opened, and additional parallel program code can be executed, returning the system to parallel execution. The syntax of this call is:

NOTE
Pay attention to the state of the system at the time that the code goes parallel, to assure that the appropriate state of the system is made available to the parallel threads.

```
CALL MT_SERIAL()
```

As an alternative to using this function, you can simply be certain that the state of the system (as regards opened files and other globally accessible data) is set before the first parallel procedure is encountered.

Calling A Parallel Fortran Procedure

Calling a parallel procedure is extremely straightforward. The *PPROC* directive used in a calling routine tells the compiler that a procedure is safe for parallel execution. As a result, the compiler considers any loop that contains a call to such a procedure to be a valid candidate for parallelization, and parallelizes that loop naturally. It may be necessary to add an IPDEP directive (ignore dependencies that would otherwise prevent parallelization) before the loop to get it to parallelize, but nothing else should be necessary.

To illustrate, consider the following loop

```
DO I = 1, N
  CALL setup(I)
  a(I) = compute(I)
ENDDO
```

NOTE
The compiler does not check that **setup** or **compute** have been compiled for parallel execution: it accepts your word for it.

To enable the compiler to parallelize this loop, it must be told that both *setup* and *compute* have been compiled for parallel execution. (Of course, you do not get consistent results if these function cannot run in parallel.) But without the directive, the loop cannot be parallelized, even though **-O3** is specified. This happens because the compiler assumes that function calls within the loop must, of necessity, prevent parallelization.

```
C$DOIT PPROC setup, compute
      DO I = 1, N
        CALL setup(I)
        a(I) = compute(I)
      ENDDO
```

The compiler expects all variables to be pass-by-reference; incorrect code may be generated if %VAL is used. In spite of this, the compiler provides some protection regarding scalar variables. A scalar variable passed down to a parallel procedure is copied into a threadlocal variable, and the address of the threadlocal variable is passed, rather than the address of the scalar itself. While this means that a parallel procedure cannot modify a passed–in scalar (in the above procedure, neither *setup* nor *compute* is able to modify *i*), it protects users from common parallel programming mistakes. Without this protection, the example loop above would be incorrect.

Parallel procedures can be called with any number of arguments and return any type of value. Because of the library calls necessary for strings, however, it is wise to avoid string functions.

A Fortran Parallel Processing Example

To illustrate the functions and methods discussed above, consider the following example:

```
COMMON /X/ J,K,L
DOUBLE PRECISION Y(100,100,100)
L = 1
DO I = 1, 100
  CALL MYFUNC (Y,I)
ENDDO
...
SUBROUTINE MYFUNC(Y,I)
DOUBLE PRECISION Y (100,100,100)
COMMON /X/ J,K,L
```

```
      COMMON /Z/ IJ
      IJ = I - 1
      DO J = 1, N
        DO K = 1, N
                IF (L .EQ. 1) CALL COMPUTE1(Y)
                CALL COMPUTE2(Y)
        ENDDO
      ENDDO
      END
```

COMPUTE1 and *COMPUTE2* perform some computations based on *J* and *K* and store the results into *Y(I,J,K)*. Assume that these computations depend on values in the *J* and *K* dimensions in such a way that these loops cannot be parallelized, but that the *I* loop can. *COMPUTE1* and *COMPUTE2* use the values in COMMON block *X* and the *IJ* value in COMMON block *Z* to know which element *Y* to compute. Now, parallelize the *I* loop in the main routine.

Storage Issues

The values *IJ*, *J*, and *K* are all computed and used within a parallel region, and different processors must use different instantiations (that is, the memory locations assigned to these variable names must be different for each processor.) These variables should be specified as threadlocal storage. *L* is needed inside the parallel region also, but it is computed outside the region, so this variable should not be in threadlocal storage. As the first step to prepare for parallelization, the common block should be split, modifying the program to read:

```
        COMMON /X/ J,K
        COMMON /X1/ L
C$DOIT THREADLOCAL X
        DOUBLE PRECISION Y (100,100,100)
        L = 1
        DO I = 1, N
           CALL MYFUNC (Y,I)
        ENDDO
        ...
        SUBROUTINE MYFUNC (Y,I)
        DOUBLE PRECISION Y (100,100, 100)
        COMMON /X/ J,K
        COMMON /X1/ L
        COMMON /Z/ IJ
C$DOIT THREADLOCAL X,Z
        IJ = I
        DO J = 1, N
           DO K = 1, N
              IF(L .EQ. 0) CALL COMPUTE1(Y)
              CALL COMPUTE2(Y)
```

```
      ENDDO
   ENDDO
   END
```

Separate Workspaces For Each Processor

As specified earlier, the function *MT_INIT* is used to assign separate workspaces (dynamic memory) for each process. Thus the call to this function must be inserted prior to any call that requires parallel operation.

```
      COMMON /X/ I,J, K
      COMMON /X1/ L
C$DOIT THREADLOCAL X
      DOUBLE PRECISION Y (100,100,100)
      CALL MT_INIT(10000)
      ...
      ...
```

Parallel Procedure Specification

Because there is a function call in a loop, the compiler must be told that parallelization is desired. The way to accomplish this is to use directives to inform the compiler that *MYFUNC* is to be compiled for parallel execution.

```
      COMMON /X/ I,J, K
      COMMON /X1/ L
C$DOIT THREADLOCAL X
      DOUBLE PRECISION Y (100,100,100)
      CALL MT_INIT(10000)
      L = 1
C$DOIT PPROC MYFUNC
      DO I = 1, N
         CALL MYFUNC(Y,I)
      ENDDO
      ...
      SUBROUTINE MYFUNC(Y,I)
      DOUBLE PRECISION Y (100,100,100)
      COMMON /X/ J,K
      COMMON /X1/ L
      COMMON /Z/ IJ
C$DOIT THREADLOCAL X,Z
C$DOIT PPROC MYFUNC
      IJ = I
      DO J = 1, N
         DO K = 1, N
               CALL COMPUTE1(Y)
               CALL COMPUTE2(Y)
         ENDDO
      ENDDO
      END
```

Note that the PPROC directive applies throughout the first routine. As a result, if there are other loops in the first

routine that call *MYFUNC*, they are also considered for parallel execution.

Notice that there are two PPROC directives in the code sample. The first use of PPROC tells the compiler that the function *MYFUNC* has been compiled to be save for parallel execution. The second use of this directive explicitly tells the compiler (inside the function itself) that it should indeed be compiled to be safe for parallel execution.

Also note that both COMPUTE1 and COMPUTE2 must be compiled either with PPROC or **–safe=procs**. The compiler assumes this. If you don't, it's an error.

Examining The VREPORT

At this point, the work is almost done. Compile the procedure using options **-O3 -vreport** and examine the vector report. A parallel loop has been generated around the call to *MY_FUNC*. The vector report may have additional suggestions about other possible modifications and options. For example, occasionally you may have to put an IPDEP directive around the loop when the vectorizer detects dependencies that may cause problems, but the IPDEP (ignore dependencies) directive use should be sufficient and the code should not require further modifications.

Note that this parallel program works only because the Stardent 1500/3000 compiler generates temporaries for scalars. If it did not, then one processor might overwrite the value of *I* needed by another processor before it gets a chance to read it.

Other Considerations

As an example, consider modifying this program to print out each time it completed a call to *MYFUNC*. This is easily done by adding in a semaphore to block on I/O.

```
        COMMON /X/ J,K
        COMMON /X1/ L
C$DOIT THREADLOCAL X
        DOUBLE PRECISION Y(100, 100, 100)
        CALL MT_INIT (10000)
        L = 1
C$DOIT PPROC MYFUNC
        DO I = 1, N
           CALL MYFUNC(Y,I)
        ENDDO
        ...
        SUBROUTINE MYFUNC(Y,I)
```

```
          DOUBLE PRECISION Y(100,100,100)
          COMMON /X/ J,K
          COMMON /X1/ L
          COMMON /Z/ IJ
          DATA ISEM/1/
C$DOIT THREADLOCAL X,Z
C$DOIT PPROC MYFUNC
          IJ = I
          DO J = 1, N
             DO K = 1, N
               IF(L .EQ. 0) CALL COMPUTE1(Y)
               CALL COMPUTE2(Y)
             ENDDO
          ENDDO
          CALL MT_LOCK(ISEM)
          PRINT*, 'Finished iteration',I
          CALL MT_UNLOCK()
          END
```

The DATA statement forces the semaphore into static storage. This is required to allow all threads access to the same memory location. All threads must lock on the same memory location.

If you run this program, you notice that the iteration order differs from that which you might expect. In general, you should not use semaphores that depend upon the iteration order of parallel loops or you may deadlock your program.

Summary of Important Points For Fortran Parallel Programming

1. The PPROC directive declares a parallel procedure, and tells the compiler to compile a procedure for parallel execution. With it, all local variables except for equivalenced and initialized variables go into stack storage.

2. The option **-safe=procs** can be used to compile a parallel procedure.

3. The THREADLOCAL directive creates copies of COMMON blocks for each thread; STATIC creates static, non-external storage.

4. Only pass-by-reference is allowed into parallel procedures. Scalar variables cannot be modified by the parallel procedure.

5. *MT_INIT* must be called before any parallel call is made. The stack size is difficult to guess accurately, but a value of 100,000 covers most programs that do not have local arrays.

6. String operations and I/O must be protected with semaphore operations using *MT_LOCK* and *MT_UNLOCK*. Semaphores must be initialized to 1 by the programmer and must be globally available.

Miscellaneous Functions

Near the beginning of a program, you may want to discover how many processors are actually installed in a system. The function to provide this information is:

```
N = MT_NUMBER_OF_PROCS()
```

There is no equivalent function provided in C.

Then, if there are, for example, three processors in the system, you may wish to explicitly use only two processors for running the threads. For this the following function is provided:

```
CALL MT_SET_THREAD_NUMBER(no_of_threads, procs)
```

procs is an array of integers (each an INTEGER*4) that gives explicit processor numbers that are to be used to run various threads. Each entry in this array corresponds to a processor number and ranges in value from 0 to 3 (0, 1, 2, or 3). As threads are created, they are assigned to the specific processors as listed in this array. For example, if there are three threads to be created, and the first two threads are to run on processor one, and the third thread is to run on processor zero, the first three elements of the array would contain the values 1, 1, and 0 respectively; *num_of_threads* defines how many elements there are in the array. Only the first *num_of_threads* entries are examined.

Operating System Implementation Highlights

The following notes may be of interest to the writer of parallel processing code for the Stardent 1500/3000.

Parallel Processes

If you ask for a particular number of processors to run your threads, because of the priority that the operating system places on running tasks in parallel where possible, there is a very good chance that scheduling for parallelism occurs as expected. On rare occasions, the UNIX *kernel* could be busy doing some form of bookkeeping or running other tasks, however, and be unable to grant that number of processors. Again, expect this to be the

exceptional case.

As an example of processor availability versus thread assignment, assume that your program encounters a parallel section of code and asks for two threads (thus asking for two processors). Two processors are installed in the system and one is busy when this code section is entered. The available processor begins one thread. If the other processor continues to be busy and the processor running your first thread becomes available, the operating system redispatches the available processor to run the second thread. Thus, your program may not necessarily run both threads in parallel as expected, depending on what is going on in your system when the parallel code segment is entered.

Creating A C Parallel Procedure

Once a parallel C program has been written, there are two ways that it can be compiled. These are:

- You can specify **C#pragma PPROC** as a directive for each named procedure for which the parallelization should be performed. Using this method compiles the procedure for parallel execution, regardless of optimization level or other options. This causes the compiler not to generate any parallel loops within the procedure, regardless of optimization level. In other words, optimization level O3 behaves just as level O2 within the procedure. (Something that is already running in parallel cannot be further parallelized.)

  There is no need to adjust local variables since C locals are allocated on the stack by default.

- If the compiler is told that a procedure may be called from one of several parallel execution threads, it assumes the following:

  1. Any procedures called within the parallel subroutine have also been compiled for parallel execution. The compiler enforces no runtime check on this; it is entirely your responsibility. If you forget to compile a procedure for parallel execution, you obtain unreproducible results.

  2. The user has provided semaphore protection around I/O routines and system calls. Most routines in the C library use only stack storage and should be safe to call from a parallelized function. However, there are a few functions that use static storage (*malloc* is an obvious example) that cannot be

called in parallel without appropriate synchronization. If you are at all in doubt, or if you get unreproducible results, use synchronization.

3. Global locations are accessed either as read-only or are semaphore protected.

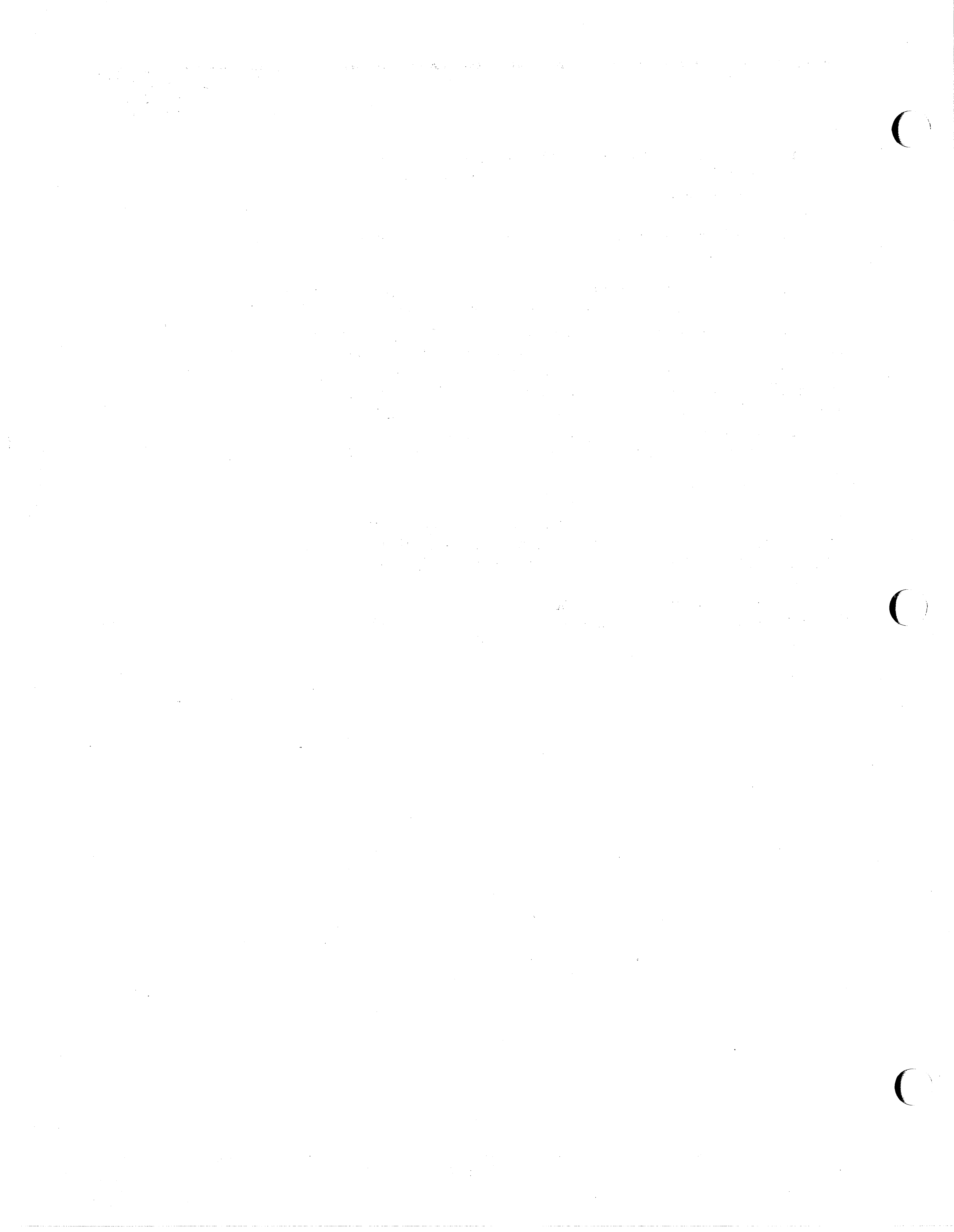
As a final warning, once the program has entered a section of parallel code, the program inherits the state of the system at that time. This is particularly important, for example, if files must be opened and the file descriptor must be accessible to all processors. If it should become necessary to open a file sometime after the first parallel procedure is executed, the function *MT_SERIAL()* can be called to force a return to serial execution, the file(s) can be opened, and additional parallel program code can be executed, returning the system to parallel execution. The syntax of this call is:

```
MT_SERIAL();
```

NOTE
Pay attention to the state of the system at the time that the code goes parallel.

As an alternative to using this function, you can simply be certain that the state of the system (regarding opened files and other globally accessible data) is set before the first parallel procedure is encountered.

Declarations for all C microtasking functions are located in */usr/include/micro.h*.



TUNING CODE

CHAPTER NINE

This chapter covers two topics that allow you to tune your code for maximum performance

- The Stardent profiler is described, allowing you to determine where your code is spending the most time, and therefore allowing you to rewrite inefficient sections if possible.
- Compiler directives are described. The compiler sometimes cannot determine whether it is safe to vectorize or parallelize a section of code. As a result, the compiler sometimes may not necessarily make the most efficient choice of which loops to vectorize or parallelize. You can use compiler directives to tell the compiler the information that it needs to make these decisions.

Profiling Programs

To speed up a program, it is first necessary to determine which parts of the program are taking the most execution time. While estimates of execution time can be done at compile time (and are done in the **vreport** facility), the most accurate way of determining execution time is to create an *execution profile* of your program. The Stardent 1500/3000 Fortran and C compilers provide a number of tools for very accurately determining the execution time of various parts of your program.

The basis of the Stardent 1500/3000 profiling utilities is the Unix program *prof*. However, unlike most other Unix systems, it is not necessary to compile your program with special options in order to obtain profiling information. Instead, the Stardent 1500/3000 contains a special program *mkprof* which when run on an existing *a.out*, creates a new executable program that generates profiling information when the program is run. As a result, profiling on a Stardent 1500/3000 is a very simple process that you will probably want to use often to tune your code.

For users who are accustomed to profiling programs on other systems, the following is a brief overview of the profiling process on the Stardent 1500/3000:

(1) Compile and link your program normally to create an executable file (assumed to be named *a.out*). It is not necessary to specify any special compiler options.

(2) Run *mkprof* on the executable file to create a new executable file that generates profiling information. The *mkprof* command takes two arguments: the name of the executable that is to be profiled (default *a.out*) and the name of the new executable to be created that generates profiling information (default *x.out*). Thus, a typical command at this stage might be the following:

```
mkprof a.out mysample.out
```

(1) Run the newly created executable file. *mysample.out* This executable file, when run, creates a file *mon.out* that contains very raw profiling data.

(2) Finally, use *prof* to convert the raw data in *mon.out* into user-readable form. The *prof* command takes two arguments: the name of the executable which generated the profiling information (default *x.out*) and the name of the file containing raw profiling data (default *mon.out*). Thus, the final step in the example sequence would be

```
prof mysample.out
```

This prints a listing of statistics from the program.

This chapter provides more details on profiling in the Stardent 1500/3000 compilers, including use of the **-ploop** option to obtain loop level profiling and use of special options to use different timers in the profiling process. Additionally, it shows how modifying a program changes the execution profile as an example of the possible uses of this technique.

Note that although the examples concentrate on Stardent 1500/3000 Fortran, the identical profiling facilities are also available in C.

Output Description

The first step in using the *prof* program is understanding its output.

On one processor systems or on programs which have not been compiled for parallelism, the **-v** (verbose) option can provide more useful information. Following is the output of *prof -v* on the same program as above, but compiled using the an optimization level of **-O2** rather than **-O3**:

```
count   microsec.     %   time/call   NAME (1 proc.)
10000     3772320  51.2         377   ADDLOOP
10000     3551423  48.2         355   SEPARATE
    1       39725   0.5       39725   _fort_program
```

Once again, the information is presented in the order of highest usage. The first column, count, represents the number of times the routine is entered. The second column, microseconds, is exactly that, the total number of microseconds that were spent in the routine. The third column, %, is the percentage of total time that was spent in the routine. The fourth column, time/call, is the number of microseconds it took to execute each call to the routine. The fifth column is the name of the procedure.

Following is additional sample output from the *prof* command. In this case, the program was compiled with an optimization level of **-O3** so that multiple processors will be used. Two processors were in the test machine, thus causing a two-column output.

```
count   microsec.       count   microsec.  NAME (2 proc.)
10000     2969734           0     2685734  SEPARATE
10000     2879429           0     2630392  ADDLOOP
    1       43869           0         298  _fort_program
```

As is evident from the information after the NAME column, this program has been run on a two processor Stardent 1500/3000. The output of *prof* is subroutine-oriented; thus, each row in the output represents information involving the time spent in the specific subroutine indicated in the rightmost column. For each processor, the "count" and "microsec" columns are used by the program to indicate how many times a particular subroutine has been called by each processor and the amount of time spent of that routine by that processor.

Thus, in the example above, the subroutine SEPARATE was called 10,000 times on processor 1 and processor 1 spent roughly 3 seconds of execution time working in SEPARATE. Similarly,

SEPARATE is never called directly on processor 2, but processor 2 does spend roughly 2.7 seconds executing SEPARATE, indicating that SEPARATE consists mostly of parallel loops which load balance well. From the output, it is easy to see that both routines SEPARATE and ADDLOOP parallelize well, and that _fort_program (which is the name given by the compiler to Fortran main programs) has a very small amount of parallelism in it. Note that the output is sorted in order of the routines requiring the most time on the first processor.

This format allows you to estimate the importance of various speedups. For instance, if you think you can cut the execution time of ADDLOOP in half, you will speed up the execution time of the whole program by roughly 25% (51.2% divided by 2).

-ploop Option

As you can tell from the sample output above, using *mkprof* and *prof* with default compiler output can tell you which subroutines are taking the most time, but do *not* indicate which parts of the subroutines are consuming the time. In large subroutines that contain many loops, it is not always easy to determine which loop nests are the important ones on which to focus. The Stardent 1500/3000 Fortran and C compilers contain an option **-ploop** to reduce the granularity reported by *prof*.

When a source program is compiled with the **-ploop** option enabled, regardless of optimization level, the compiler inserts labels before and after each loop nest in the program, making that loop nest appear to be a separate subroutine to the profiler. The labels have the following form

```
_$lp_InLoop_ . . .
```

and

```
_$lp_AfterLoop_ . . .
```

The . . . indicates a label that identifies the source routine and the line number of the loop within that routine. When *prof* is invoked on the resulting code, it indicates a subroutine `_$lp_InLoop_` that holds the execution time spent in that loop, and another subroutine `_$lp_AfterLoop_` that holds the execution time of the code that comes after that loop and before the next loop.

To illustrate, consider the following Fortran program:

```
(the program is named zzz.f)

      program main
      double precision a(100,100), b(100,100), c(100,100)
      data n/100/, m/100/
      data b/10000 * 1.0d0/, c/10000 * 1.0d0/
      if (m .eq. n) then
         do i = 1, n
            do j = 1, n
               a(i,j) = a(i,j) * b(i,j)
            enddo
         enddo
      else
         do i = 1, n
            do j = 1, m
               a(i,j) = a(i,j) / b(i,j) / c(i,j)
            enddo
         enddo
      endif
      if (m .ne. n) then
         do i = 1, n
            do j = 1, n
               a(i,j) = a(i,j) * b(i,j)
            enddo
         enddo
      else
         do i = 1, n
            do j = 1, m
               a(i,j) = a(i,j) / b(i,j) / c(i,j)
            enddo
         enddo
      endif
      end
```

The following four commands were issued on the command line to compile the program, create the executable for profiling, execute the program and create the profile file, and run the profiler.

```
fc -02 -ploop zzz.f
mkprof a.out zzz.out
zzz.out
prof -v zzz.out
```

The preceeding statements are explained as follows:

1. *fc -02 -ploop zzz.f*

   Compile the Fortran file *zzz.f* at optimization level **-02** and profile loops within each routine. The optimization level can be **-O3** (or any other) if you wish. The output from this compilation is *a.out*.

2. *mkprof a.out zzz.out*

   Cause the *mkprof* to profile the object file *a.out* (obtained from step #1) and store the result in *zzz.out*. You may choose any name you wish for the final executable file.

3. *zzz.out*

   Run the executable file *zzz.out*. This run creates the *mon.out* needed by prof.

4. *prof -v zzz.out*

   Cause the command *prof* to output the statistics of profiling.

The following is the output produced by the *prof* command on *zzz.out* from an executable run on a Stardent 1500/3000 P2 processor:

```
count   microsec.    %   time/call    NAME (1 proc.)
    1       38769  89.2      38769    _$lp_InLoop_main_program_25
    1        1991   4.6       1991    _$lp_InLoop_main_program_6
    1         569   1.3        569    _IO_Initialize_FORTRAN_IO
   31         427   1.0         13    sigset
    3         373   0.9        124    malloc
    3         285   0.7         95    _IO_Close
    3         135   0.3         45    _IO_Allocate_Unit
    3         134   0.3         44    calloc
    1         100   0.2        100    _IO_Finish_FORTRAN_IO
    6          84   0.2         14    _IO_Find_Unit
    3          78   0.2         26    _IO_Can_This_File_Seek
    2          74   0.2         37    sbrk
    3          68   0.2         22    ioctl
    3          65   0.1         21    isatty
    4          61   0.1         15    _IO_Unit_Walk
    3          60   0.1         20    fstat
    3          55   0.1         18    _IO_Change_Unit_Number
    1          40   0.1         40    _IO_Initialize_Unit_Walk
    2          40   0.1         20    fflush
    1          31   0.1         31    _start
    1          17   0.0         17    _fort_program
    1          11   0.0         11    _$lp_AfterLoop_main_program_6
    1           6   0.0          6    _$lp_AfterLoop_main_program_25
    1           0   0.0          0    exit
```

Since Stardent 1500/3000 P2 processors do not have a hardware vector divide, the results are pretty much what one would expect: the divide loop at line 25 (_$lp_InLoop_main_program_25)dominates the execution time. Far behind it is the vector multiply loop at line 6 (_$lp_InLoop_main_program_6). The serial code after the loop at

line 6 (_$lp_AfterLoop_main_program_6) and after the loop at line 25 (_$lp_AfterLoop_main_program_25) take almost no time at all. The vector loops completely dominate the execution. The loops at line 12 and line 19 are not executed.

Note that **-ploop** is only a compile-time option that causes extra labels to be inserted in the generated code. It has no effect at link time, and with the exception of the generated labels, has no effect on the efficiency of the code generated by the compiler. The compiler option only generates labels around the outermost loop nests (where loops are defined as **DO** and **DO WHILE** statements in Fortran); at the moment, it is unable to profile any inner loops separately.

-p Option

On many Unix systems, profiling is accomplished by compiling a program with **-p** to cause the compiler to insert profiling code. The Stardent 1500/3000 compilers accept the **-p** option, but most users will generally not care to use it. For compiles, specifying **-p** on the command line has no effect; the compiler accepts it but ignores it. For links, **-p** causes *mkprof* to be run on the executable file that is created, thereby making an executable that contains profiling code. Because *mkprof* is so easy to use on its own, you will probably not need to use **-p** on link lines to obtain profiling.

Interpreting Profiled Programs

Once you have found that the loop at line 25 consumes the majority of the execution time in the previous example, the next question to ask is whether that loop can be rewritten to run faster. The loop runs so slowly because Stardent 1500/3000 P2 does not have vector hardware divides, and the scalar hardware divide it has is one of its slower instructions. If a way can be found to compute the same thing without a divide, the program can execute more quickly.

With a little thought, it is easy to realize that the loop can be sped up tremendously. Algebraically, (a/b)/c is equal to a/(b*c). Unfortunately, the floating point hardware in Stardent 1500/3000 (and in all other digital computers) does not exactly obey all the laws that real numbers obey. In particular, floating point hardware is not truly associative, and it is also not always true that (a/b)*b == a. Hence, if accuracy to the very last bit is necessary, then the algebraic identity (a/b)/c == a/(b*c) cannot be used. However, if absolute accuracy is not required (and for most

applications, it rarely is), rewriting the code in loop 25 as the following provides a significant speedup.

```
do i = 1, n
   do j = 1, m
      a(i,j) = a(i,j) / (b(i,j) * c(i,j))
   enddo
enddo
```

However, there is an even easier way to get the same speedup. The Stardent 1500/3000 compilers focus heavily on accuracy of results: they are very careful to avoid optimizations that may change the results of a computation. Since many cases do not require absolute accuracy, the compilers have a command-line option that enables optimizations that may involve a small loss of accuracy. This option, **-fast**, can often provide tremendous speedups. For instance, compiling the original source file *zzz.f* with the compile line shown below causes additional optimizations:

```
fc -O2 -fast -ploop zzz.f
```

(1) Using the **-fast** option enables the reassociation described above—(a/b)/c to a/(b*c).

(2) It allows the compiler to approximate vector division in software by computing a vector inverse of (b*c), then multiplying that times a. While this is not exactly equivalent to the division results, it differs in only the last bit, and it is faster than the equivalent sequence of scalar divides.

The results of *-ploop* profiling with the *-fast* option are listed below:

```
count   microsec.     %   time/call    NAME (1 proc.)
 400       24812   78.0        62      _MA_VDDIV
   1        2321    7.3      2321      _$lp_InLoop_main_program_25
   1        2096    6.6      2096      _$lp_InLoop_main_program_6
   1         457    1.4       457      _IO_Initialize_FORTRAN_IO
  31         396    1.2        12      sigset
   3         376    1.2       125      malloc
   3         232    0.7        77      _IO_Close
   3         161    0.5        53      _IO_Allocate_Unit
   3         134    0.4        44      calloc
   3         112    0.4        37      _IO_Can_This_File_Seek
   1          90    0.3        90      _IO_Finish_FORTRAN_IO
   2          88    0.3        44      sbrk
   6          82    0.3        13      _IO_Find_Unit
   3          70    0.2        23      fstat
   3          64    0.2        21      isatty
   3          64    0.2        21      ioctl
   4          55    0.2        13      _IO_Unit_Walk
```

```
3          43   0.1          14    _IO_Change_Unit_Number
1          34   0.1          34    _IO_Initialize_Unit_Walk
2          34   0.1          17    fflush
1          32   0.1          32    _start
1          26   0.1          26    _fort_program
1          14   0.0          14    _$lp_AfterLoop_main_program_25
1           7   0.0           7    _$lp_AfterLoop_main_program_6
1           0   0.0           0    exit
```

You can see that the time for loop 25 has decreased dramatically. Part of this time is hidden; since the vector division is done by a subroutine (_MA_VDDIV), much of the time that was originally in the loop itself is now done within the subroutine. However, taking the time for the division subroutine (24,812) and the time for the loop itself (2,321) together gives 27,133. This still compares very favorably to the original time of 38,769.

The most useful aspect of the profiler, particularly when combined with **-ploop**, is to indicate which loops require the most attention. Quite often, the slowest loops are ones which can profit by **-fast** as above, or which can be simply rewritten to better utilize the Stardent 1500/3000 vector hardware. The **-vreport** option indicates which operations have to be done in scalar mode, which is the most common cause of slow loops.

Other Timing Options for mkprof

The *mkprof* command has two options that you can specify to control the timing intervals for profiling.

- -s*N*; this option selects the scaling factor. The default for *N* is 4, giving microsecond granularity.

  Larger numbers allow longer times with less accuracy; each increase of *N* by 1 doubles the maximum range and halves the accuracy. If you are profiling long programs (more than two minutes), you might want to use this option.

- -T*letter*, the timing option where *letter* can be *f*, *c*, or *e*, in either upper or lower case. If *letter* is *f*, *c*, or *e*, then the Floating Point Unit (FPU) time (the default setting), the Integer Processing Unit (IPU) time, or the elapsed time is used, respectively. IPU time and elapsed time are always measured in ticks (hundredths of a second). The **-s** option is ignored when either -T*c* or -T*e* is specified.

For most programs, the default of FPU time is the best timer to use. Its granularity is roughly 60 nanoseconds in P2 and 30 nanoseconds in P3, which is far more precise than any other timer. For programs that execute very quickly, such as running the Linpack benchmark on a 100x100 matrix (which takes around 0.08 seconds on P2), the only possible way to obtain reproducible results is with FPU time.

FPU time does not accurately measure programs that have very large data areas accessed in vector mode, because it does not capture some of the operating system overhead involved in these operations. CPU time is a better measure in those cases. For instance, the Linpack benchmark on a 1000x1000 matrix is not very accurately measured by FPU time; CPU time is a more reproducible measure for it. Elapsed time is generally useful only on standalone systems where a user is willing to make significant I/O changes.

Please refer to the *Commands Reference Manual* for additional information.

Compiler Directives

Restructuring compilers determine at compile-time the optimal run-time use of parallel and vector hardware. The compiler cannot have full knowledge at compile-time of important run-time information, so in some cases it may be desireable to supply compiler directives which override the compiler's natural caution. Use the compiler option **-vreport** to help you decide if you need to supply a compiler directive. Directives influence all phases of compilation and are essential for optimal use of vector and parallel hardware.

The Stardent 1500/3000 directives have been designed to look like comments to other compilers, and the Stardent 1500/3000 compiler does not attempt to issue any error messages regarding directives (since something that looks similar to a Stardent 1500/3000 directive may simply be a comment). As a result, it is often useful to examine the vreport, where all directives found are listed, to ensure that a typographic error has not caused the compiler to overlook an intended directive.

Through compiler directives, you help the compiler to generate more efficient code by doing the following

- forcing vectorization or parallelization of a particular loop

- indicating the best loops to vectorize or parallelize
- requesting that a loop be executed exactly as written
- requesting that a procedure be in-lined within a loop
- declaring a procedure that can take vectors as arguments

The following four points summarize the information supplied by the directives to the compiler.

- *Symbolic array references.* The compiler cannot always accurately compute dependencies. You can provide an **IVDEP** or **IPDEP** directive to help the compiler out.
- *Best loop selection.* The compiler cannot always know loop lengths at compile-time. You can provide **PBEST** or **VBEST** directives to help the compiler in its selection of the best loop.
- *Stability or time considerations.* A vectorizer may spend a lot of time analyzing a loop which you do not want vectorized for stability reasons. It may also spend a lot of time analyzing a loop which can not be vectorized for dependency reasons. You can provide an **ASIS** directive to make the compiler bypass the loop.
- *Procedure calls.* Procedure calls hide information from the compiler and prevent it from vectorizing or parallelizing a loop. You can provide vector calculation routines and the **VPROC** directive or ask that procedures be in-lined (**INLINE**) to help loops containing procedure calls.

The following sections discuss the formats of the Fortran and C compiler directives. All examples and syntax are expressed in Fortran notation; however it is important to note the syntax for the directives for C programs. The syntax for the directives for use in C programs is to replace the **C$DOIT** before the directive with **#pragma**. As an example, the **ASIS** directive for C programs becomes

```
#pragma ASIS
```

ASIS

C$DOIT ASIS

The **ASIS** directive tells the compiler not to optimize the following loop for vector hardware. The vectorizer leaves an **ASIS** loop alone, not even applying transformations such as induction variable substitution. This directive can be used for loops with very sensitive numeric properties. Note that this directive orders the vectorizer to leave all statements in the following loop alone, and not change them in anyway. As a result, this directive inhibits vectorization of all loops outside of the controlled loop, as well as those inside the loop.

To illustrate using the **ASIS** directive, consider summing a vector that contains very large and very small values alternated to avoid overflow. The sum reduction hardware in the Stardent 1500/3000 very nearly preserves the associativity of original operations, but not quite. Hence, summing this vector on the reduction hardware may change the results. To prevent the Stardent 1500/3000 vectorizer from using the reduction hardware, place an **ASIS** directive around the loop.

```
C$DOIT ASIS
      DO 160 I = 1, N
        T = T + A(I)
  160 CONTINUE
```

INLINE

The **INLINE** directive tells the compiler to in-line the named functions in the following loops, if possible. Its format is as follows

C$DOIT INLINE *function1, function2, ...*

Use the **INLINE** directive to remove function calls from important loops, thereby permitting the loops to be vectorized. This directive must appear at the place where you want to inline the function, not in the function to be inlined.

IVDEP

C$DOIT IVDEP

Upon encountering the **IVDEP** directive, the Stardent 1500/3000 compiler ignores any array dependencies. Any defined scalars are either in a straightforward recurrence or need to be expanded into temporaries and the compiler decides which.

Note that the **IVDEP** directive does not guarantee vector execution. The Stardent 1500/3000 compiler applies the directive only to array dependencies and assumes that the compiler correctly calculates scalar dependencies. Thus, scalars may inhibit vector execution even when an **IVDEP** directive is present.

In many cases, a loop can be vectorized even though it appears to the compiler that the loop cannot be vectorized. Consider this common loop in relaxation codes.

```
      DO 130 I = 1, N
        DO 131 J = 1, N
          A(I,J) = (A(I,J-1)+A(I,J+1)+A(I-1,J)+A(I+1,J))/4.0
131     CONTINUE
130 CONTINUE
```

Even though this loop computes slightly different results if executed in the vector unit, the differences are negligible in these loops' contexts. Therefore you want to force vector execution of the J loop for speed. Use the **IVDEP** directive to tell the compiler to ignore dependencies that appear to inhibit vectorization.

```
      DO 130 I = 1, N
C$DOIT IVDEP
        DO 131 J = 1, N
          A(I,J) = (A(I,J-1)+A(I,J+1)+A(I-1,J)+A(I+1,J))/4.0
  131   CONTINUE
  130 CONTINUE
```

IPDEP

C$DOIT IPDEP

The **IPDEP** directive is the analog to the **IVDEP** directive and tells the compiler to ignore the dependencies that appear to inhibit parallelization in the following loop. Any defined scalars are assumed to be temporaries local to each processor.

In the **IVDEP** example, the outer loop is the best candidate for parallelization. Insert an **IPDEP** directive in addition to the **IVDEP** directive as shown below.

```
C$DOIT IPDEP;
      DO 130 I = 1, N
C$DOIT IVDEP
        DO 131 J = 1, N
          A(I,J) = (A(I,J-1)+A(I,J+1)+A(I-1,J)+A(I+1,J))/4.0
  131   CONTINUE
  130 CONTINUE
```

VBEST

C$DOIT VBEST

The **VBEST** directive indicates the best loop to vectorize **if the loop is vectorizable.** It does **not** mean that the loop can be vectorized.

Because the compiler does not always know loop lengths at compile-time, it may vectorize a less-than-optimal loop. When presented with the following code and no knowledge of M and N, the compiler assumes the the J loop is the best loop to vectorize because it accesses locations which are adjacent to each other in memory.

```
      DO 140 I = 1, M
        DO 141 J = 1, N
          A(J,I) = B(J,I) + C(J,I)
141     CONTINUE
140 CONTINUE
```

However, if N is equal to 3 and M is equal to 1000, this choice would produce poor execution because the vector loop is too short to make profitable use of the vector unit. In this case, insert a **VBEST** directive, telling the compiler to vectorize the I loop.

```
C$DOIT VBEST
      DO 140 I = 1, M
        DO 141 J = 1, N
          A(J,I) = B(J,I) + C(J,I)
  141   CONTINUE
  140 CONTINUE
```

Now the compiler vectorizes the outer loop (if possible) rather than the inner loop, resulting in more effective operation.

PBEST

C$DOIT PBEST

PBEST is similar to **VBEST**; it indicates the best loop to parallelize rather than to vectorize.

Normally, the Stardent 1500/3000 compiler parallelizes the outermost loop long enough to justify parallel execution. Consider the **VBEST** example.

```
      DO 150 I = 1, M
        DO 151 J = 1, N
          A(J,I) = B(J,I) + C(J,I)
151   CONTINUE
150 CONTINUE
```

With no knowledge of M or N, the compiler parallelizes the outer (I) loop. This is a poor choice if M is only 2 or 3 and N is 1000 and you are running on a four-processor system. In this case, use a **PBEST** directive to signal parallelization of the inner loop.

```
      DO 150 I = 1, M
C$DOIT PBEST
        DO 151 J = 1, N
          A(J,I) = B(J,I) + C(J,I)
151   CONTINUE
150 CONTINUE
```

With the **PBEST** directive, the compiler parallelizes the **151** loop and moves it to the outermost position. This now becomes the following:

```
DO PARALLEL
        J = 1, N
   DO I = 1, M
        A(J,I) = B(J,I) + C(J,I)
```

To summarize, the **VBEST** and **PBEST** directives tell the compiler which of several parts of a nested loop is best to parallelize or vectorize.

The compiler optimizes loops in two phases. First, it chooses the best loop to vectorize. The best loop to vectorize is the loop with the smallest stride and the longest length. If the loop bound is read in at run-time, the compiler must assume a loop length long enough to make vectorization profitable.

Next, the compiler chooses the best loop to parallelize. The best loop to parallelize is the loop with the longest length. Again, if the loop bound is read in at run-time, the compiler must assume a loop length long enough to make parallelization worthwhile.

Use **VBEST/PBEST** directives to keep the compiler from assuming that the longest lengths are those of loops whose bounds are read at run-time.

VPROC

C$DOIT VPROC *fname, vfname*

The **VPROC** directive tells the compiler that the named function has an assembly-language version that can take vector arguments. Upon encountering the **VPROC** directive, the compiler assumes that the named function has no side effects and attempts to vectorize any loop containing a call to that function. If vectorizable, the compiler replaces the function with the vector function name *vfname.*

If no *vfname* is provided, the compiler calls *fname* with vector arguments. Note that using **VPROC** this way is dangerous because you assume that the compiler vectorizes all loops containing this function.

PPROC

C$DOIT PPROC SUBR_NAME

The **PPROC** is a dual-usage directive. The following two examples illustrate how it can be used.

(1) In code to be compiled at optimization level 03, the directive **PPROC** promises to the compiler that a subprogram has been compiled for parallel execution. For example

```
C$DOIT PPROC DOEM
        DO J = 1, 6000
        CALL DOEM(A,B,C,J,K)
        END DO
```

This directive tells that compiler that DOEM has been compiled to be safe for parallel execution. As a result, the compiler parallelizes the DO loop around DOEM.

(1) If the procedure named in the PPROC directive is the same as the procedure being compiled, the compiler interprets the directive as requesting that the procedure be compiled to be safe for parallel execution. As a result, reentrant code is generated for the procedure, just as through the compiler option **-safe=procs** had been specified. For example,

```
          SUBROUTINE DOEM(M,N,X,Y,Z)
C$DOIT PPROC DOEM
                    .
                    .
                    .
```

In this case, the compiler compiles this subroutine for parallel execution. The dual use of the PPROC directive allows users to use one include file across a large number of procedures in many files. This compiler directive takes the form:

C$DOIT PPROC *DOEM*

DOEM is a procedure name, or a list of procedure names.

THREADLOCAL

C$DOIT THREADLOCAL COMMON_NAME

The **THREADLOCAL** directive cause the Fortran compiler to move a named common block into threadlocal storage. Using the **THREADLOCAL** directive on a common block has the equivalent effect as of Cray TASK$COMMON variables. Here is an example:

```
          COMMON /X0/ A,B,C
C$DOIT THREADLOCAL X0
```

X0 is the threadlocal storage holding variables A, B, and C, and these variables are externally known. *Externally known* means that two different procedures can access the variables.

STATIC

C$DOIT STATIC COMMON_NAME

The **STATIC** directive hides the name of the common block so that it is not externally known. Here is an example:

```
          COMMON /X0/ A,B,C
C$DOIT STATIC X0
```

X0 is the static storage holding variables A, B, and C.

VREPORT

C$DOIT VREPORT

The **VREPORT** directive invokes the vector reporting facility on the next loop and is used to tell the user what vectorization has been done to the program. **VREPORT** provides a detailed listing of exactly what the compiler did to each loop nest.

The **VREPORT** directive can only be turned on for one loop at a time and only at the outermost loop level.

NO_PARALLEL

C$DOIT NO_PARALLEL

The **NO_PARALLEL** directive tells the compiler not to run the next loop in parallel.

The following is an example of when you might want to use this directive.

```
        SUBROUTINE  FUZZY(A,M,N)
        DOUBLE PRECISION A(N)
C$DOIT IVDEP
C$DOIT NO_PARALLEL
        DO I = 1, N
          A(I) = A(I) + A(M)
        ENDDO
        END
```

NO_VECTOR

C$DOIT NO_VECTOR

The **NO_VECTOR** directive tells the compiler not to run the next loop in vector. This is useful at optimization level **-O3** when you wish to tell the compiler to parallelize but not to vectorize the code.

SCALAR

C$DOIT SCALAR

The **SCALAR** directive tells the compiler to run the next loop in scalar mode.

This directive has the effect of not allowing vectorization or parallelization on the loop.

OPT_LEVEL

C$DOIT OPT_LEVEL *n*

The **OPT_LEVEL** compiler directive allows you to set the optimization level for a specific procedure within a file.

n is the number 0, 1, 2, or 3. This is the optimization level.

The syntax for using this directive in C the same as Fortran except to replace C$DOIT with #pragma.

This directive should be placed immediately before the first line of the procedure declaration, and overrides any command line options. At the end of the procedure, the optimization reverts to what it was previously.

LANGUAGE INTERFACING

CHAPTER TEN

This chapter contains information that may be useful for system programmers. The topics discussed in this appendix are register sets, floating point computations, the stack frame, calling subprograms, and data layout in memory.

Register Sets

The following section describes the CPU, scalar, and vector register sets. Each processor in the Stardent 1500/3000 system contains its own set of CPU registers, scalar registers, and vector registers. The scalar registers are simply selected members of the vector register set. The use of these registers is described in this chapter.

CPU Registers

Table 10-1 gives the names and functions of the CPU registers.

Table 10-1. CPU Registers

Register	Function
$0	Always 0
$at	Assembler temporary (scratch)
$v0	Function return value (int, pointers)
$v1	Scratch register (caller saves)
$a0–$a3	First four function parameters
$t0–$t7	Scratch registers (caller saves)
$s0–$s7	Register variables (callee saves)
$t8–$t9	Reserved for code generator
$h0–$h1	Reserved for kernel use
$gp	Global pointer
$sp	Stack pointer
$fp	Reserved register (used for profiling)
$ra	Return address

Scalar Registers

Table 10-2 gives the names and functions of the scalar floating point registers.

Table 10-2. Scalar Floating Point Registers

Register	Function
F0	Real/double/complex function return
F1	Complex function return
F2–F3	Scratch registers
F4–F7	First four floating point input arguments
F8–F11	Scratch registers (caller saves)
F12–F15	Register variables (callee saves)
F16–F19	Scratch registers (caller saves)
F20–F23	Register variables (callee saves)
F24–F27	Scratch registers (caller saves)
F28–F31	Register variables (callee saves)

The *F* in the register names in Table 10-2 means the register is holding a single precision floating point quantity. It may be replaced with one of the following builtin names: **D** or **d**, if the register is holding a double-precision floating point quantity; or **I** or **i**, if the register is holding an integer quantity.

In addition, there are four accumulator registers indicated by the builtin name **A** or **a**.

Vector Registers

Vector registers are arranged in four symmetric and interchangeable banks. There are 1024 vector registers available to each user process for working storage. Each bank may be thought of as containing eight vector blocks, 0 through 7, each containing 32 registers, 0 through 31.

Block 0 of each bank (which includes the scalar registers) is reserved for system use. Table 10-3 gives the general scheme for usage of the first 32 vector registers in each bank.

Most vector operations take place on vector blocks, but this is not necessary. Vector operations may begin at any register and

Table 10-3. Vector Registers

Registers	Use
0–7	Scalar registers
8–15	Scratch and constants (used by the compiler)
16–31	Graphics

continue upwards within the same bank. A vector block is designated **V***b.n* and a particular register within that block is designated **V***b.n.m*. A particular register within a bank may be referred to by

The other vector registers are assumed to be scratch registers, although library functions may use other conventions.

Floating Point Computations

The Stardent 1500/3000 architecture permits floating point computations to operate in parallel with the integer processor. Special instructions are used to synchronize the floating point and integer processors at key points in the computation, notably conditional branches and function returns. When a value is computed into a scalar or vector register and then used by a later instruction, the second instruction stalls until the needed value is available. However, because loads from and stores to memory may be proceeding in parallel and take complicated forms, avoiding conflicts in memory is the software's responsibility.

The following summary lists the actions taken by the software to avoid memory conflicts:

- All functions and subroutines must wait for floating point memory stores to finish before returning.
- Functions that return floating point values must complete all floating point memory stores, but may return while the computation of the return value is in progress.
- All loads from the stack must be completed before returning from the function.
- Programs must wait for values to be stored before loading them into integer or vector registers.

The compiler automatically generates code to ensure that these conditions are satisfied.

Table 10-4. FPU Instructions, by Mnemonic

Mnemonic	Parameters	Location	Comments
d1	AC	`0xffff9f00`	Accumulator Op 16
dabs	Dx, Dz	`0xffff9c10`	Scalar 36, Unconditional
dabs,c	Dx, Dz	`0xffff9c90`	Scalar 36, Nullify if StStack<0> clear
dadd	Aw, Dy, Dz	`0xffff9840`	Scalar 64, Unconditional
dadd	Aw, Dz, Aw	`0xffff9440`	Scalar 48, Unconditional
dadd	Dx, Dy, Dz	`0xffff9800`	Scalar 16, Unconditional
dadd	Dx, Dz, Aw	`0xffff9c40`	Scalar 80, Unconditional
dadd,c	Aw, Dy, Dz	`0xffff98c0`	Scalar 64, Nullify if StStack<0> clear
dadd,c	Aw, Dz, Aw	`0xffff94c0`	Scalar 48, Nullify if StStack<0> clear
dadd,c	Dx, Dy, Dz	`0xffff9880`	Scalar 16, Nullify if StStack<0> clear
dadd,c	Dx, Dz, Aw	`0xffff9cc0`	Scalar 80, Nullify if StStack<0> clear
damax	Dx, Dy, Dz	`0xffff9c70`	Scalar 92, Unconditional
damax,c	Dx, Dy, Dz	`0xffff9cf0`	Scalar 92, Nullify if StStack<0> clear
dceq	Dy, Dz	`0xffff9170`	Scalar CMC, Set Mask, Cond=3, Op=4
dcge	Dy, Dz	`0xffff9150`	Scalar CMC, Set Mask, Cond=2, Op=4
dcgt	Dy, Dz	`0xffff9130`	Scalar CMC, Set Mask, Cond=1, Op=4
dcle	Dy, Dz	`0xffff91b0`	Scalar CMC, Set Mask, Cond=5, Op=4
dclt	Dy, Dz	`0xffff91d0`	Scalar CMC, Set Mask, Cond=6, Op=4
dcne	Dy, Dz	`0xffff9190`	Scalar CMC, Set Mask, Cond=4, Op=4
dcun	Dy, Dz	`0xffff9110`	Scalar CMC, Set Mask, Cond=0, Op=4
ddiv	Aw, Dy, Dz	`0xffff9858`	Scalar 70, Unconditional
ddiv	Dx, Dy, Dz	`0xffff9818`	Scalar 22, Unconditional
dhalf	AC	`0xffff9f20`	Accumulator Op 24
dlog10	AC	`0xffff9f38`	Accumulator Op 30
dloge	AC	`0xffff9f30`	Accumulator Op 28
dm1	AC	`0xffff9f08`	Accumulator Op 18
dma	Dx, Dy, Dz, Aw	`0xffff9400`	Scalar 0, Unconditional
dma,c	Dx, Dy, Dz, Aw	`0xffff9480`	Scalar 0, Nullify if StStack<0> clear
dma	Dx, Dy, Aw, Dz	`0xffff9410`	Scalar 4, Unconditional
dma,c	Dx, Dy, Aw, Dz	`0xffff9490`	Scalar 4, Nullify if StStack<0> clear
dmax	Dx, Dy, Dz	`0xffff9428`	Scalar 10, Unconditional
dmax,c	Dx, Dy, Dz	`0xffff94a8`	Scalar 10, Nullify if StStack<0> clear
dmin	Dx, Dy, Dz	`0xffff9420`	Scalar 8, Unconditional
dmin,c	Dx, Dy, Dz	`0xffff94a0`	Scalar 8, Nullify if StStack<0> clear
dmove	Dx, Dz	`0xffff9020`	Scalar CMC, NoSetMsk, Cond=1, Op=0

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
dms	Dx, Dy, Aw, Dz	0xffff9418	Scalar 6, Unconditional
dms	Dx, Dy, Dz, Aw	0xffff9c00	Scalar 32, Unconditional
dms,c	Dx, Dy, Aw, Dz	0xffff9498	Scalar 6, Nullify if StStack<0> clear
dms,c	Dx, Dy, Dz, Aw	0xffff9c80	Scalar 32, Nullify if StStack<0> clear
dmul	Aw, Dy, Dz	0xffff9850	Scalar 68, Unconditional
dmul	Aw, Dz, Aw	0xffff9450	Scalar 52, Unconditional
dmul	Dx, Dy, Dz	0xffff9810	Scalar 20, Unconditional
dmul	Dx, Dz, Aw	0xffff9c50	Scalar 84, Unconditional
dmul,c	Aw, Dy, Dz	0xffff98d0	Scalar 68, Nullify if StStack<0> clear
dmul,c	Aw, Dz, Aw	0xffff94d0	Scalar 52, Nullify if StStack<0> clear
dmul,c	Dx, Dy, Dz	0xffff9890	Scalar 20, Nullify if StStack<0> clear
dmul,c	Dx, Dz, Aw	0xffff9cd0	Scalar 84, Nullify if StStack<0> clear
dneg	Dx, Dz	0xffff9c18	Scalar 38, Unconditional
dneg,c	Dx, Dz	0xffff9c98	Scalar 38, Nullify if StStack<0> clear
dninf	AC	0xffff9f18	Accumulator Op 22
dpinf	AC	0xffff9f10	Accumulator Op 20
dradabs	AC, Vy	0xffff9990	Vec.Reduc. 20, Non-Masked
dradabs,mf	AC, Vy	0xffff99d0	Vec.Reduc. 68, Use /Mask
dradabs,mt	AC, Vy	0xffff9950	Vec.Reduc. 68, Use Mask
dradd	AC, Vy	0xffff9980	Vec.Reduc. 16, Non-Masked
dradd,mf	AC, Vy	0xffff99c0	Vec.Reduc. 64, Use /Mask
dradd,mt	AC, Vy	0xffff9940	Vec.Reduc. 64, Use Mask
drma	AC, Vy, AC, Vz	0xffff9590	Vec.Reduc. 4, Non-Masked
drma	AC, Vy, Vz, AC	0xffff9580	Vec.Reduc. 0, Non-Masked
drma	AC, Vy, [Vz], AC	0xffff9998	Vec.Reduc. 22, Non-Masked
drma,mf	AC, Vy, AC, Vz	0xffff95d0	Vec.Reduc. 52, Use /Mask
drma,mf	AC, Vy, Vz, AC	0xffff95c0	Vec.Reduc. 48, Use /Mask
drma,mf	AC, Vy, [Vz], AC	0xffff99d8	Vec.Reduc. 70, Use /Mask
drma,mt	AC, Vy, AC, Vz	0xffff9550	Vec.Reduc. 52, Use Mask
drma,mt	AC, Vy, Vz, AC	0xffff9540	Vec.Reduc. 48, Use Mask
drma,mt	AC, Vy, [Vz], AC	0xffff9958	Vec.Reduc. 70, Use Mask
drmax	AC, Vy	0xffff95a8	Vec.Reduc. 10, Non-Masked
drmax,mf	AC, Vy	0xffff95e8	Vec.Reduc. 58, Use /Mask
drmax,mt	AC, Vy	0xffff9568	Vec.Reduc. 58, Use Mask
drmin	AC, Vy	0xffff95a0	Vec.Reduc. 8, Non-Masked
drmin,mf	AC, Vy	0xffff95e0	Vec.Reduc. 56, Use /Mask
drmin,mt	AC, Vy	0xffff9560	Vec.Reduc. 56, Use Mask
drms	AC, Vy, AC, Vz	0xffff9598	Vec.Reduc. 6, Non-Masked
drms	AC, Vy, Vz, AC	0xffff9d80	Vec.Reduc. 32, Non-Masked
drms,mf	AC, Vy, AC, Vz	0xffff95d8	Vec.Reduc. 54, Use /Mask
drms,mf	AC, Vy, Vz, AC	0xffff9dc0	Vec.Reduc. 80, Use /Mask
drms,mt	AC, Vy, AC, Vz	0xffff9558	Vec.Reduc. 54, Use Mask
drms,mt	AC, Vy, Vz, AC	0xffff9d40	Vec.Reduc. 80, Use Mask
drmxabs	AC, Vy	0xffff9988	Vec.Reduc. 18, Non-Masked
drmxabs,mf	AC, Vy	0xffff99c8	Vec.Reduc. 66, Use /Mask
drmxabs,mt	AC, Vy	0xffff9948	Vec.Reduc. 66, Use Mask
drpi	AC	0xffff9f28	Accumulator Op 26

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
drsm	AC, AC, Vz, Vy	0xffff9588	Vec.Reduc. 2, Non-Masked
drsm	AC, Vz, Vy, AC	0xffff9d88	Vec.Reduc. 34, Non-Masked
drsm,mf	AC, AC, Vz, Vy	0xffff95c8	Vec.Reduc. 50, Use /Mask
drsm,mf	AC, Vz, Vy, AC	0xffff9dc8	Vec.Reduc. 82, Use /Mask
drsm,mt	AC, AC, Vz, Vy	0xffff9548	Vec.Reduc. 50, Use Mask
drsm,mt	AC, Vz, Vy, AC	0xffff9d48	Vec.Reduc. 82, Use Mask
dsm	Dx, Aw, Dy, Dz	0xffff9408	Scalar 2, Unconditional
dsm	Dx, Dz, Dy, Aw	0xffff9c08	Scalar 34, Unconditional
dsm,c	Dx, Aw, Dy, Dz	0xffff9488	Scalar 2, Nullify if StStack<0> clear
dsm,c	Dx, Dz, Dy, Aw	0xffff9c88	Scalar 34, Nullify if StStack<0> clear
dsub	Aw, Aw, Dz	0xffff9448	Scalar 50, Unconditional
dsub	Aw, Dy, Aw	0xffff9458	Scalar 54, Unconditional
dsub	Aw, Dy, Dz	0xffff9848	Scalar 66, Unconditional
dsub	Dx, Aw, Dz	0xffff9c48	Scalar 82, Unconditional
dsub	Dx, Dy, Aw	0xffff9c58	Scalar 86, Unconditional
dsub	Dx, Dy, Dz	0xffff9808	Scalar 18, Unconditional
dsub,c	Aw, Aw, Dz	0xffff94c8	Scalar 50, Nullify if StStack<0> clear
dsub,c	Aw, Dy, Aw	0xffff94d8	Scalar 54, Nullify if StStack<0> clear
dsub,c	Aw, Dy, Dz	0xffff98c8	Scalar 66, Nullify if StStack<0> clear
dsub,c	Dx, Aw, Dz	0xffff9cc8	Scalar 82, Nullify if StStack<0> clear
dsub,c	Dx, Dy, Aw	0xffff9cd8	Scalar 86, Nullify if StStack<0> clear
dsub,c	Dx, Dy, Dz	0xffff9888	Scalar 18, Nullify if StStack<0> clear
dtdnm	Dz	0xffff9128	Scalar CMC, Set Mask, Cond=1, Op=2
dteq	Dy	0xffff9160	Scalar CMC, Set Mask, Cond=3, Op=0
dtge	Dy	0xffff9140	Scalar CMC, Set Mask, Cond=2, Op=0
dtgt	Dy	0xffff9120	Scalar CMC, Set Mask, Cond=1, Op=0
dtinf	Dz	0xffff9108	Scalar CMC, Set Mask, Cond=0, Op=2
dtle	Dy	0xffff91a0	Scalar CMC, Set Mask, Cond=5, Op=0
dtlt	Dy	0xffff91c0	Scalar CMC, Set Mask, Cond=6, Op=0
dtnan	Dz	0xffff9188	Scalar CMC, Set Mask, Cond=4, Op=2
dtne	Dy	0xffff9180	Scalar CMC, Set Mask, Cond=4, Op=0
dtnrm	Dz	0xffff9148	Scalar CMC, Set Mask, Cond=2, Op=2
dtof	Fx, Dz	0xffff9c2c	Scalar 43, Unconditional
dtof,c	Fx, Dz	0xffff9cac	Scalar 43, Nullify if StStack<0> clear
dtoi	Ix, Dz	0xffff9c30	Scalar 44, Unconditional
dtoi,c	Ix, Dz	0xffff9cb0	Scalar 44, Nullify if StStack<0> clear
dtsgn	Dz	0xffff91e8	Scalar CMC, Set Mask, Cond=7, Op=2
dtun	Dy	0xffff9100	Scalar CMC, Set Mask, Cond=0, Op=0
dtze	Dz	0xffff9168	Scalar CMC, Set Mask, Cond=3, Op=2
duvceq	Vy, Vz	0xffff9270	Vector CMC, Move, Cond=3, Op=4
duvcge	Vy, Vz	0xffff9250	Vector CMC, Move, Cond=2, Op=4
duvcgt	Vy, Vz	0xffff9230	Vector CMC, Move, Cond=1, Op=4
duvcle	Vy, Vz	0xffff92b0	Vector CMC, Move, Cond=5, Op=4

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
duvclt	Vy, Vz	0xffff92d0	Vector CMC, Move, Cond=6, Op=4
duvcne	Vy, Vz	0xffff9290	Vector CMC, Move, Cond=4, Op=4
duvcun	Vy, Vz	0xffff9210	Vector CMC, Move, Cond=0, Op=4
duvdnm	Vz	0xffff9228	Vector CMC, Move, Cond=1, Op=2
duvinf	Vz	0xffff9208	Vector CMC, Move, Cond=0, Op=2
duvnan	Vz	0xffff9288	Vector CMC, Move, Cond=4, Op=2
duvnrm	Vz	0xffff9248	Vector CMC, Move, Cond=2, Op=2
duvsgn	Vz	0xffff92e8	Vector CMC, Move, Cond=7, Op=2
duvze	Vz	0xffff9268	Vector CMC, Move, Cond=3, Op=2
dvabs	Vx, Vz	0xffff9e90	Vec.Dyadic 36, Non-Masked
dvabs,mf	Vx, Vz	0xffff9ed0	Vec.Dyadic 84, Use /Mask
dvabs,mt	Vx, Vz	0xffff9e50	Vec.Dyadic 84, Use Mask
dvadd	Vx, Vy, Vz	0xffff9b80	Vec.Triadic 16, Non-Masked
dvadd	Vx, Vz, Aw	0xffff9680	Vec.Dyadic 0, Non-Masked
dvadd,mf	Vx, Vy, Vz	0xffff9bc0	Vec.Triadic 64, Use /Mask
dvadd,mf	Vx, Vz, Aw	0xffff96c0	Vec.Dyadic 48, Use /Mask
dvadd,mf	Vx, [Vz], Vy	0xffff9ac0	Vec.Dyadic 64, Use /Mask
dvadd,mt	Vx, Vy, Vz	0xffff9b40	Vec.Triadic 64, Use Mask
dvadd,mt	Vx, Vz, Aw	0xffff9640	Vec.Dyadic 48, Use Mask
dvadd,mt	Vx, [Vz], Vy	0xffff9a40	Vec.Dyadic 64, Use Mask
dvam	Vx, Vy, Vz, Aw	0xffff9780	Vec.Triadic 0, Non-Masked
dvceq	Vy, [Vz]	0xffff9360	Vector CMC, Compare, Cond=3, Op=0
dvceq	Vy, Vz	0xffff9370	Vector CMC, Compare, Cond=3, Op=4
dvcge	Vy, [Vz]	0xffff9340	Vector CMC, Compare, Cond=2, Op=0
dvcge	Vy, Vz	0xffff9350	Vector CMC, Compare, Cond=2, Op=4
dvcgt	Vy, [Vz]	0xffff9320	Vector CMC, Compare, Cond=1, Op=0
dvcgt	Vy, Vz	0xffff9330	Vector CMC, Compare, Cond=1, Op=4
dvcle	Vy, [Vz]	0xffff93a0	Vector CMC, Compare, Cond=5, Op=0
dvcle	Vy, Vz	0xffff93b0	Vector CMC, Compare, Cond=5, Op=4
dvclt	Vy, [Vz]	0xffff93c0	Vector CMC, Compare, Cond=6, Op=0
dvclt	Vy, Vz	0xffff93d0	Vector CMC, Compare, Cond=6, Op=4
dvcne	Vy, [Vz]	0xffff9380	Vector CMC, Compare, Cond=4, Op=0
dvcne	Vy, Vz	0xffff9390	Vector CMC, Compare, Cond=4, Op=4
dvcun	Vy, [Vz]	0xffff9300	Vector CMC, Compare, Cond=0, Op=0
dvcun	Vy, Vz	0xffff9310	Vector CMC, Compare, Cond=0, Op=4
dvdnm	Vz	0xffff9328	Vector CMC, Compare, Cond=1, Op=2
dvinf	Vz	0xffff9308	Vector CMC, Compare, Cond=0, Op=2
dvma	Vx, Vy, Aw, Vz	0xffff9790	Vec.Triadic 4, Non-Masked
dvma,mf	Vx, Vy, Aw, Vz	0xffff97d0	Vec.Triadic 52, Use /Mask
dvma,mf	Vx, Vy, Vz, Aw	0xffff97c0	Vec.Triadic 48, Use /Mask
dvma,mt	Vx, Vy, Aw, Vz	0xffff9750	Vec.Triadic 52, Use Mask
dvma,mt	Vx, Vy, Vz, Aw	0xffff9740	Vec.Triadic 48, Use Mask
dvmax	Vx, Vy, Vz	0xffff97a8	Vec.Triadic 10, Non-Masked

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
dvmax	Vx, [Vz], Vy	0xffff96a8	Vec.Dyadic 10, Non-Masked
dvmax,mf	Vx, Vy, Vz	0xffff97e8	Vec.Triadic 58, Use /Mask
dvmax,mf	Vx, [Vz], Vy	0xffff96e8	Vec.Dyadic 58, Use /Mask
dvmax,mt	Vx, Vy, Vz	0xffff9768	Vec.Triadic 58, Use Mask
dvmax,mt	Vx, [Vz], Vy	0xffff9668	Vec.Dyadic 58, Use Mask
dvmin	Vx, Vy, Vz	0xffff97a0	Vec.Triadic 8, Non-Masked
dvmin	Vx, [Vz], Vy	0xffff96a0	Vec.Dyadic 8, Non-Masked
dvmin,mf	Vx, Vy, Vz	0xffff97e0	Vec.Triadic 56, Use /Mask
dvmin,mf	Vx, [Vz], Vy	0xffff96e0	Vec.Dyadic 56, Use /Mask
dvmin,mt	Vx, Vy, Vz	0xffff9760	Vec.Triadic 56, Use Mask
dvmin,mt	Vx, [Vz], Vy	0xffff9660	Vec.Dyadic 56, Use Mask
dvmove	Vx, Vz	0xffff9220	Vector CMC, Move, Cond=1, Op=0
dvms	Vx, Vy, Aw, Vz	0xffff9798	Vec.Triadic 6, Non-Masked
dvms	Vx, Vy, Vz, Aw	0xffff9f80	Vec.Triadic 32, Non-Masked
dvms,mf	Vx, Vy, Aw, Vz	0xffff97d8	Vec.Triadic 54, Use /Mask
dvms,mf	Vx, Vy, Vz, Aw	0xffff9fc0	Vec.Triadic 80, Use /Mask
dvms,mt	Vx, Vy, Aw, Vz	0xffff9758	Vec.Triadic 54, Use Mask
dvms,mt	Vx, Vy, Vz, Aw	0xffff9f40	Vec.Triadic 80, Use Mask
dvmul	Vx, Vy, Vz	0xffff9b90	Vec.Triadic 20, Non-Masked
dvmul	Vx, Vz, Aw	0xffff9690	Vec.Dyadic 4, Non-Masked
dvmul	Vx, [Vz], Vy	0xffff9a90	Vec.Dyadic 20, Non-Masked
dvmul,mf	Vx, Vy, Vz	0xffff9bd0	Vec.Triadic 68, Use /Mask
dvmul,mf	Vx, Vz, Aw	0xffff96d0	Vec.Dyadic 52, Use /Mask
dvmul,mf	Vx, [Vz], Vy	0xffff9ad0	Vec.Dyadic 68, Use /Mask
dvmul,mt	Vx, Vy, Vz	0xffff9b50	Vec.Triadic 68, Use Mask
dvmul,mt	Vx, Vz, Aw	0xffff9650	Vec.Dyadic 52, Use Mask
dvmul,mt	Vx, [Vz], Vy	0xffff9a50	Vec.Dyadic 68, Use Mask
dvnan	Vz	0xffff9388	Vector CMC, Compare, Cond=4, Op=2
dvneg	Vx, Vz	0xffff9e98	Vec.Dyadic 38, Non-Masked
dvneg,mf	Vx, Vz	0xffff9ed8	Vec.Dyadic 86, Use /Mask
dvneg,mt	Vx, Vz	0xffff9e58	Vec.Dyadic 86, Use Mask
dvnrm	Vz	0xffff9348	Vector CMC, Compare, Cond=2, Op=2
dvsgn	Vz	0xffff93e8	Vector CMC, Compare, Cond=7, Op=2
dvsm	Vx, Aw, Vy, Vz	0xffff9788	Vec.Triadic 2, Non-Masked
dvsm	Vx, Vz, Vy, Aw	0xffff9f88	Vec.Triadic 34, Non-Masked
dvsm,mf	Vx, Aw, Vy, Vz	0xffff97c8	Vec.Triadic 50, Use /Mask
dvsm,mf	Vx, Vz, Vy, Aw	0xffff9fc8	Vec.Triadic 82, Use /Mask
dvsm,mt	Vx, Aw, Vy, Vz	0xffff9748	Vec.Triadic 50, Use Mask
dvsm,mt	Vx, Vz, Vy, Aw	0xffff9f48	Vec.Triadic 82, Use Mask
dvsub	Vx, Aw, Vz	0xffff9688	Vec.Dyadic 2, Non-Masked
dvsub	Vx, Vy, Vz	0xffff9b88	Vec.Triadic 18, Non-Masked
dvsub	Vx, Vz, Aw	0xffff9698	Vec.Dyadic 6, Non-Masked
dvsub	Vx, [Vz], Vy	0xffff9a98	Vec.Dyadic 22, Non-Masked
dvsub,mf	Vx, Aw, Vz	0xffff96c8	Vec.Dyadic 50, Use /Mask
dvsub,mf	Vx, Vy, Vz	0xffff9bc8	Vec.Triadic 66, Use /Mask
dvsub,mf	Vx, Vy, [Vz]	0xffff9ac8	Vec.Dyadic 66, Use /Mask

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
dvsub,mf	Vx, Vz, Aw	0xffff96d8	Vec.Dyadic 54, Use /Mask
dvsub,mf	Vx, [Vz], Vy	0xffff9ad8	Vec.Dyadic 70, Use /Mask
dvsub,mt	Vx, Aw, Vz	0xffff9648	Vec.Dyadic 50, Use Mask
dvsub,mt	Vx, Vy, Vz	0xffff9b48	Vec.Triadic 66, Use Mask
dvsub,mt	Vx, Vy, [Vz]	0xffff9a48	Vec.Dyadic 66, Use Mask
dvsub,mt	Vx, Vz, Aw	0xffff9658	Vec.Dyadic 54, Use Mask
dvsub,mt	Vx, [Vz], Vy	0xffff9a58	Vec.Dyadic 70, Use Mask
dvze	Vz	0xffff9368	Vector CMC, Compare, Cond=3, Op=2
f1	AC	0xffff9f04	Accumulator Op 17
fabs	Fx, Fz	0xffff9c14	Scalar 37, Unconditional
fabs,c	Fx, Fz	0xffff9c94	Scalar 37, Nullify if StStack<0> clear
fadd	Aw, Fy, Fz	0xffff9844	Scalar 65, Unconditional
fadd	Aw, Fz, Aw	0xffff9444	Scalar 49, Unconditional
fadd	Fx, Fy, Fz	0xffff9804	Scalar 17, Unconditional
fadd	Fx, Fz, Aw	0xffff9c44	Scalar 81, Unconditional
fadd,c	Aw, Fy, Fz	0xffff98c4	Scalar 65, Nullify if StStack<0> clear
fadd,c	Aw, Fz, Aw	0xffff94c4	Scalar 49, Nullify if StStack<0> clear
fadd,c	Fx, Fy, Fz	0xffff9884	Scalar 17, Nullify if StStack<0> clear
fadd,c	Fx, Fz, Aw	0xffff9cc4	Scalar 81, Nullify if StStack<0> clear
fam	Fx, Fy, Fz, Aw	0xffff9404	Scalar 1, Unconditional
famax	Fx, Fy, Fz	0xffff9c74	Scalar 93, Unconditional
famax,c	Fx, Fy, Fz	0xffff9cf4	Scalar 93, Nullify if StStack<0> clear
fam,c	Fx, Fy, Fz, Aw	0xffff9484	Scalar 1, Nullify if StStack<0> clear
fceq	Fy, Fz	0xffff9174	Scalar CMC, Set Mask, Cond=3, Op=5
fcge	Fy, Fz	0xffff9154	Scalar CMC, Set Mask, Cond=2, Op=5
fcgt	Fy, Fz	0xffff9134	Scalar CMC, Set Mask, Cond=1, Op=5
fcle	Fy, Fz	0xffff91b4	Scalar CMC, Set Mask, Cond=5, Op=5
fclt	Fy, Fz	0xffff91d4	Scalar CMC, Set Mask, Cond=6, Op=5
fcne	Fy, Fz	0xffff9194	Scalar CMC, Set Mask, Cond=4, Op=5
fcun	Fy, Fz	0xffff9114	Scalar CMC, Set Mask, Cond=0, Op=5
fdiv	Aw, Fy, Fz	0xffff985c	Scalar 71, Unconditional
fdiv	Fx, Fy, Fz	0xffff981c	Scalar 23, Unconditional
fhalf	AC	0xffff9f24	Accumulator Op 25
flog10	AC	0xffff9f3c	Accumulator Op 31
floge	AC	0xffff9f34	Accumulator Op 29
fm1	AC	0xffff9f0c	Accumulator Op 19
fma	Fx, Fy, Aw, Fz	0xffff9414	Scalar 5, Unconditional
fma,c	Fx, Fy, Aw, Fz	0xffff9494	Scalar 5, Nullify if StStack<0> clear
fmax	Fx, Fy, Fz	0xffff942c	Scalar 11, Unconditional
fmax,c	Fx, Fy, Fz	0xffff94ac	Scalar 11, Nullify if StStack<0> clear
fmin	Fx, Fy, Fz	0xffff9424	Scalar 9, Unconditional
fmin,c	Fx, Fy, Fz	0xffff94a4	Scalar 9, Nullify if StStack<0> clear
fmove	Fx, Fz	0xffff9024	Scalar CMC, NoSetMsk, Cond=1, Op=1

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
fms	Fx, Fy, Aw, Fz	0xffff941c	Scalar 7, Unconditional
fms	Fx, Fy, Fz, Aw	0xffff9c04	Scalar 33, Unconditional
fms,c	Fx, Fy, Aw, Fz	0xffff949c	Scalar 7, Nullify if StStack<0> clear
fms,c	Fx, Fy, Fz, Aw	0xffff9c84	Scalar 33, Nullify if StStack<0> clear
fmul	Aw, Fy, Fz	0xffff9854	Scalar 69, Unconditional
fmul	Aw, Fz, Aw	0xffff9454	Scalar 53, Unconditional
fmul	Fx, Fy, Fz	0xffff9814	Scalar 21, Unconditional
fmul	Fx, Fz, Aw	0xffff9c54	Scalar 85, Unconditional
fmul,c	Aw, Fy, Fz	0xffff98d4	Scalar 69, Nullify if StStack<0> clear
fmul,c	Aw, Fz, Aw	0xffff94d4	Scalar 53, Nullify if StStack<0> clear
fmul,c	Fx, Fy, Fz	0xffff9894	Scalar 21, Nullify if StStack<0> clear
fmul,c	Fx, Fz, Aw	0xffff9cd4	Scalar 85, Nullify if StStack<0> clear
fneg	Fx, Fz	0xffff9c1c	Scalar 39, Unconditional
fneg,c	Fx, Fz	0xffff9c9c	Scalar 39, Nullify if StStack<0> clear
fninf	AC	0xffff9f1c	Accumulator Op 23
fpinf	AC	0xffff9f14	Accumulator Op 21
fradabs	AC, Vy	0xffff9994	Vec.Reduc. 21, Non-Masked
fradabs,mf	AC, Vy	0xffff99d4	Vec.Reduc. 69, Use /Mask
fradabs,mt	AC, Vy	0xffff9954	Vec.Reduc. 69, Use Mask
fradd	AC, Vy	0xffff9984	Vec.Reduc. 17, Non-Masked
fradd,mf	AC, Vy	0xffff99c4	Vec.Reduc. 65, Use /Mask
fradd,mt	AC, Vy	0xffff9944	Vec.Reduc. 65, Use Mask
frma	AC, Vy, AC, Vz	0xffff9594	Vec.Reduc. 5, Non-Masked
frma	AC, Vy, Vz, AC	0xffff9584	Vec.Reduc. 1, Non-Masked
frma	AC, Vy, [Vz], AC	0xffff999c	Vec.Reduc. 23, Non-Masked
frma,mf	AC, Vy, AC, Vz	0xffff95d4	Vec.Reduc. 53, Use /Mask
frma,mf	AC, Vy, Vz, AC	0xffff95c4	Vec.Reduc. 49, Use /Mask
frma,mf	AC, Vy, [Vz], AC	0xffff99dc	Vec.Reduc. 71, Use /Mask
frma,mt	AC, Vy, AC, Vz	0xffff9554	Vec.Reduc. 53, Use Mask
frma,mt	AC, Vy, Vz, AC	0xffff9544	Vec.Reduc. 49, Use Mask
frma,mt	AC, Vy, [Vz], AC	0xffff995c	Vec.Reduc. 71, Use Mask
frmax	AC, Vy	0xffff95ac	Vec.Reduc. 11, Non-Masked
frmax,mf	AC, Vy	0xffff95ec	Vec.Reduc. 59, Use /Mask
frmax,mt	AC, Vy	0xffff956c	Vec.Reduc. 59, Use Mask
frmin	AC, Vy	0xffff95a4	Vec.Reduc. 9, Non-Masked
frmin,mf	AC, Vy	0xffff95e4	Vec.Reduc. 57, Use /Mask
frmin,mt	AC, Vy	0xffff9564	Vec.Reduc. 57, Use Mask
frms	AC, Vy, AC, Vz	0xffff959c	Vec.Reduc. 7, Non-Masked
frms	AC, Vy, Vz, AC	0xffff9d84	Vec.Reduc. 33, Non-Masked
frms,mf	AC, Vy, AC, Vz	0xffff95dc	Vec.Reduc. 55, Use /Mask
frms,mf	AC, Vy, Vz, AC	0xffff9dc4	Vec.Reduc. 81, Use /Mask
frms,mt	AC, Vy, AC, Vz	0xffff955c	Vec.Reduc. 55, Use Mask
frms,mt	AC, Vy, Vz, AC	0xffff9d44	Vec.Reduc. 81, Use Mask
frmxabs	AC, Vy	0xffff998c	Vec.Reduc. 19, Non-Masked
frmxabs,mf	AC, Vy	0xffff99cc	Vec.Reduc. 67, Use /Mask
frmxabs,mt	AC, Vy	0xffff994c	Vec.Reduc. 67, Use Mask
frpi	AC	0xffff9f2c	Accumulator Op 27

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
frsm	AC, AC, Vz, Vy	`0xffff958c`	Vec.Reduc. 3, Non-Masked
frsm	AC, Vz, Vy, AC	`0xffff9d8c`	Vec.Reduc. 35, Non-Masked
frsm,mf	AC, AC, Vz, Vy	`0xffff95cc`	Vec.Reduc. 51, Use /Mask
frsm,mf	AC, Vz, Vy, AC	`0xffff9dcc`	Vec.Reduc. 83, Use /Mask
frsm,mt	AC, AC, Vz, Vy	`0xffff954c`	Vec.Reduc. 51, Use Mask
frsm,mt	AC, Vz, Vy, AC	`0xffff9d4c`	Vec.Reduc. 83, Use Mask
fsm	Fx, Aw, Fy, Fz	`0xffff940c`	Scalar 3, Unconditional
fsm	Fx, Fz, Fy, Aw	`0xffff9c0c`	Scalar 35, Unconditional
fsm,c	Fx, Aw, Fy, Fz	`0xffff948c`	Scalar 3, Nullify if StStack<0> clear
fsm,c	Fx, Fz, Fy, Aw	`0xffff9c8c`	Scalar 35, Nullify if StStack<0> clear
fsub	Aw, Aw, Fz	`0xffff944c`	Scalar 51, Unconditional
fsub	Aw, Fy, Aw	`0xffff945c`	Scalar 55, Unconditional
fsub	Aw, Fy, Fz	`0xffff984c`	Scalar 67, Unconditional
fsub	Fx, Aw, Fz	`0xffff9c4c`	Scalar 83, Unconditional
fsub	Fx, Fy, Aw	`0xffff9c5c`	Scalar 87, Unconditional
fsub	Fx, Fy, Fz	`0xffff980c`	Scalar 19, Unconditional
fsub,c	Aw, Aw, Fz	`0xffff94cc`	Scalar 51, Nullify if StStack<0> clear
fsub,c	Aw, Fy, Aw	`0xffff94dc`	Scalar 55, Nullify if StStack<0> clear
fsub,c	Aw, Fy, Fz	`0xffff98cc`	Scalar 67, Nullify if StStack<0> clear
fsub,c	Fx, Aw, Fz	`0xffff9ccc`	Scalar 83, Nullify if StStack<0> clear
fsub,c	Fx, Fy, Aw	`0xffff9cdc`	Scalar 87, Nullify if StStack<0> clear
fsub,c	Fx, Fy, Fz	`0xffff988c`	Scalar 19, Nullify if StStack<0> clear
ftdnm	Fz	`0xffff9138`	Scalar CMC, Set Mask, Cond=1, Op=6
fteq	Fy	`0xffff9164`	Scalar CMC, Set Mask, Cond=3, Op=1
ftge	Fy	`0xffff9144`	Scalar CMC, Set Mask, Cond=2, Op=1
ftgt	Fy	`0xffff9124`	Scalar CMC, Set Mask, Cond=1, Op=1
ftinf	Fz	`0xffff9118`	Scalar CMC, Set Mask, Cond=0, Op=6
ftle	Fy	`0xffff91a4`	Scalar CMC, Set Mask, Cond=5, Op=1
ftlt	Fy	`0xffff91c4`	Scalar CMC, Set Mask, Cond=6, Op=1
ftnan	Fz	`0xffff9198`	Scalar CMC, Set Mask, Cond=4, Op=6
ftne	Fy	`0xffff9184`	Scalar CMC, Set Mask, Cond=4, Op=1
ftnrm	Fz	`0xffff9158`	Scalar CMC, Set Mask, Cond=2, Op=6
ftod	Dx, Fz	`0xffff9c28`	Scalar 42, Unconditional
ftod,c	Dx, Fz	`0xffff9ca8`	Scalar 42, Nullify if StStack<0> clear
ftoi	Ix, Fz	`0xffff9c34`	Scalar 45, Unconditional
ftoi,c	Ix, Fz	`0xffff9cb4`	Scalar 45, Nullify if StStack<0> clear
ftsgn	Fz	`0xffff91f8`	Scalar CMC, Set Mask, Cond=7, Op=6
ftun	Fy	`0xffff9104`	Scalar CMC, Set Mask, Cond=0, Op=1
ftze	Fz	`0xffff9178`	Scalar CMC, Set Mask, Cond=3, Op=6
fuvceq	Vy, Vz	`0xffff9274`	Vector CMC, Move, Cond=3, Op=5
fuvcge	Vy, Vz	`0xffff9254`	Vector CMC, Move, Cond=2, Op=5
fuvcgt	Vy, Vz	`0xffff9234`	Vector CMC, Move, Cond=1, Op=5
fuvcle	Vy, Vz	`0xffff92b4`	Vector CMC, Move, Cond=5, Op=5

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
fuvclt	Vy, Vz	0xffff92d4	Vector CMC, Move, Cond=6, Op=5
fuvcne	Vy, Vz	0xffff9294	Vector CMC, Move, Cond=4, Op=5
fuvcun	Vy, Vz	0xffff9214	Vector CMC, Move, Cond=0, Op=5
fuvdnm	Vz	0xffff9238	Vector CMC, Move, Cond=1, Op=6
fuvinf	Vz	0xffff9218	Vector CMC, Move, Cond=0, Op=6
fuvnan	Vz	0xffff9298	Vector CMC, Move, Cond=4, Op=6
fuvnrm	Vz	0xffff9258	Vector CMC, Move, Cond=2, Op=6
fuvsgn	Vz	0xffff92f8	Vector CMC, Move, Cond=7, Op=6
fuvze	Vz	0xffff9278	Vector CMC, Move, Cond=3, Op=6
fvabs	Vx, Vz	0xffff9e94	Vec.Dyadic 37, Non-Masked
fvabs,mf	Vx, Vz	0xffff9ed4	Vec.Dyadic 85, Use /Mask
fvabs,mt	Vx, Vz	0xffff9e54	Vec.Dyadic 85, Use Mask
fvadd	Vx, Vy, Vz	0xffff9b84	Vec.Triadic 17, Non-Masked
fvadd	Vx, Vz, Aw	0xffff9684	Vec.Dyadic 1, Non-Masked
fvadd,mf	Vx, Vy, Vz	0xffff9bc4	Vec.Triadic 65, Use /Mask
fvadd,mf	Vx, Vz, Aw	0xffff96c4	Vec.Dyadic 49, Use /Mask
fvadd,mf	Vx, [Vz], Vy	0xffff9ac4	Vec.Dyadic 65, Use /Mask
fvadd,mt	Vx, Vy, Vz	0xffff9b44	Vec.Triadic 65, Use Mask
fvadd,mt	Vx, Vz, Aw	0xffff9644	Vec.Dyadic 49, Use Mask
fvadd,mt	Vx, [Vz], Vy	0xffff9a44	Vec.Dyadic 65, Use Mask
fvam	Vx, Vy, Vz, Aw	0xffff9784	Vec.Triadic 1, Non-Masked
fvceq	Vy, [Vz]	0xffff9364	Vector CMC, Compare, Cond=3, Op=1
fvceq	Vy, Vz	0xffff9374	Vector CMC, Compare, Cond=3, Op=5
fvcge	Vy, [Vz]	0xffff9344	Vector CMC, Compare, Cond=2, Op=1
fvcge	Vy, Vz	0xffff9354	Vector CMC, Compare, Cond=2, Op=5
fvcgt	Vy, [Vz]	0xffff9324	Vector CMC, Compare, Cond=1, Op=1
fvcgt	Vy, Vz	0xffff9334	Vector CMC, Compare, Cond=1, Op=5
fvcle	Vy, [Vz]	0xffff93a4	Vector CMC, Compare, Cond=5, Op=1
fvcle	Vy, Vz	0xffff93b4	Vector CMC, Compare, Cond=5, Op=5
fvclt	Vy, [Vz]	0xffff93c4	Vector CMC, Compare, Cond=6, Op=1
fvclt	Vy, Vz	0xffff93d4	Vector CMC, Compare, Cond=6, Op=5
fvcne	Vy, [Vz]	0xffff9384	Vector CMC, Compare, Cond=4, Op=1
fvcne	Vy, Vz	0xffff9394	Vector CMC, Compare, Cond=4, Op=5
fvcun	Vy, [Vz]	0xffff9304	Vector CMC, Compare, Cond=0, Op=1
fvcun	Vy, Vz	0xffff9314	Vector CMC, Compare, Cond=0, Op=5
fvdnm	Vz	0xffff9338	Vector CMC, Compare, Cond=1, Op=6
fvinf	Vz	0xffff9318	Vector CMC, Compare, Cond=0, Op=6
fvma	Vx, Vy, Aw, Vz	0xffff9794	Vec.Triadic 5, Non-Masked
fvma,mf	Vx, Vy, Aw, Vz	0xffff97d4	Vec.Triadic 53, Use /Mask
fvma,mf	Vx, Vy, Vz, Aw	0xffff97c4	Vec.Triadic 49, Use /Mask
fvma,mt	Vx, Vy, Aw, Vz	0xffff9754	Vec.Triadic 53, Use Mask
fvma,mt	Vx, Vy, Vz, Aw	0xffff9744	Vec.Triadic 49, Use Mask
fvmax	Vx, Vy, Vz	0xffff97ac	Vec.Triadic 11, Non-Masked

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
fvmax	Vx, [Vz], Vy	0xffff96ac	Vec.Dyadic 11, Non-Masked
fvmax,mf	Vx, Vy, Vz	0xffff97ec	Vec.Triadic 59, Use /Mask
fvmax,mf	Vx, [Vz], Vy	0xffff96ec	Vec.Dyadic 59, Use /Mask
fvmax,mt	Vx, Vy, Vz	0xffff976c	Vec.Triadic 59, Use Mask
fvmax,mt	Vx, [Vz], Vy	0xffff966c	Vec.Dyadic 59, Use Mask
fvmin	Vx, Vy, Vz	0xffff97a4	Vec.Triadic 9, Non-Masked
fvmin	Vx, [Vz], Vy	0xffff96a4	Vec.Dyadic 9, Non-Masked
fvmin,mf	Vx, Vy, Vz	0xffff97e4	Vec.Triadic 57, Use /Mask
fvmin,mf	Vx, [Vz], Vy	0xffff96e4	Vec.Dyadic 57, Use /Mask
fvmin,mt	Vx, Vy, Vz	0xffff9764	Vec.Triadic 57, Use Mask
fvmin,mt	Vx, [Vz], Vy	0xffff9664	Vec.Dyadic 57, Use Mask
fvmove	Vx, Vz	0xffff9224	Vector CMC, Move, Cond=1, Op=1
fvms	Vx, Vy, Aw, Vz	0xffff979c	Vec.Triadic 7, Non-Masked
fvms	Vx, Vy, Vz, Aw	0xffff9f84	Vec.Triadic 33, Non-Masked
fvms,mf	Vx, Vy, Aw, Vz	0xffff97dc	Vec.Triadic 55, Use /Mask
fvms,mf	Vx, Vy, Vz, Aw	0xffff9fc4	Vec.Triadic 81, Use /Mask
fvms,mt	Vx, Vy, Aw, Vz	0xffff975c	Vec.Triadic 55, Use Mask
fvms,mt	Vx, Vy, Vz, Aw	0xffff9f44	Vec.Triadic 81, Use Mask
fvmul	Vx, Vy, Vz	0xffff9b94	Vec.Triadic 21, Non-Masked
fvmul	Vx, Vz, Aw	0xffff9694	Vec.Dyadic 5, Non-Masked
fvmul	Vx, [Vz], Vy	0xffff9a94	Vec.Dyadic 21, Non-Masked
fvmul,mf	Vx, Vy, Vz	0xffff9bd4	Vec.Triadic 69, Use /Mask
fvmul,mf	Vx, Vz, Aw	0xffff96d4	Vec.Dyadic 53, Use /Mask
fvmul,mf	Vx, [Vz], Vy	0xffff9ad4	Vec.Dyadic 69, Use /Mask
fvmul,mt	Vx, Vy, Vz	0xffff9b54	Vec.Triadic 69, Use Mask
fvmul,mt	Vx, Vz, Aw	0xffff9654	Vec.Dyadic 53, Use Mask
fvmul,mt	Vx, [Vz], Vy	0xffff9a54	Vec.Dyadic 69, Use Mask
fvnan	Vz	0xffff9398	Vector CMC, Compare, Cond=4, Op=6
fvneg	Vx, Vz	0xffff9e9c	Vec.Dyadic 39, Non-Masked
fvneg,mf	Vx, Vz	0xffff9edc	Vec.Dyadic 87, Use /Mask
fvneg,mt	Vx, Vz	0xffff9e5c	Vec.Dyadic 87, Use Mask
fvnrm	Vz	0xffff9358	Vector CMC, Compare, Cond=2, Op=6
fvsgn	Vz	0xffff93f8	Vector CMC, Compare, Cond=7, Op=6
fvsm	Vx, Aw, Vy, Vz	0xffff978c	Vec.Triadic 3, Non-Masked
fvsm	Vx, Vz, Vy, Aw	0xffff9f8c	Vec.Triadic 35, Non-Masked
fvsm,mf	Vx, Aw, Vy, Vz	0xffff97cc	Vec.Triadic 51, Use /Mask
fvsm,mf	Vx, Vz, Vy, Aw	0xffff9fcc	Vec.Triadic 83, Use /Mask
fvsm,mt	Vx, Aw, Vy, Vz	0xffff974c	Vec.Triadic 51, Use Mask
fvsm,mt	Vx, Vz, Vy, Aw	0xffff9f4c	Vec.Triadic 83, Use Mask
fvsub	Vx, Aw, Vz	0xffff968c	Vec.Dyadic 3, Non-Masked
fvsub	Vx, Vy, Vz	0xffff9b8c	Vec.Triadic 19, Non-Masked
fvsub	Vx, Vz, Aw	0xffff969c	Vec.Dyadic 7, Non-Masked
fvsub	Vx, [Vz], Vy	0xffff9a9c	Vec.Dyadic 23, Non-Masked
fvsub,mf	Vx, Aw, Vz	0xffff96cc	Vec.Dyadic 51, Use /Mask
fvsub,mf	Vx, Vy, Vz	0xffff9bcc	Vec.Triadic 67, Use /Mask
fvsub,mf	Vx, Vy, [Vz]	0xffff9acc	Vec.Dyadic 67, Use /Mask

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
fvsub,mf	Vx, Vz, Aw	0xffff96dc	Vec.Dyadic 55, Use /Mask
fvsub,mf	Vx, [Vz], Vy	0xffff9adc	Vec.Dyadic 71, Use /Mask
fvsub,mt	Vx, Aw, Vz	0xffff964c	Vec.Dyadic 51, Use Mask
fvsub,mt	Vx, Vy, Vz	0xffff9b4c	Vec.Triadic 67, Use Mask
fvsub,mt	Vx, Vy, [Vz]	0xffff9a4c	Vec.Dyadic 67, Use Mask
fvsub,mt	Vx, Vz, Aw	0xffff965c	Vec.Dyadic 55, Use Mask
fvsub,mt	Vx, [Vz], Vy	0xffff9a5c	Vec.Dyadic 71, Use Mask
fvze	Vz	0xffff9378	Vector CMC, Compare, Cond=3, Op=6
i0	AC	0xffff9b20	Accumulator Op 8
i0	Aw	0xffff9b24	Accumulator Op 9
iadd	Ix, Iy, Iz	0xffff9830	Scalar 28, Unconditional
iadd,c	Ix, Iy, Iz	0xffff98b0	Scalar 28, Nullify if StStack<0> clear
iand	Ix, Iy, Iz	0xffff943c	Scalar 15, Unconditional
iand,c	Ix, Iy, Iz	0xffff94bc	Scalar 15, Nullify if StStack<0> clear
iceq	Iy, Iz	0xffff917c	Scalar CMC, Set Mask, Cond=3, Op=7
icge	Iy, Iz	0xffff915c	Scalar CMC, Set Mask, Cond=2, Op=7
icgt	Iy, Iz	0xffff913c	Scalar CMC, Set Mask, Cond=1, Op=7
icle	Iy, Iz	0xffff91bc	Scalar CMC, Set Mask, Cond=5, Op=7
iclt	Iy, Iz	0xffff91dc	Scalar CMC, Set Mask, Cond=6, Op=7
icne	Iy, Iz	0xffff919c	Scalar CMC, Set Mask, Cond=4, Op=7
icov	Iy, Iz	0xffff911c	Scalar CMC, Set Mask, Cond=0, Op=7
icsgn	Iy, Iz	0xffff91fc	Scalar CMC, Set Mask, Cond=7, Op=7
imax	Ix, Iy, Iz	0xffff983c	Scalar 31, Unconditional
imax,c	Ix, Iy, Iz	0xffff98bc	Scalar 31, Nullify if StStack<0> clear
imin	Ix, Iy, Iz	0xffff9834	Scalar 29, Unconditional
imin,c	Ix, Iy, Iz	0xffff98b4	Scalar 29, Nullify if StStack<0> clear
imove	Ix, Iz	0xffff902c	Scalar CMC, NoSetMsk, Cond=1, Op=3
inot	Ix, Iz	0xffff9c38	Scalar 46, Unconditional
inot,c	Ix, Iz	0xffff9cb8	Scalar 46, Nullify if StStack<0> clear
ior	Ix, Iy, Iz	0xffff9434	Scalar 13, Unconditional
ior,c	Ix, Iy, Iz	0xffff94b4	Scalar 13, Nullify if StStack<0> clear
iradd	AC, Vy	0xffff99b0	Vec.Reduc. 28, Non-Masked
iradd,mf	AC, Vy	0xffff99f0	Vec.Reduc. 76, Use /Mask
iradd,mt	AC, Vy	0xffff9970	Vec.Reduc. 76, Use Mask
irand	AC, Vy	0xffff95bc	Vec.Reduc. 15, Non-Masked
irand,mf	AC, Vy	0xffff95fc	Vec.Reduc. 63, Use /Mask
irand,mt	AC, Vy	0xffff957c	Vec.Reduc. 63, Use Mask
irmax	AC, Vy	0xffff99bc	Vec.Reduc. 31, Non-Masked
irmax,mf	AC, Vy	0xffff99fc	Vec.Reduc. 79, Use /Mask
irmax,mt	AC, Vy	0xffff997c	Vec.Reduc. 79, Use Mask
irmin	AC, Vy	0xffff99b4	Vec.Reduc. 29, Non-Masked
irmin,mf	AC, Vy	0xffff99f4	Vec.Reduc. 77, Use /Mask
irmin,mt	AC, Vy	0xffff9974	Vec.Reduc. 77, Use Mask

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
iror	AC, Vy	0xffff95b4	Vec.Reduc. 13, Non-Masked
iror,mf	AC, Vy	0xffff95f4	Vec.Reduc. 61, Use /Mask
iror,mt	AC, Vy	0xffff9574	Vec.Reduc. 61, Use Mask
irxor	AC, Vy	0xffff95b8	Vec.Reduc. 14, Non-Masked
irxor,mf	AC, Vy	0xffff95f8	Vec.Reduc. 62, Use /Mask
irxor,mt	AC, Vy	0xffff9578	Vec.Reduc. 62, Use Mask
isub	Ix, Iy, Iz	0xffff9838	Scalar 30, Unconditional
isub,c	Ix, Iy, Iz	0xffff98b8	Scalar 30, Nullify if StStack<0> clear
iteq	Iy	0xffff916c	Scalar CMC, Set Mask, Cond=3, Op=3
itge	Iy	0xffff914c	Scalar CMC, Set Mask, Cond=2, Op=3
itgt	Iy	0xffff912c	Scalar CMC, Set Mask, Cond=1, Op=3
itle	Iy	0xffff91ac	Scalar CMC, Set Mask, Cond=5, Op=3
itlt	Iy	0xffff91cc	Scalar CMC, Set Mask, Cond=6, Op=3
itne	Iy	0xffff918c	Scalar CMC, Set Mask, Cond=4, Op=3
itod	Dx, Iz	0xffff9c20	Scalar 40, Unconditional
itod,c	Dx, Iz	0xffff9ca0	Scalar 40, Nullify if StStack<0> clear
itof	Fx, Iz	0xffff9c24	Scalar 41, Unconditional
itof,c	Fx, Iz	0xffff9ca4	Scalar 41, Nullify if StStack<0> clear
itov	Iy	0xffff910c	Scalar CMC, Set Mask, Cond=0, Op=3
itsgn	Iy	0xffff91ec	Scalar CMC, Set Mask, Cond=7, Op=3
iuvceq	Vy, Vz	0xffff927c	Vector CMC, Move, Cond=3, Op=7
iuvcge	Vy, Vz	0xffff925c	Vector CMC, Move, Cond=2, Op=7
iuvcgt	Vy, Vz	0xffff923c	Vector CMC, Move, Cond=1, Op=7
iuvcle	Vy, Vz	0xffff92bc	Vector CMC, Move, Cond=5, Op=7
iuvclt	Vy, Vz	0xffff92dc	Vector CMC, Move, Cond=6, Op=7
iuvcne	Vy, Vz	0xffff929c	Vector CMC, Move, Cond=4, Op=7
iuvcov	Vy, Vz	0xffff921c	Vector CMC, Move, Cond=0, Op=7
iuvcsgn	Vy, Vz	0xffff92fc	Vector CMC, Move, Cond=7, Op=7
ivadd	Vx, Vy, Vz	0xffff9bb0	Vec.Triadic 28, Non-Masked
ivadd	Vx, [Vz], Vy	0xffff9ab0	Vec.Dyadic 28, Non-Masked
ivadd,mf	Vx, Vy, Vz	0xffff9bf0	Vec.Triadic 76, Use /Mask
ivadd,mf	Vx, [Vz], Vy	0xffff9af0	Vec.Dyadic 76, Use /Mask
ivadd,mt	Vx, Vy, Vz	0xffff9b70	Vec.Triadic 76, Use Mask
ivadd,mt	Vx, [Vz], Vy	0xffff9a70	Vec.Dyadic 76, Use Mask
ivand	Vx, Vy, Vz	0xffff97bc	Vec.Triadic 15, Non-Masked
ivand	Vx, [Vz], Vy	0xffff96bc	Vec.Dyadic 15, Non-Masked
ivand,mf	Vx, Vy, Vz	0xffff97fc	Vec.Triadic 63, Use /Mask
ivand,mf	Vx, [Vz], Vy	0xffff96fc	Vec.Dyadic 63, Use /Mask
ivand,mt	Vx, Vy, Vz	0xffff977c	Vec.Triadic 63, Use Mask
ivand,mt	Vx, [Vz], Vy	0xffff967c	Vec.Dyadic 63, Use Mask
ivceq	Vy, [Vz]	0xffff936c	Vector CMC, Compare, Cond=3, Op=3
ivceq	Vy, Vz	0xffff937c	Vector CMC, Compare, Cond=3, Op=7
ivcge	Vy, [Vz]	0xffff934c	Vector CMC, Compare, Cond=2, Op=3
ivcge	Vy, Vz	0xffff935c	Vector CMC, Compare, Cond=2, Op=7
ivcgt	Vy, [Vz]	0xffff932c	Vector CMC, Compare, Cond=1, Op=3
ivcgt	Vy, Vz	0xffff933c	Vector CMC, Compare, Cond=1, Op=7
ivcle	Vy, [Vz]	0xffff93ac	Vector CMC, Compare, Cond=5, Op=3

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
ivcle	Vy, Vz	0xffff93bc	Vector CMC, Compare, Cond=5, Op=7
ivclt	Vy, [Vz]	0xffff93cc	Vector CMC, Compare, Cond=6, Op=3
ivclt	Vy, Vz	0xffff93dc	Vector CMC, Compare, Cond=6, Op=7
ivcne	Vy, [Vz]	0xffff938c	Vector CMC, Compare, Cond=4, Op=3
ivcne	Vy, Vz	0xffff939c	Vector CMC, Compare, Cond=4, Op=7
ivcov	Vy, [Vz]	0xffff930c	Vector CMC, Compare, Cond=0, Op=3
ivcov	Vy, Vz	0xffff931c	Vector CMC, Compare, Cond=0, Op=7
ivcsgn	Vy, [Vz]	0xffff93ec	Vector CMC, Compare, Cond=7, Op=3
ivcsgn	Vy, Vz	0xffff93fc	Vector CMC, Compare, Cond=7, Op=7
ivmax	Vx, Vy, Vz	0xffff9bbc	Vec.Triadic 31, Non-Masked
ivmax	Vx, [Vz], Vy	0xffff9abc	Vec.Dyadic 31, Non-Masked
ivmax,mf	Vx, Vy, Vz	0xffff9bfc	Vec.Triadic 79, Use /Mask
ivmax,mf	Vx, [Vz], Vy	0xffff9afc	Vec.Dyadic 79, Use /Mask
ivmax,mt	Vx, Vy, Vz	0xffff9b7c	Vec.Triadic 79, Use Mask
ivmax,mt	Vx, [Vz], Vy	0xffff9a7c	Vec.Dyadic 79, Use Mask
ivmin	Vx, Vy, Vz	0xffff9bb4	Vec.Triadic 29, Non-Masked
ivmin	Vx, [Vz], Vy	0xffff9ab4	Vec.Dyadic 29, Non-Masked
ivmin,mf	Vx, Vy, Vz	0xffff9bf4	Vec.Triadic 77, Use /Mask
ivmin,mf	Vx, [Vz], Vy	0xffff9af4	Vec.Dyadic 77, Use /Mask
ivmin,mt	Vx, Vy, Vz	0xffff9b74	Vec.Triadic 77, Use Mask
ivmin,mt	Vx, [Vz], Vy	0xffff9a74	Vec.Dyadic 77, Use Mask
ivmove	Vx, Vz	0xffff922c	Vector CMC, Move, Cond=1, Op=3
ivnot	Vx, Vz	0xffff9eb8	Vec.Dyadic 46, Non-Masked
ivnot,mf	Vx, Vz	0xffff9ef8	Vec.Dyadic 94, Use /Mask
ivnot,mt	Vx, Vz	0xffff9e78	Vec.Dyadic 94, Use Mask
ivor	Vx, Vy, Vz	0xffff97b4	Vec.Triadic 13, Non-Masked
ivor	Vx, [Vz], Vy	0xffff96b4	Vec.Dyadic 13, Non-Masked
ivor,mf	Vx, Vy, Vz	0xffff97f4	Vec.Triadic 61, Use /Mask
ivor,mf	Vx, [Vz], Vy	0xffff96f4	Vec.Dyadic 61, Use /Mask
ivor,mt	Vx, Vy, Vz	0xffff9774	Vec.Triadic 61, Use Mask
ivor,mt	Vx, [Vz], Vy	0xffff9674	Vec.Dyadic 61, Use Mask
ivsub	Vx, Vy, Vz	0xffff9bb8	Vec.Triadic 30, Non-Masked
ivsub	Vx, Vy, [Vz]	0xffff9ab8	Vec.Dyadic 30, Non-Masked
ivsub	Vx, [Vz], Vy	0xffff96b0	Vec.Dyadic 12, Non-Masked
ivsub,mf	Vx, Vy, Vz	0xffff9bf8	Vec.Triadic 78, Use /Mask
ivsub,mf	Vx, Vy, [Vz]	0xffff9af8	Vec.Dyadic 78, Use /Mask
ivsub,mf	Vx, [Vz], Vy	0xffff96f0	Vec.Dyadic 60, Use /Mask
ivsub,mt	Vx, Vy, Vz	0xffff9b78	Vec.Triadic 78, Use Mask
ivsub,mt	Vx, Vy, [Vz]	0xffff9a78	Vec.Dyadic 78, Use Mask
ivsub,mt	Vx, [Vz], Vy	0xffff9670	Vec.Dyadic 60, Use Mask
ivxor	Vx, Vy, Vz	0xffff97b8	Vec.Triadic 14, Non-Masked
ivxor	Vx, [Vz], Vy	0xffff96b8	Vec.Dyadic 14, Non-Masked
ivxor,mf	Vx, Vy, Vz	0xffff97f8	Vec.Triadic 62, Use /Mask
ivxor,mf	Vx, [Vz], Vy	0xffff96f8	Vec.Dyadic 62, Use /Mask

Table 10-4. FPU Instructions, by Mnemonic (continued)

Mnemonic	Parameters	Location	Comments
ivxor,mt	Vx, Vy, Vz	0xffff9778	Vec.Triadic 62, Use Mask
ivxor,mt	Vx, [Vz], Vy	0xffff9678	Vec.Dyadic 62, Use Mask
ixor	Ix, Iy, Iz	0xffff9438	Scalar 14, Unconditional
ixor,c	Ix, Iy, Iz	0xffff94b8	Scalar 14, Nullify if StStack<0> clear
pass	Aw, Vz	0xffff9b38	Accumulator Op 14
pass	Dx, Dz	0xffff9060	Scalar CMC, NoSetMsk, Cond=3, Op=0
pass	Fx, Iz	0xffff9064	Scalar CMC, NoSetMsk, Cond=3, Op=1
pass	Ix, Iz	0xffff906c	Scalar CMC, NoSetMsk, Cond=3, Op=3
pass	Vx, Aw	0xffff9b30	Accumulator Op 12
pass4	AC, Vz	0xffff9b3c	Accumulator Op 15
pass4	Vx, AC	0xffff9b34	Accumulator Op 13
passcf	Dx, Dz	0xffff90e0	Scalar CMC, NoSetMsk, Cond=7, Op=0
passcf	Fx, Fz	0xffff90e4	Scalar CMC, NoSetMsk, Cond=7, Op=1
passcf	Ix, Fz	0xffff90ec	Scalar CMC, NoSetMsk, Cond=7, Op=3
passct	Dx, Dz	0xffff90c0	Scalar CMC, NoSetMsk, Cond=6, Op=0
passct	Fx, Fz	0xffff90c4	Scalar CMC, NoSetMsk, Cond=6, Op=1
passct	Ix, Fz	0xffff90cc	Scalar CMC, NoSetMsk, Cond=6, Op=3
vdtof	Vx, Vz	0xffff9eac	Vec.Dyadic 43, Non-Masked
vdtof,mf	Vx, Vz	0xffff9eec	Vec.Dyadic 91, Use /Mask
vdtof,mt	Vx, Vz	0xffff9e6c	Vec.Dyadic 91, Use Mask
vdtoi	Vx, Vz	0xffff9eb0	Vec.Dyadic 44, Non-Masked
vdtoi,mf	Vx, Vz	0xffff9ef0	Vec.Dyadic 92, Use /Mask
vdtoi,mt	Vx, Vz	0xffff9e70	Vec.Dyadic 92, Use Mask
vftod	Vx, Vz	0xffff9ea8	Vec.Dyadic 42, Non-Masked
vftod,mf	Vx, Vz	0xffff9ee8	Vec.Dyadic 90, Use /Mask
vftod,mt	Vx, Vz	0xffff9e68	Vec.Dyadic 90, Use Mask
vftoi	Vx, Vz	0xffff9eb4	Vec.Dyadic 45, Non-Masked
vftoi,mf	Vx, Vz	0xffff9ef4	Vec.Dyadic 93, Use /Mask
vftoi,mt	Vx, Vz	0xffff9e74	Vec.Dyadic 93, Use Mask
vitod	Vx, Vz	0xffff9ea0	Vec.Dyadic 40, Non-Masked
vitod,mf	Vx, Vz	0xffff9ee0	Vec.Dyadic 88, Use /Mask
vitod,mt	Vx, Vz	0xffff9e60	Vec.Dyadic 88, Use Mask
vitof	Vx, Vz	0xffff9ea4	Vec.Dyadic 41, Non-Masked
vitof,mf	Vx, Vz	0xffff9ee4	Vec.Dyadic 89, Use /Mask
vitof,mt	Vx, Vz	0xffff9e64	Vec.Dyadic 89, Use Mask
vpass	Vx, Vz	0xffff9260	Vector CMC, Move, Cond=3, Op=0
vpass,mf	Vx, Vz	0xffff92c0	Vector CMC, Move, Cond=6, Op=0
vpass,mfc	Vx, Vz	0xffff9280	Vector CMC, Move, Cond=4, Op=0
vpass,mt	Vx, Vz	0xffff92e0	Vector CMC, Move, Cond=7, Op=0
vpass,mtc	Vx, Vz	0xffff92a0	Vector CMC, Move, Cond=5, Op=0
vsexp	AC, [Vz]	0xffff9b2c	Accumulator Op 11
vsexp	Vx, [Vz]	0xffff9240	Vector CMC, Move, Cond=2, Op=01

Stack Frame

The stack frame for a function call is illustrated in Figure 10-1.

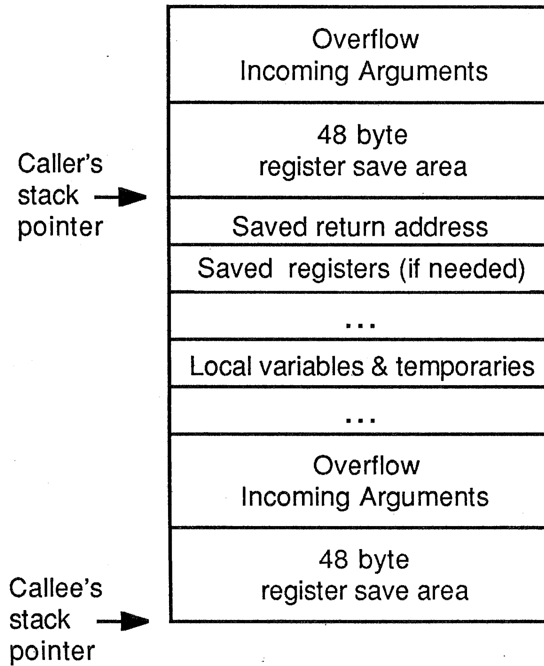


Figure 10-1. Stack Frame

Double precision arguments are aligned on 8-byte boundaries and holes are left if needed for this alignment. Short and byte arguments are widened to 32-bit values and stored on 4-byte boundaries.

By convention, the first several arguments are stored in registers. However, space is always kept on the stack for these arguments, in case they need to be saved.

The stack frame moves exactly once for each call and return. Very simple functions may not need to use the stack and the stack frame may not move at all. The stack pointer is always 8-byte aligned. All stack sizes must be rounded up to multiples of 8 bytes.

The call/return process consists of 4 phases:

- The calling program:
  - Saves those of registers $v1–$t7, and the *caller save* floating point registers that are needed after the call.
  - Puts the arguments in registers.
  - Puts arguments that won't fit in registers on the stack.
  - Does a branch and link to the beginning of the subroutine.

- The called program:
  - Decrements the stack pointer to allocate the new stack frame.
  - If the called program in turn calls other programs, the return address is saved on the stack.
  - Saves those of registers $s0–$s7 and the *callee save* floating point registers that are used.
  - Begins execution.

- To return to the caller, the called program:
  - Puts the return value into register $v0 if it is an integer or a pointer, and into register F0 if it is a float or a double.
  - Restores the return address to the link register.
  - Restores any of the registers $s0–$s7 and the *callee save* floating point registers that were saved.
  - Increments the stack pointer to the previous value.
  - Returns to the return address.

- After the return, the calling program:
  - Restores any of the registers $v1–$t7 and the *caller save* floating point registers that were saved before the call was made.

If a function makes no calls or does not change registers $s0–$s7 and the *callee save* floating point registers, there is no need for the function to touch the stack (except, perhaps, to store its arguments there). In this case, the code is simplified because there is no need to get a new stack frame nor to do any stores or reloads of the stack. The link address remains in register $ra throughout the called function's execution.

Data Layout In Memory

NOTE
In the floating point formats, when dealing with numbers at or close to the limits of the range, it is possible to exceed the range during ASCII to binary conversion and vice versa. This is caused by rounding errors.

Stardent Fortran has eleven data types:

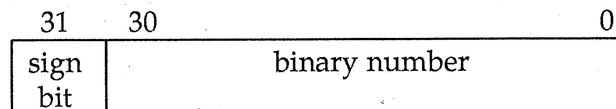
- integer, also called INTEGER*4
- short integer, also called INTEGER*2
- BYTE
- real, also called REAL*4
- double precision, also called REAL*8
- complex, also called COMPLEX*8
- double complex, also called COMPLEX*16
- logical, also called LOGICAL*4
- short logical, also called LOGICAL*2
- byte logical, also called LOGICAL*1
- character

When stored in memory, each has the format described in this appendix.

Integer Format

An integer datum is always an exact representation of a short integer of value positive, negative, or 0. The integer format, **INTEGER*4**, occupies one 32-bit word and has a range of -2^{31} to $+2^{31}$-1 or –2 147 483 648 to +2 147 483 647.

Table 10-5. Integer Data Format (INTEGER*4)



Short Integer Format

A short integer format, **INTEGER*2**, occupies half of one 32-bit word and has a range of -2^{15} to $+2^{15}-1$ or -32768 to 32767.

Table 10-6. Short Integer Format (INTEGER*2)

15	14 — 0
sign bit	binary number

Real Format

A real datum is a processor approximation to a real number having a positive, negative, or zero value. In Stardent Fortran, real format corresponds to IEEE Single Format. This format is capable of representing numbers in the range of – inf to zero to + inf, as well as **NaN**, which stands for "Not a Number". The real format, **REAL*4**, occupies one 32-bit word in memory and has an approximate range of 1.175495E-38 to 3.402823E+38.

The real format has 1-bit sign, an 8-bit exponent, and a 23-bit fraction. Significance is approximately seven decimal digits.

- The sign bit is 0 for plus, 1 for minus.
- The exponent field contains 127 plus the actual exponent (power of 2) of the number. Exponent fields containing all 0's and all 1's are "reserved." Special interpretations given to the numeric representation are as follows:
  - If the exponent is 0 and the fraction 0, the number is interpreted as a signed 0.
  - If the exponent is 0 and the fraction not 0, the number is assumed to be "denormalized." Floating point numbers are usually stored in a "normalized" form with a binary point to the left of the fraction field and an implied leading 1 to the left of the binary point.
  - If the exponent is all 1s and the fraction is 0, the number is regarded as a signed infinity. If the exponent is all 1s and the fraction is not 0, then the interpretation is "not-a-number" (NaN).

Table 10-7. Real Format

31 sign of frac	30 exponent bits 23	22 fraction bits 0
sign of frac	exponent bits	fraction bits

Double Precision Format

A double precision datum is a processor approximation to a real number having a positive, negative or zero value. The Stardent Fortran double precision format corresponds to the IEEE Double Format. The double precision format, **REAL*8** or **DOUBLE PRECISION**, occupies two consecutive 32-bit words in memory, and has an approximate range of

```
-1.79769313486231*10^308 to -2.22507385850721*10^-308
0
+2.22507385850721*10^-308 to +1.79769313486231*10^308
```

The double precision format has an 11-bit exponent and a 52-bit fraction. Significance is approximately 16 decimal digits. The sign bit is 0 for plus, 1 for minus. The exponent field contains 1023 plus the actual exponent (power of 2) of the number. Exponent fields containing all 0's and all 1's are reserved.

- If the exponent is 0 and the fraction 0, the number is interpreted as a signed 0.
- If the exponent is 0 and the fraction not 0, the number is assumed to be "denormalized." Floating point numbers are usually stored in a "normalized" form with a binary point to the left of the fraction field and an implied leading 1 to the left of the binary point.
  - If the exponent is all 1s and the fraction is 0, the number is regarded as a signed infinity.
  - If the exponent is all 1s and the fraction is not 0, then the interpretation is "not-a-number" (NAN).

In Table 10-8 below, the position at which the implied binary decimal point is placed is just to the left of bit position 51.

Table 10-8. Double Precision Format

63	62 ... 52	51 ... 0
sign bit	exponent bits	binary fraction bits

Complex Format

A complex datum is a processor approximation to the value of a complex number. The complex format, **COMPLEX*8**, occupies two consecutive 32-bit words in memory. Both the real and imaginary parts have an approximate range of $1.2*10^{-39}$ to $3.4*10^{38}$.

Both the real and the imaginary parts have 23-bit fractions and 8-bit exponents; both have the same significance as a real number. The sign of the exponent is determined in the same manner as that of a real number.

Table 10-9. Complex Format

Real Part

31	30 ... 23	22 ... 0	
sign of frac	exponent bits	fraction bits	Word 1

Imaginary Part

31	30 ... 23	22 ... 0	
sign of frac	exponent bits	fraction bits	Word 2

Double Complex Format

A complex datum is a processor approximation to the value of a complex number. The double complex format, **COMPLEX*16** or **DOUBLE COMPLEX**, occupies four consecutive 32-bit words in memory. Both the real and imaginary parts have an approximate range of $2.2*10^{-308}$ to $1.8*10^{308}$. Both the real and the imaginary parts have 52-bit fractions and 11-bit exponents; both have the same significance as a double precision number. The sign of the exponent is determined in the same manner as that of a double precision number.

Table 10-10. Double Complex

63	62 — 52	51 — 0	
sign bit	exponent bits	binary fraction bits	Words 1,2
	Imaginary Part		
63	62 — 52	51 — 0	
sign bit	exponent bits	binary fraction bits	Words 3,4

Logical Format

A logical datum is a representation of *true* or *false*, with 0 representing *false*, and any nonzero value representing *true*. The logical format, **LOGICAL*4**, occupies one 32-bit word in memory.

Table 10-11. Logical Data Format

31 — 1	0	
zeros	0	FALSE
31 — 1	0	
undefined	1	TRUE

Short Logical Format

A logical datum is a representation of *true* or *false*, with 0 representing *false*, and any nonzero value representing *true*. The short logical format, **LOGICAL*2**, occupies half of one 32-bit word in memory.

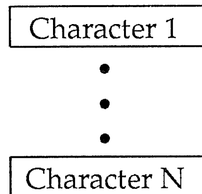
Table 10-12. Short Logical Data Format

15	10	
zeros	0	FALSE
15	10	
undefined	1	TRUE

Character Format

A character datum is a character string taken from the ASCII character set. ASCII characters occupy one byte (eight bits) of a 32-bit word, and are packed four to a word in memory. Character strings are stored in memory as a sequence of ASCII codes, 1 per byte.

Table 10-13. Character Data Format



PORTING CODE

CHAPTER ELEVEN

This chapter describes many of the issues involved in porting your code from another system to the Stardent 1500/3000. These issues include the way that the Stardent 1500/3000 compilers handle certain data types and conditions as compared to the way these same conditions are handled by other compilers. In situations where a difference may exist between the results obtained on a Stardent 1500/3000 and the results obtained on some other system, the descriptions here show the Stardent 1500/3000 results, without necessarily directly contrasting the result with that of other systems. The important points that are covered here include the following:

- The data types that the Stardent 1500/3000 compilers handle.
- The alignment of those data types.
- The Stardent 1500/3000 floating point representation and precision.
- How Stardent 1500/3000 generates and handles exception conditions.
- How parameter passing methods could affect the results of your computations.
- For compatibility with CRAY Fortran, certain additional compiler directives (of the form CDIR$ XXX) are accepted.
- The general syntax and features of Stardent 1500/3000 Fortran and C, including extensions that can be useful.

Finally, there is a section that contains the answers to a few frequently asked porting questions.

Data Types

The following data types are handled by the compilers. Two tables of data types are provided, one for C and the other for Fortran.

Data Types In C		
Type	Size	Comment
char	8 bits	
int	32 bits	
long	32 bits	long integer
short	16 bits	short integer
float	32 bits	single–precision floating point value
double	64 bits	double–precision floating point value
void		value-less

Data Types in Fortran		
Type	Size	Comment
BYTE	*1	byte aligned
INTEGER	*2,*4	half/full aligned
REAL	*4,*8	Full/double aligned
COMPLEX	*8,*16	double aligned
CHARACTER	*n	

Data Alignment

The processor used by the Stardent 1500/3000, requires **strict** alignment of data for half-word, full-word, and double-word operations. Run-time alignment errors result in *floating exceptions*, such as when a Fortran program passes an address of a REAL*4 to a subprogram that in turn treats it as though it is a REAL*8, or when a C program passes an off-alignment pointer to a double-precision value.

When storage is allocated in COMMON, or placed in local storage, or obtained by the C function *malloc*, the compiler properly aligns data placed in these areas by appropriately padding the data to correct the alignment. As on other systems, the most efficient use of memory is to arrange each common block such that the largest elements are declared first. This results in the use

of the fewest padding bytes being added to the storage request.

When contradictory requests are made, a compiler error (starting with version 2.0 of the Stardent 1500/3000 compiler) is issued.

A special note about the handling of data for which an EQUIVALENCE has been issued — the *leftmost* byte of the equivalenced block is double-word aligned. In this case, *leftmost* is defined as that equivalenced item which is specified first in the parameter list given to the EQUIVALENCE statement. As an example of a misalignment that causes a runtime error, consider the following:

```
BYTE Z(1000)
REAL*8 W(100)
EQUIVALENCE(Z(100), W(1))
PRINT *, W(1)
```

This causes an alignment error because the compiler places the first element Z on a double-word boundary, then attempts to reference the W array from the 100th element of Z. Since the MIPS processor requires strict alignment of data items, a REAL*8 must be aligned on a double-word boundary, and the 100th element of Z is not so-aligned, this is an error.

Floating Point Representation

The floating point representation of a Stardent 1500/3000 is IEEE floating point. It is **not** binary representation compatible with any of the following processors: DEC VAX, IBM 3090, CDC, or CRAY. The order of magnitude (shown in the table), and the range of numbers that can be represented and the precision are different:

Processor	Single Precision	Double Precision
Stardent 1500/3000	10**38	10**308
IBM	10**75	10**75
VAX	10**65	10**65
CRAY		10**2466

Machine Representations: IEEE Floating Point

In Chapter 10, *Language Interfacing*, in the section titled *Data Layout In Memory*, the format for both single- and double-precision floating point IEEE representations are shown. The primary point to notice in that section is that the single-precision representation is **not** a prefix of a double-precision representation of a number. The formats are entirely different so as to allow a double-precision number to have a wider range of exponents.

Please refer to the referenced section to see the layout. Viewing the layout reinforces the requirement for proper data alignment when passing parameters from one function to another.

Since Fortran does not check for agreement of type between CALL and INVOCATION parameters of subprograms, **wrong** results can be generated. (VAX and IBM are both tolerant of this error because of their floating point representations.)

Here is a sample Fortran program that illustrates the problem.

```
C      SINGE.F
       REAL*8 ZED,GFUNC
          ZED=1.0d0
          ZED=GFUNC(ZED)
          WRITE(6,*)'returned value is: ',ZED
       END

       REAL*4 FUNCTION GFUNC(X)
       IF(X.NE.1.0) THEN
          WRITE(6,*)'value received is not equal to 1'
          WRITE(6,*)'so returning 1 to caller'
          GFUNC=1.0
       ELSE
          GFUNC=X+1.0
       ENDIF
       END
```

And here is the compilation and its result:

```
test@aarrgghh[2] fc -O0 singe.f
    singe.f
test@aarrgghh[3] a.out
         value received is not equal to 1
         so returning 1 to caller
         returned value is -2.8954213432107195-308
```

Unless prototypes are used, in C parameters passed by value are always coerced to double, thus avoiding the problem. However, if you pass parameters in C by *reference*, it **can** be a problem.

Error Conditions During Floating Point Operations

During IEEE floating point operations, error conditions are represented by silently (that is, without generating any interrupts) inserting special representations into the results. Here are the special representations that occur during error conditions:

Exponent	Fraction	Meaning
0	0	0
0	Not zero	denormalized number
11111...11	0	signed infinity
11111...11	Not zero	Not A Number (NaN)

Underflow occurs when it is not possible to represent a result as a normalized number. In other words, the denormalized number IEEE representation is not used.

Sample Program

The following sample program shows results from certain calculations using the underlying IEEE format on the Stardent 1500/3000:

```
      REAL*4 a,b,c,d
C
      READ(5,*) b,c
      WRITE(6,*)b,c
      IF(b.NE.0.0) a = SQRT(-1.0)
      WRITE(6,*) 'square root of negative is:',a

      d = b/c
      WRITE(6,*) 'attempt at overflow is:',d

      d = c/b
      WRITE(6,*) 'attempt at underflow is:',d

      END
```

Here are the results of running this program:

```
1.000000E+80   1.000000E-80
square root of negative is:        NaN
attempt at overflow is:            INF
attempt at underflow is: 0.0000000E+00
```

Here are some interesting computational results.

INF (infinity, positive or negative) is produced by:

Division by zero

Floating point overflow

NaN is produced by:

Input is NaN

Input in incorrect format for floating point

Indeterminate form (for example: INF-INF, INF/INF, INF**–INF, 0./0.)

Non-existent answer (for example: square root of a negative number, logarithm of a negative, arcsin(2.0), dmod(1.0d0,0.0d0))

Trapping Arithmetic Exceptions

As noted above, the default situation for IEEE format handling of exception conditions is to silently insert NaN, INF or Zero as the result of arithmetic calculations. However, you can override the default operation, and actually cause your program to exit with an error on occurence of any of the following conditions:

- Input operand is NaN
- Divide by zero
- Invalid input operand
- Floating point overflow
- Floating point underflow
- Inexact results

To change the behavior of the program to issue an error when each of the above conditions occurs, use a built in function to set a mask that looks for those conditions. Each mask bit has a specific function that sets or clears it. The name of the function relates directly to the bit it affects.

```
SET_ALL_MASKS                 CLEAR_ALL_MASKS
SET_INPUT_IS_NAN              CLEAR_INPUT_IS_NAN
SET_DIVIDE_BY_ZERO            CLEAR_DIVIDE_BY_ZERO
SET_INVALID_OPERAND           CLEAR_INVALID_OPERAND
SET_ARITHMETIC_OVERFLOW       CLEAR_ARITHMETIC_OVERFLOW
SET_ARITHMETIC_UNDERFLOW      CLEAR_ARITHMETIC_UNDERFLOW
SET_ARITHMETIC_INEXACT        CLEAR_ARITHMETIC_INEXACT
```

If an event matching the mask occurs while the mask is SET, a signal is sent, causing the termination of the program. If an event occurs while the mask is CLEAR, the program takes the IEEE-specified action and continues execution.

If you set all masks, your program will terminate in the first library routine you call that does integer to floating point conversions as a result of the arithmetic

NOTE
INEXACT exceptions arise **very** often.

```
CALL SET_ALL_MASKS
CALL CLEAR_ARITHMETIC_INEXACT
```

If a flag is not set, the results are not necessarily *correct* and a user may want to develop a function that checks the validity of the results. For example, to check whether a single-precision number is a NaN, you can use the function "isnanf(x)", a macro function available to C programmers as a macro contained in */usr/include/ieeefp.h*. Fortran programmers could use the same function by writing a simple Fortran to C interface.

Performance By Data Type

To obtain the highest performance from the Stardent 1500/3000, it is more efficient to use REAL*4 or REAL*8 or INTEGER*4 (in C, float, double, or long values) than to use BYTE, INTEGER*2 (in C, char, or short values). Single-precision or double-precision floating point equations or equations using long integers can often be vectorized (use the vector unit) and can run as much as 10 times as fast as the same equations using other data types. As examples, timing experiments were run using the Stardent 1500/3000 Fortran compiler, version 2.0, and the following results were obtained.

For the sum of 900,000 elements of a given data type, under various optimization levels (level 0 is no optimization, level 2 is vectorization) the following performance was obtained:

Data Type	Opt Level 0	Opt Level 2
BYTE	1.77 Secs	1.71 Secs
INTEGER*2	1.94 Secs	1.83 Secs
INTEGER*4	0.89 Secs	0.20 Secs
REAL*4	0.65 Secs	0.19 Secs
REAL*8	0.67 Secs	0.16 Secs

Notes. BYTE and INTEGER*2 operations do not vectorize, but improve slightly due to scheduling of operations done at optimization levels 1 and 2.

For squaring all of the elements of a 900,000-element matrix of a given data type:

Data Type	Opt Level 0	Opt Level 2
BYTE	3.29 Secs	3.27 Secs
INTEGER*2	3.16 Secs	3.12 Secs
INTEGER*4	1.71 Secs	1.65 Secs
REAL*4	1.90 Secs	0.23 Secs
REAL*8	1.94 Secs	0.28 Secs

BYTE, INTEGER*2 and INTEGER*4 multiplication is not implemented in the vector unit. BYTE and INTEGER*2 suffer in performance to to the extra time needed for off-alignment memory access time.

Cray Directives

Directives for the Cray compiler have existed for roughly a decade. Because their functionality is very similar to Stardent 1500/3000's directives, the Stardent 1500/3000 compilers accept Cray directives on a switched on or off mode, as in the Cray compiler. The following directives have been added to the Stardent 1500/3000 compilers solely for compatibility with Cray directives.

CDIR$ IVDEP

The **IVDEP** directive is a Cray directive specifying that the compiler should ignore vector dependences. This directive does not guarantee vector execution.

CDIR$ NORECURRENCE = N

The **NORECURRENCE** directive is a Cray directive inhibiting generation of vector recurrence code for loops that are below a given size. It is similar to the **NOVECTOR** mode.

If you are coding for the Stardent 1500/3000, **VBEST** and **ASIS** are better directives to use.

CDIR$ NOVECTOR = N

The **NOVECTOR** directive is a Cray directive inhibiting vectorization of loops below a given size.

If you are coding for the Stardent 1500/3000, **VBEST** and **ASIS** are better alternative directives.

CDIR$ VECTOR

The **VECTOR** directive is a Cray directive toggling **NORECURRENCE** and **NOVECTOR** between default and specified values.

Stardent Fortran Syntax

In this section, specific differences are highlighted, but not necessarily described in full detail. Please refer to the *Fortran Reference Manual* for complete details.

- Stardent Fortran is syntax compatible with DEC VAX/VMS Fortran version 3.0 (prior to addition of *structures*). VMS RTL instructions are not provided — functions equivalent to the RTL instructions are supported through standard Unix libraries.
- Stardent Fortran complies witth ANSI Fortran 77 (certain extensions are provided from ANSI Fortran 77).
- Stardent Fortran supports Unix library functions (malloc, drand, and so on) in both System V and BSD 4.3 versions.
- Stardent Fortran supports CRAY-style vectorization and parallelization directives.
- Stardent Fortran supports MIL-STD-1753 extensions (such as INCLUDE 'file').

Fortran 77 Extensions

The following features are extensions to the Fortran 77 implementation:

- Additional data types are defined.
- Arithmetic with logical expressions is available.
  - Numeric data can include logical data items (logical data items are treated as integers).
  - Real operations can be performed on combinations of real, integer and logical data elements.

  - Logical operations can include integer elements with an integer data type as a result.
- I/O options
  - NAMELIST directed I/O is allowed.
  - OPEN statement has extentions NAME, TYPE, and CARRIAGECONTROL.
  - INQUIRE statement permits use of DEFAULTFILE = dfile.
- Format statement descriptors.
  - Octal, hexadecimal, and blanks are permitted in format codes.
  - Implicit widths.
  - Expressions are allowed in formats.
- Flexibility in input source format.
  - More comment conventions are provided (use of '!').
  - Long variable names are allowed (32 characters).
  - Tab formatting may be performed.
- Options to control compilation and execution.
- Extended functions and statements.
  - DO WHILE is recognized (but note that to vectorize, the compiler will try to turn this construct into a standard DO statement).
  - END DO is recognized (allows loops without statement labels).
  - Built-in functions are provided (such as %VAL, %REF and %LOC, which are used to communicate with sub-programs in other languages).
  - AREAD, ABLOCK, ASTATUS are provided for asyn-chronous I/O plus access to standard System V and BSD I/O libraries for other unusual I/O facilities.
  - COMMON blocks can be data loaded into any routines. (That is, BLOCK DATA subprograms are not required.)

Common Porting Questions

Here are a few questions that seem to come up somewhat often in porting classes Stardent has given.

About DCL Define

In VAX/VMS, I can name the file later by using the DCL DEFINE service or the command ASSIGN. Can I do the same thing on the Stardent 1500/3000?

YES. We use the Unix environment variables in place of the logical names used by DCL.

```
      CHARACTER*256 FILENM
      CALL GETENV('CSFSAVE',FILENM)
      OPEN(UNIT=NFT17, FILE=FILENM, STATUS='UNKNOWN',
     *     FORM='UNFORMATTED', ACCESS='SEQUENTIAL')
```

NOTE
This question and answer must be expanded to show 'setenv' and a shell script to use this feature... in other words, the example should be more complete.

About Async I/O

I want to use Asynchronous READ to compute while the system is READing from disk.

Be aware that Asynchronous READ works only on files that have been opened with ACCESS = 'ASYNC'. Asynchronous I/O requests are limited to a block length that is a multiple of 4096 bytes. Example:

```
CHARACTER*1 buffer(4096)
INTEGER unit, blockno, nbytes, ablock
LOGICAL ast, astatus

CALL AREAD(unit, buffer, blockno, nbytes)
...(computation continues)...
ast = astatus(unit) !true if read complete, false if pending
if(.not.ast) icode=ablock(unit) !wait I/O complete
```

Appendix A of this manual is a complete description of Asynchronous I/O as implemented on the Stardent 1500/3000.

About Polling Devices

I would like to be able to test whether or not there is input on a device, and read what is there if and only if something has changed (for example, the mouse, keyboard, dials and so on).

There are two different methods for performing device I/O: by polling the devices, and by responding to a signal. If you have installed the */opt/examples* directory on your Stardent 1500/3000, you'll find a complete example source code for signal driven I/O (in this case, to the knobs) in the file */opt/examples/utils/knobs/knobs.c*.

Signals are the software equivalent of interrupts. Where a polling program sits in a loop, continuously testing the status of some bit to determine whether some input is pending, a signal-driven program presents a considerably lower load to the system. In effect, a signal-driven program gives its time resources to the system, allowing other programs to run while this signal-driven program sleeps, waiting for the signal to be received. Because the sample program, *knobs.c*, is rather long, it has not been repeated here. See the sample source for more information.

Polled device I/O is performed by using either the System V *poll* function, or by the BSD *select* service. Here is an example in C.

```
#include <stdio.h>
#include <poll.h>
#include <fcntl.h>

main()
{
     unsigned char buffer[1024];
     struct pollfd Pollfd;

     Pollfd.fd     = open("/dev/dials",O_RDWR);
     Pollfd.events = POLLIN;

     while(1) {
        poll(&Pollfd,1,0);
        if(Pollfd.revents == 0) {
            printf("*"); fflush(stdout);
        }
        else {
            read(Pollfd.fd,buffer,3);
            printf("\n");
            printf("%2x,%2x,%2x\n",
                    buffer[0],buffer[1],buffer[2]);
        }
     }
}
```

In Fortran, a dummy C routine is used to access the required service. Here is the example of polling two devices in Fortran, along with the supporting C routines.

```
      INTEGER C_OPEN

      INTEGER O_RDWR
      DATA    O_RDWR /2/
      INTEGER POLLIN
      DATA    POLLIN /1/

      INTEGER*2   DUMMY(100)
      INTEGER     FD_DIALS,        FD_MOUSE
      INTEGER*2   EVENTS_DIALS,    EVENTS_MOUSE
      INTEGER*2   REVENTS_DIALS,   REVENTS_MOUSE
      EQUIVALENCE (FD_DIALS,       DUMMY(1))
      EQUIVALENCE (EVENTS_DIALS,   DUMMY(3))
      EQUIVALENCE (REVENTS_DIALS,  DUMMY(4))
      EQUIVALENCE (FD_MOUSE,       DUMMY(5))
      EQUIVALENCE (EVENTS_MOUSE,   DUMMY(7))
      EQUIVALENCE (REVENTS_MOUSE,  DUMMY(8))

      BYTE BUFFER(1024)

      FD_DIALS     = C_OPEN('/dev/dials',O_RDWR,'rb')
      EVENTS_DIALS = POLLIN
      FD_MOUSE     = C_OPEN('/dev/mouse',O_RDWR,'rb')

100   CONTINUE

      CALL C_POLL(DUMMY,2,0)

      IF(REVENTS_DIALS.NE.0) THEN
         CALL C_READ(FD_DIALS,BUFFER,3)
         WRITE(*,200) BUFFER(1),BUFFER(2),BUFFER(3)
200      FORMAT('Dials: ',3Z3)
      ENDIF
      IF(REVENTS_MOUSE.NE.0) THEN
         CALL C_READ(FD_MOUSE,BUFFER,3)
         WRITE(*,300) BUFFER(1),BUFFER(2),BUFFER(3)
300      FORMAT('Mouse: ',3Z3)
      ENDIF

      GOTO 100
      END
```

The C programs that support the Fortran polling example as follows:

```
int C_OPEN(path, oflag, mode)
int *path, *oflag, *mode;
{
    return(open(*path,*oflag,*mode));
}

void C_POLL(fds,nfds,timeout)
short *fds;
unsigned long *nfds;
int *timeout;
```

```
{
    poll(fds,*nfds,*timeout);
}

void C_READ(filedes, buf, nbytes)
int filedes, *buf, nbytes;
{
    read(filedes,*buf,nbytes);
}
```

About Interfacing C and Fortran

Can you explain the general issues involved in having a C program call a Fortran program or vice versa?

You'll find an entire section that deals with language interfacing in Chapter 10. It includes details about how parameter passing is performed in each language, as well as details about how Fortran strings are stored.

Inline Expansion

What kind of support is provided for inlining code?

When the Stardent 1500/3000 compiler cannot gather enough information to determine whether a loop can safely vectorize or parallelize, it assumes the worst possible case and processes the loop sequentially. Procedure calls, by their data abstraction, hide vital information from the compiler. They cannot be executed on a vector unit. The following loop does not vectorize because of the lack of information about the effects of the call and because of the procedure call itself.

```
      DO 120 I = 1, N
          A(I) = HYPOT(B(I), C(I))
120   CONTINUE
          .
          .
          .
      FUNCTION HYPOT(A,B)
          HYPOT = A*A + B*B
      RETURN
```

The Stardent 1500/3000 compiler contains options which allow the user to cause procedures such as this to be substituted *inline* automatically. For instance, when the Stardent 1500/3000 compiler is told to inline HYPOT, the resulting vreport is as follows:

```
DO iv=1, N, 32
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      tv_in_a_8(2 + I - iv) = B(I)
      tv_in_b_7(2 + I - iv) = C(I)
      tv_return_type_6(2 + I - iv) = tv_in_a_8(2 + I- iv) * tv_in_a_8(2 + I - iv) +
                                     tv_in_b_7(2 + I - iv) * tv_in_b_7(2 + I - iv)
      A(I) = tv_return_type_6(2 + I - iv)
   END DO
   IF (rv .EQ. N) THEN
      Return_Type_6 = tv_return_type_6(2 + rv - iv)
   END IF
END DO
```

The report is a little difficult to read, because the compiler creates a number of temporaries to handle the inlining process correctly. (All variables which have "in_" in them were created for inlining.) But the loop has now vectorized, and runs much faster than with the procedure call in it. This optimization allows users to use lots of small functions (which generally creates code that is easier to read and maintain), but still get the efficiency of having large procedures.

There are two ways to invoke the inlining facilities of the Stardent 1500/3000 compiler:

(1) In the simplest form, you merely need to add the option **-inline** to the command line. When inlining is enabled, the Stardent 1500/3000 automatically inlines any functions found in the same source file that appear to be good candidates for inlining. "Good candidates" include very short functions which have no other function calls or input-output statements as well as functions specified in INLINE directives.

(2) A mechanism that gives the user more control over the inlining process is the use of an inline catalog. Under this mechanism, the user can create a database or catalog of functions that he wishes to inline frequently. Thereafter, he can refer to that catalog in a **-Npaths=** option on the command line, and the compiler automatically inlines all functions found in that catalog.

The HYPOT example above illustrates the first type of inlining—if HYPOT and the main function are both in the same file, compiling with **-inline** is enough to automatically inline HYPOT. To use the same example to illustrate the second method, assume that the function HYPOT is in a file *hypot.f* and the main procedure is in a

file *main.f*. The first step is to put HYPOT into a catalog, which is accomplished by using the **-catalog** option on the compiler. By default, the suffix for inline catalogs is assumed to be ".in", so *hypot.in* is a reasonable name for the catalog. The following command creates the catalog:

```
fc -c -catalog=hypot.in hypot.f
```

Note that this command actually creates two files *hypot.in* and also *hypot.id*. In general, every ".in" file has an associated ".id" file; neither of these files works without the other. At this point, HYPOT can be inlined in any file by compiling that file with the **-inline** option and specifying hypot.in on the **-Npaths** option:

```
fc -c -inline -Npaths=hypot.in main.f
```

The compiler prints out warning messages whenever it inlines files, letting you know the origination of the inlined file.

There are a number of important points regarding inline catalogs:

(1) It generally does not matter what optimization level is used in the creation of the catalog, because the inlining process occurs early in a compilation, long before any other optimizations are done. As a result, inlined code gets fully optimized at the time it is inlined; there is little value in optimizing it twice.

(2) The **-catalog** option adds to an existing catalog as well as create a new one. When inlining, the compiler takes that *last* entry when multiple entries exist for a routine.

(3) Both files associated with a catalog must be present for the catalog to work. If either is missing, the compiler does not inline. Also, notice that the use of the two characters "in" and the 14 character filename limit in System V may conflict if long file names are used, causing the ".in" and ".id" files to overwrite themselves.

(4) Most of the time, the object file created during the creation of a catalog can be thrown away. However, in a few instances, it cannot. If the function to be inlined contains initialized data or a static variable whose value is to be retained from call to call, the compiler generates an external name for that variable, and uses it in the inlined code. In such cases, the object file used to create the catalog must be linked at load time, because it contains the definition for that variable. To illustrate with a concrete example,

consider the following sample random number generator:

```
function rand()
data seed /314159/
seed = seed * 27818
rand = seed
return
end
```

Since a user of this function wants the variable *seed* modified at every call, regardless of whether it is inlined or not, the compiler creates an external variable with a unique name to replace it, both in the inlined version and in the object. The name of this external variable is based from the procedure name, to avoid the possibility of accidentally creating the same variable in two different files. For instance, the name created for *seed* above is

```
$$IE_RAND_$$IE__lcl_Block_SEED_$$1
```

The definition of this variable is in the object created with the catalog. As a result, if **RAND** is inlined from this catalog, the resulting object does not link without the rand.o created in the catalog step, because $$IE_RAND_$$IE__lcl_Block_SEED_$$1 is undefined.

(5) Whenever you receive new compiler versions, it may be necessary to recreate your catalogs using the new compiler. Occassional changes to the intermediate representation used in the compiler may invalidate existing catalogs. Note that the compiler clearly warns you when this occurs; it does not inline from an existing catalog if that catalog is out of date.

(6) By default, the **-inline** option uses a Stardent supplied library of inline functions—*/usr/lib/libbF77.in*. This default can be overridden by specifying a blank Npaths, that is, **-Npaths=**, which acts the same way as the **-I** option acts with the *cpp*. A list of the functions provided in the the system inline catalog is provided in the following section, called *Inline Functions*.

(7) As long as the calling conventions are respected, Fortran functions can be correctly inlined in C and C functions can be correctly inlined in Fortran.

(8) There are a small number of function types that cannot be inlined. The compiler does not inline character valued functions, functions that have Fortran character strings as parameters, or C vararg functions. It also does not inline a function into the while condition of a **WHILE** loop.

Inline Functions

The best functions to inline are small functions that are often used in loops. The Stardent 1500/3000 compiling system includes a sample library of inline functions that illustrate both the utility and the power of inlining in the Stardent 1500/3000 compiler. This library is comprised of the BLAS (Basic Linear Algebra Subroutines)—a set of vector linear algebra routines.

NOTE
The inline functions library is located in */usr/lib/libbF77.in,* and also in compiled form in */usr/lib/libbF77.a.*

These functions are automatically candidates for inlining whenever **-inline** is used. The following table lists all the routines in the BLAS inline library.

Table 11-1. BLAS Function Names

caxpy	dcopy	isamax	sscal
ccopy	ddot	izamax	sswap
cdotc	dmach*	sasum	zaxpy
cdotu	dnrm2	saxpy	zcopy
cmach*	drot	scasum	zdotc
crotg	drotg	scnrm2	zdotu
cscal	dscal	scopy	zdrot
csrot	dswap	sdot	zdscal
csscal	dzasum	smach*	zmach*
cswap	dznrm2	snrm2	zrotg
dasum	icamax	srot	zscal
daxpy	idamax	srotg	zswap

The first letter of any function indicates the data type it works on—"i" indicates integer, "s" indicates single precision, "d" indicates double precision, "c" indicates complex, and "z" indicates double complex. The rest of the name indicates the function provided. Following is an argument description of the single

* This function is not required by Linpack proper. It is the Stardent 1500/3000 function proper used in testing only.

precision routines—the analagous routines in other precisions have the same arguments (but with different types) and perform the same function. If you need additional information concerning these inline functions (for example, to determine how they are derived, and so on), please refer to the *LINPACK User's Guide*.

Note that there are also some limitations within the Stardent 1500/3000 Fortran. This means that some cases can not be inlined, and those are: character functions, a function that takes character variables, and the while condition of a **WHILE** loop. The following section shows their syntax and descriptions. Only the single precision functions are defined here. The only difference between the single precision functions and the double, complex, and double complex ones are the data types that they accept as parameters and the data type of the return value.

ISAMAX

```
integer function isamax(n,x,incx)
integer n,incx
real x(1)
```

Function: finds the largest component of a vector. If $n \le 0$, the function sets the result to zero and returns immediately. *isamax.f* a real type function.

SASUM

```
real function sasum(n,x,incx)
integer n,incx
real x(1)
```

Function: computes the sum of magnitudes of vector components. This function takes summation only on the absolute values of the vector components when n > 0. If $n \le 0$, the function returns immediately. *sasum.f* is a real type function.

SAXPY

```
subroutine saxpy(n,a,x,incx,y,incy)
integer n,incx,incy
real x(1),y(1),a
```

Subroutine: performs an elementary vector operation such as y = ax + y. If a = 0 or if $n \le 0$, this subroutine returns immediately. *saxpy.f* is a real type subroutine.

SCASUM

```
complex function scasum(n,x,incx)
integer n,incx
complex x(*)
```

Function: computes the sum of magnitudes of vector components. This function takes summation on both the real and imaginary parts of the vector components when n > 0. If $n \leq 0$, the function returns immediately. *scasum.f* is a real type function.

SCNRM2

```
complex function scnrm2(n,x,incx)
integer n,incx
complex x(*)
```

Function: computes the 2-Norm (Euclidean length) of a vector. This function uses both the absolute values of the real and imaginary parts of the vector component to do its its computation and when n > 0. If $n \leq 0$, the function returns immediately. *scnrm2.f* is a real type function.

SCOPY

```
subroutine scopy(n,x,incx,y,incy)
integer n,incx,incy
real x(1),y(1)
```

Subroutine: copies a vector x component into y component. If $n \leq 0$, the subroutine returns immediately. *scopy.f* is a real type subroutine.

SDOT

```
subroutine sdot(n,x,incx,y,incy)
integer n,incx,incy
real x(1),y(1)
```

Function: computes the Dot Product of a vector. If $n \leq 0$, the result of the Dot Product is zero. *sdot.f* is a real type function.

SMACH

```
real function smach(job)
integer job
```

job is 1, 2, or 3 for *epsilon, tiny, or huge*, respectively.

If *epsilon* then return value is 1.19209e-07
If *tiny* then return value is 9.8608e-30
If *huge* then return value is 4.0565e+29

Function: computes machine parameters of floating point arithmetic for use in testing only. It is not required by Linpack proper. *smach.f* is a real type function.

SNRM2

```
real function snrm2(n,x,incx)
integer n,incx
real x(1)
```

Function: computes the 2-Norm (Euclidean Length) of a vector. This function uses only the absolute value of the vector component to do its computation and when n > 0. If $n \leq 0$, the function returns immediately. *snrm2.f* is a real type function.

SROT

```
subroutine srot(n,x,incx,y,incy,c,s)
integer n,incx,incy
real x(1),y(1),c,s
```

Subroutine: applies a plane rotation on vector components. The subroutine returns immediately if $n \leq 0$ or if c = 1 and s = 0. *srot.f* is a real type subroutine.

SROTG

```
subroutine srotg(a,b,c,s)
real a,b,c,s
```

Subroutine: constructs Givens Plane Rotation. *srotg.f* is a real type function.

SSCAL

```
subroutine sscal(n,a,x,incx)
integer n,incx
real a,x(1)
```

Subroutine: performs a vector scaling (multiply a vector with a constant). The subroutine returns immediately if $n \leq 0$. *sscal.f* is a real type subroutine.

SSWAP

```
subroutine sswap(n,x,incx,y,incy)
integer n,incx,incy
complex x(1),y(1)
```

Subroutine: interchanges vector x components with y components. If $n \leq 0$, the subroutine returns immediately. *sswap.f* is a real type subroutine.

About FFT Support

What kind of support is provided for FFT's?

The following shows the support for FFT's (Fast Fourier Transforms). Both one dimensional and two dimensional

FFT's are supported.

One Dimensional FFT

```
SUBROUTINE CFFTN(IS, X, Y, M)

SUBROUTINE DCFFTN(IS, X, Y, M)
```

For Complex FFT (CFFTN) or Double Complex FFT (DCFFTN), the parameters take on the following meanings:

when IS = 0: Initialize transform routine for FFT's of size 2**M. Call this only once unless changing M.

when IS = 1: Forward FFT, positive argument for roots of unity

when IS = -1: Inverse FFT, negative argument for roots of unity

Source is COMPLEX*n in X array.

Destination is COMPLEX*n in Y array.

Source and destination may not overlay.

Two Dimensional FFT

```
SUBROUTINE CFFTN2D(IS, X, Y, M, STRIDE)

SUBROUTINE DCFFTN2D(IS, X, Y, M, STRIDE)
```

For Complex FFT (CFFTN) or Double Complex FFT (DCFFTN), the parameters take on the following meanings:

> when IS = 0: Initialize transform routine for FFT's of size 2**M or less. Call this only once unless changing M.
>
> when IS = 1: Forward FFT, positive argument for roots of unity
>
> when IS = -1: Inverse FFT, negative argument for roots of unity.

Source and destination is COMPLEX*n in X array.

STRIDE: First dimension of X. That is, X is declared as COMPLEX**n* X(STRIDE, *)

About Performance

What about performance issues?

If a program code has already been tailored to a specific machine, to get better performance on the Stardent 1500/3000, it might be necessary to retune the code to allow the compiler to do its job. For example, there might be some cases where code has been rearranged to get the best performance on a non-vector machine, placing one kind of operation immediately following another specific kind of operation where it is known on that specific machine that this allows the resulting code to be pipelined. Sometimes, however, this very kind of construct could potentially prevent vectorization from happening.

Use the tools described in chapter 9, *Tuning Code*, chapter 7, *Efficient Programming Techniques* and the general information in chapter 6, *Vector and Parallel Optimization* and chapter 8, *Explicit Parallel Programming* to make it possible for your program to run most efficiently on the Stardent 1500/3000.

LIBRARY FUNCTIONS

CHAPTER TWELVE

This chapter briefly describes the functions in the Fortran support library. Man-pages are available for each of these functions. To access the man-page on a Stardent 1500/3000, issue the command:

```
man 3f <function>
```

where <function> represents the name of the function as given in the first column of any of the tables shown below.

The Fortran support library contains the following kinds of functions:

- Asynchronous Read/Write
- Command-line argument inquiry
- Commands for date and time
- Filing System control commands
- File access commands
- Random number generation
- Miscellaneous functions
- System interface functions

Asynchronous I/O

ablock	wait for asynchronous read to complete
aread	asynchronous read
astatus	check status of an asynchronous read

The functions for Asynchronous I/O are fully described in *Appendix A* of this manual.

Access To Command Line Arguments

iargc	Return the number of command line arguments
getarg	Return Fortran command-line argument

Date And Time

fdate	Return date and time in an ASCII string
idate, itime	Return date or time in numerical form
etime, dtime	Return elapsed execution time
time, ctime, ltime, gmtime	Return system time
cputim, systim, fputim	Perform timing functions
mclock	Return Fortran time accounting

Filing System Control

access	Determine accessibility of a file
chdir	Change default directory
chmod	Change mode of a file
getcwd	Get pathname of current working directory
link	Make a link to an existing file
rename	Rename a file
stat, lstat, fstat	Get file status
unlink	Remove a directory entry

Random Number Generation

drand, rand, irand	Return random values
rand, irand, srand	Generate random numbers
drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48	Generate uniformly distributed pseudo-random numbers

File I/O

fseek, ftell	Reposition a file on a logical unit
getc, fgetc	Get a character from a logical unit
putc, fputc	Write character to fortran logical unit
topen,tclose, tread,twrite, trewin,tskipf, tstate	Perform Fortran tape I/O
flush	Flush output to a logical unit

Miscellaneous Functions

loc	Return the address of an object
long, short	Integer object conversion
qsort	Quick sort
rindex, lnblnk	Tell about character objects
flmin, flmax, ffrac, dflmin, dflmax, dffrac, inmax	Return extreme values

System Interface Functions

abort	Terminate Fortran program
exit	Terminate process with status
exception	Arithmetic exception control routines
alarm	Execute a subroutine after a specified time
fork	Create a copy of this process
getenv	Return Fortran environment variable
gethostid	Get host processor identification number
getlog	Get user's login name
getpid	Get process ID
getuid, getgid	Get user or group ID of the caller
hostnm	Get name of current host
malloc, free, falloc	Dynamic memory allocator
kill	Send a signal to a process
perror, gerror, ierrno	Get system error messages
signal	Specify Fortran action on receipt of a system signal
system	Issue a shell command from Fortran
sleep	Suspend execution for an interval
ttynam, isatty	Find name of a terminal port

ASYNCHRONOUS INPUT/OUTPUT

APPENDIX A

This appendix discusses asynchronous input/output file handling. The overview provides a functional definition and discusses the major differences between asynchronous I/O and other input/output methods. The remaining sections describe the system calls available in Fortran and C and include coding examples for each language.

Overview

A normal pattern for input/output in a program is to read data, perform computations, and then to write data. Under normal UNIX, as soon as a write has passed through the operating system, the computations may continue. In the case of a read, under normal UNIX, the read does not return until the data is actually transferred into the program from the disk. To summarize, writing and processing are overlapped and reading and processing are not overlapped under normal UNIX input/output.

Asynchronous I/O allows reading to be overlapped with processing and writing. A very general scheme is illustrated as follows: the first input is read in and when completed, the read for the second input begins. The processing for the first input now starts followed by the write for the first input. This is then followed by the read for the third input, the processing for the second input, and so on. This cycle continues always reading one input ahead.

The major usage difference with asynchronous I/O is that the user must check to see if the I/O is complete before using the data.

There are benefits to using asynchronous I/O, but there are also several cautions that should be noted. The primary benefit when using asynchronous I/O is that the CPU is used while disk transfer is occurring. A secondary benefit is that the data is transferred directly from disk to data storage with no intervening buffering. Thus, there is one less copy of the data and correspondingly less CPU effort required.

The following cautions should be noted when using asynchronous I/O. You must always check on the completion of the I/O operations. You must observe the restrictions about block size and any other restrictions that are mentioned in the *Commands Reference Manual* for the specific system calls. Finally, be aware that asynchronous I/O is not a portable feature and it is probably wise to place this code in an isolated subroutine.

Asynchronous System Calls for C

There are three system calls that are available for asynchronous I/O and these allow you to perform an asynchronous read, obtain the status of the file read, and wait for the read to complete. The calls are **read**, (after opening the file with **O_BASYNC** specified), **astat**, and **await**. These calls are explained in detail, with their syntax, in the *Commands Reference Manual.*

EXAMPLE

```
/*
 * The following section of code illustrates how
 * the asynchronous I/O capability is used.  The
 * open() call specifies that the file is to be
 * accessed asynchronously.  The file must be read
 * in multiples of 4 bytes at a time, and the buffer
 * and offset must be aligned on sector (1024 byte)
 * boundaries.  The read call on the first iteration
 * returns immediately, but read calls in subsequent
 * iterations will block until the previous call completes.
*/
fd = open("file",  O_BASYNC);
if (fd < 0) {
        fprintf(stderr, "cannot open %s\n", argv[1]);
        exit(1);
}
n = 0;
cp = buf;
while ((i = read(fd, cp, 4096)) > 0) {
        n += i;
        cp += i;
}

/*
 * The astat call enquires the status of the outstanding
 * (in this case, the last) read call.
*/

if (astat(fd) == 1)
        fprintf(stderr, "last read completed\n");
else {
        fprintf(stderr, "last read still pending\n");
```

```
/*
* await blocks execution until the read completes.
*/
        nbytesread = await(fd);
}

if (i < 0) {
        fprintf(stderr, "read error %d\n", errno);
        exit(1);
}
close(fd);
```

Asynchronous Library Functions for Fortran

There are three Fortran library functions that are available for asynchronous I/O and these allow you to perform an asynchronous read, obtain the status of the file read, and wait for the read to complete. These functions are explained in detail, with their syntax, in *Chapter 6, Fortran Language Library Functions* in the *Fortran Reference Manual.*

The section that follows illustrates a very basic example of how to code **aread, astatus**, and **ablock**. Notice that the record length of the file is a multiple of 4096 bytes and that the number of bytes to be read by **aread** is a multiple of 4096 bytes.

EXAMPLE

```
      CHARACTER*1  BUFF(4096)
      EXTERNAL     ABLOCK, AREAD, ASTATUS
      LOGICAL      ASTATUS, DONE
      INTEGER      ABLOCK, AREAD
      INTEGER      STAT

C     Open the file with asynchronous access.

      OPEN(2, FILE='data', ACCESS='async')
      CALL AREAD (BUFF, 0, 4096)
      DONE = ASTATUS(2)
      IF (DONE) THEN
                PRINT *, 'read finished successfully'
      ELSE
                PRINT *, 'read transfer not done, going to ablock'
                STAT = ABLOCK(2)
                PRINT *, 'ablock read', STAT
      ENDIF

C     buff is now filled with data

      END
```

USING THE POSTLOADER

APPENDIX B

Executable files compiled on a Stardent 1500 system will not execute directly on a Stardent 3000 system. The executable loader programs for the Stardent 3000 check the executable file before loading; if it is not 3000 compatible, the loader refuses to load them and suggests conversion or recompilation.

Release 3.0 of the Stardent 1500/3000 system software contains a new command *adapt* for converting executables made on a Stardent 1500 systems to run on Stardent 3000 systems. The method of executing this command is

```
% adapt 1500_a.out 3000_a.out
```

1500_a.out is the name of an executable made on a Stardent 1500 system, and *3000_a.out* will be the name of a newly created executable that will run on a Stardent 3000.

adapt should correctly handle any executable made from compiled code, although some unusual anomalies may appear. For instance, when the Fortran I/O library detects a runtime error such as a segmentation violation, it checks the version of the executable to generate a stack backtrace. Since Stardent 3000 executables have a different version than Stardent 1500 executables, and since postloaded executables will have the 3000 version number on the executable and a 1500 version number hidden away inside the Fortran library itself, the segmentation violation will produce an internal Fortran I/O error message, rather than the stack traceback that is generated on Stardent 1500. Likewise, attempts to postload commands such as *ld* that check executable version numbers will not succeed.

It is possible to create assembler programs that *adapt* cannot correctly postload. Doing so, however, requires very intimiate knowledge of the Stardent 1500/3000 hardware and instruction set; it is Stardent's expectation that every Stardent 1500 user program should be postloadable to run on Stardent 3000 hardware.

While *adapt* exists to allow users to convert Stardent 1500 executables for Stardent 3000 execution, Stardent strongly recommends that users recompile source where possible. *adapt* should permit correct execution, and the resulting executable should run faster on Stardent 3000 hardware than the original on a Stardent 1500. However, even faster execution should result by recompiling on the 3000.

INDEX

-c **5**:39
-cpp **2**:3
-double_precision **7**:20
-E **2**:3
-e **2**:3
-fast **7**:29; **9**:8–9
-g **4**:1; **5**:2–2, 5, 18, 39
-L **3**:1, 3–4
-lcurses **3**:3
-O **5**:39
-o **5**:39
-O0 **5**:5; **11**:4
-O2 **7**:2, 10; **9**:3, 5, 8
-O3 **7**:26; **8**:20, 23; **9**:3, 5, 18
-P **2**:3–14, 17
-p **2**:3–14, 17
-ploop **9**:2, 4–5, 7–9
-S **2**:2, 9
-s **2**:2, 9
-T **9**:9–10
-t **9**:9–10
-V **9**:3, 15–16
-v **9**:3, 15–16
-vreport **6**:9; **7**:2, 10; **8**:23; **9**:9–10

A

abbreviation **1**:9; **5**:24, 26
ablock **12**:1
accumulator **5**:36; **10**:2
accumulator registers **10**:2
adapt command **B**:1
alignment **7**:30; **11**:1–4
  data **7**:31; **11**:2, 4

ar **3:**1–3
archive **3:**1–4
archive libraries **3:**1–2, 4
aread **12:**1; **A:**3
ASIS **7:**6; **8:**18; **9:**11–12
assembler **7:**9; **10:**1
astat **A:**2
astatus **12:**1; **A:**3
asynchronous **12:**1; **A:**1–3
await **A:**2

B

backslash **5:**15, 27
banks **7:**21; **10:**2
breakpoint **5:**1–4, 11–12, 17, 21–24, 40–42, 44
breakpoints **4:**3; **5:**1, 9, 11, 17, 21–24, 32–33
BSD **3:**3; **4:**1

C

C compiler **7:**2, 15, 28; **9:**11
C$DOIT **8:**7, 10, 18, 21; **9:**11–19
calculations **7:**21; **8:**3; **11:**5–6
CALL **2:**1; **5:**4, 12–14, 25, 40; **6:**15; **7:**10, 16; **8:**6–7, 14–17, 19–25, 27; **9:**3, 6, 8, 16; **11:**4, 7
callee **10:**1–2
caller **10:**1; **11:**4; **12:**4–2
calling subprograms **10:**1
CANCEL **5:**11, 24, 35
case-sensitive **1:**4; **2:**2
CATCH **5:**12–13, 32
cc **3:**1, 3; **5:**39
code, dead **6:**9
column-major format **7:**21
comma-separated list **5:**9
command language **5:**13, 27
command line options **2:**2; **3:**2
command,
  adapt **B:**1
  compound **5:**13, 15
  keyword **5:**9, 15, 26
Comments **5:**2, 29; **9:**10
COMMON **7:**2, 7–9, 16–18, 22, 25, 28; **8:**5–29; **9:**9, 13, 17; **11:**2
compilation control **2:**2

compilation control statements **2**:2
compile **2**:1; **5**:5, 18; **6**:8; **7**:10; **8**:7, 17–18, 23–24, 27; **9**:1–2, 5, 8
compiler **2**:1; **3**:3; **5**:2–3, 5, 43; **6**:1, 3; **7**:1–11, 13–15, 19–20, 22; **8**:1–30, 3–5, 7–8, 10–13, 18–20, 22–24, 26; **9**:1–2, 4, 7–8, 10; **10**:2; **11**:1–3, 7
compiler directives **2**:2; **9**:1, 10; **11**:1–11
compiler options **2**:1; **9**:2–2
compiler techniques **6**:6
C **7**:2, 15, 28; **9**:11
Fortran **3**:3; **6**:9; **8**:5–11, 7, 19; **9**:17; **11**:7
compiling **4**:1; **7**:26; **8**:7, 18; **9**:7–8
COMPLEX **5**:13; **11**:2
compound command **5**:13, 15
concurrentization **6**:6
constant **5**:8, 14, 17, 37; **6**:1, 8; **7**:6–9, 8, 25
constant propagation **6**:8–9
control, compilation **2**:2
execution **5**:32
COPY **5**:4; **7**:19
core **5**:5–6, 16, 18–19, 31, 37–39, 41, 47
corefile **5**:2, 5–6, 18
CPU registers **10**:1
Cray **9**:17

D

data alignment **7**:31; **11**:2, 4
data dependence **6**:3
data type **5**:3, 21, 30; **11**:7
dbg **2**:2; **4**:2; **5**:1–18, 20–39, 44–45, 47–48
DCOMPLEX **5**:13
dead code **6**:9
debugger **2**:2; **4**:1; **5**:1–3, 5, 7, 17, 37, 40
declaration **5**:28; **8**:8, 10, 12; **9**:19
#DEFINE **1**:12; **3**:2; **5**:12; **8**:10–13, 12
delimited **5**:8
dependence, data **6**:3
dependences **7**:8
dependencies **6**:3, 6, 8; **8**:19, 23; **9**:11, 13
DIMENSION **7**:25
directive **6**:12; **7**:6, 27; **8**:6–7, 9–10, 12, 18–20, 23–24, 26; **9**:10–19
directives, compiler **2**:2; **9**:1, 10; **11**:1–11
DIS **5**:12, 36, 40, 42, 47
disassemble **5**:35, 47
dummy **5**:15–16, 33

dump, octal **4**:3

E

editing **1**:1–2, 6; **5**:5–7
editor, link **3**:1, 3; **4**:1–4
#ELSE **0**:3; **1**:9; **5**:16; **7**:10, 12, 14, 23, 27; **8**:20
#ENDIF **0**:3; **6**:14; **7**:9–14, 17, 19, 23, 27
entry point **5**:31
EQUIVALENCE **6**:9, 13; **7**:7–9, 29; **8**:7, 18; **11**:3
equivalenced **6**:13; **7**:1, 8–9, 30; **8**:24; **11**:3
errors **2**:1; **8**:4; **11**:2–5
execution control **5**:32
expansion, scalar **6**:12
expressions **5**:9, 29, 31, 36, 39; **6**:3; **7**:5–6, 15
extensions **8**:6; **11**:1

F

Facility, Vector Reporting **9**:18
FALSE **7**:12, 15
FAVOR_KEYWORDS **5**:12, 27
FAVOR_NAMES **5**:12, 27
fc **9**:5, 8; **11**:4
file, object **4**:3; **5**:2, 8; **6**:9; **9**:6
files, postloading executable **B**:1
format, column-major **7**:21
Fortran compiler **3**:3; **6**:9; **8**:5–11, 7, 19; **9**:17; **11**:7
FPU **5**:2, 18, 36–37, 39; **9**:9–10
frame, stack **10**:1
function **5**:4, 7–9, 12–14, 17, 20, 25, 29, 38, 40, 43; **8**:11–14, 18–20, 22–23, 25, 27; **9**:12–13, 16; **10**:1; **11**:2–3, 4, 6; **12**:1–7

H

help **4**:3; **5**:9, 11, 17–18, 26, 37, 47; **6**:1; **7**:6–48, 25, 31; **9**:10–11
hex **5**:30, 35–36
hexadecimal **4**:3; **5**:30
HISTORY **5**:13, 33

I

I/O, unformatted **7**:30–31
IEEE **11**:3–6
IF **5**:3–8, 10, 13-18, 20–23, 25–32, 34, 36, 38, 41, 44, 47; **7**:13–20
#INCLUDE **1**:3; **3**:2; **4**:2; **5**:9–3, 20, 37; **7**:25; **8**:6, 11, 13; **9**:17; **11**:1, 7
INLINE **9**:11–12
inlined **6**:9; **9**:13
int **5**:4, 13, 28, 44, 46; **7**:2–3, 5, 10, 28; **8**:8–9, 16; **10**:1; **11**:2
integer **5**:7–8, 13–14, 17, 24, 28–29, 33, 36; **6**:15; **7**:7–38, 20, 29; **8**:9; **9**:9; **10**:2; **11**:2–3, 7; **12**:3
interchange, loop **6**:10
interfacing **11**:4
interrupt **8**:2
INTRINSIC **8**:18
IPDEP **8**:19, 23; **9**:11, 13–14
IPU **9**:9
IVDEP **9**:11, 13–14, 18

K

keyword **5**:9–10, 12, 15, 19, 22–23, 25–28, 30–31, 39
keyword command **5**:9, 15, 26
keywords **5**:1, 9–11, 26–27, 31, 39, 48
KILL **4**:3; **5**:11, 32, 41; **12**:4

L

language, command **5**:13, 27
ld **3**:1; **B**:1
lib **3**:3; **5**:6–4
libc **3**:2; **5**:18–3, 43
libcurses **3**:3
libraries **2**:2; **3**:1; **7**:19–4
  archive **3**:1–2, 4
  system **2**:2
library **3**:1, 3; **5**:4; **6**:15; **8**:6, 17–18, 20, 27; **10**:3; **11**:7; **12**:1
LINE **1**:1, 3–10, 12; **2**:2; **3**:1; **4**:3; **5**:3–5, 7–8, 12, 15, 17, 20, 24, 27, 31–32, 38–39, 41, 47; **8**:18; **9**:4–8, 19; **12**:2
link editor **3**:1, 3; **4**:1–4
linking **2**:1
LIST **2**:2; **5**:3, 5, 8–9, 11–12, 20, 26, 31, 39, 44; **7**:2; **9**:17; **11**:3–48
list, comma-separated **5**:9

load 3:3; 6:1, 5; 7:19; 8:6, 8; 9:4
loader 2:2; 3:1
loading 2:2; 3:3; 5:41–4, 43; 6:1; 10:3–2
LOG 5:12, 34, 42
loop induction variable 6:7; 7:4
loop interchange 6:10
loop unrolling 6:4; 7:20

M

MAXVAL 7:9–10
messages, warning 7:5; 8:19
microtasking 8:27
MIN 6:8–14, 16; 7:3, 10–11, 14, 17, 22, 24, 26
MIPS 5:27, 31; 11:3
mkprof 4:4; 9:1–2, 4–7, 9
MNEXT 5:11
MSTEP 5:11, 41–42
multiprocessing 6:2
multiprocessor 6:1, 3
MWINDOW 5:12, 20, 47

N

NAN 11:5–7
NEXT 5:4, 11, 24–25, 37
nm 4:3
non-sticky 5:9, 22–23
NONE 1:8; 5:25; 6:7
NO_PARALLEL 9:18

O

OBJECT 1:10, 12; 2:1; 3:1; 4:3; 5:2–3, 5–6, 8, 12, 17–18, 32, 39; 6:9; 8:9; 9:6; 12:3
object file 4:3; 5:2, 8; 6:9; 9:6
octal dump 4:3
od 4:3–4
optimization 4:1; 5:5; 6:6; 7:5, 8–9, 20, 28; 8:18–29, 26; 9:3–5, 16, 18–19
option 1:7; 2:2; 3:1–3, 3; 4:1–2, 4; 5:2, 5, 18; 7:5, 29; 8:7, 24; 9:2–4, 7–10, 17
options, command line 2:2; 3:2
  compiler 2:1; 9:2–2

preprocessor **2**:3
OPT_LEVEL **9**:19
output file **B**:1
OVERFLOW **7**:7; **9**:12; **11**:5–6
O_BASYNC **A**:2

P

PARALLEL **4**:3; **5**:1, 18, 31, 39; **6**:1–3, 5–6, 11; **7**:1–12, 14, 25; **8**:1–7, 10; **9**:4–27, 10, 15; **10**:3–18
parallel processing **6**:5; **8**:3–6, 5, 18, 20, 26
parallelism **6**:3, 7; **8**:1, 4, 26; **9**:3–4
parallelization **6**:6; **7**:1; **8**:1, 3, 19–22, 26; **9**:10, 13, 15–16, 19
parallelize **6**:3, 6; **7**:13–14, 24; **8**:1–25, 3–4, 20; **9**:1–21, 4, 11, 15–16, 18
parameter passing **11**:1
passing, parameter **11**:1
PBEST **7**:27; **9**:11, 15–16
performance **7**:1, 9, 11–13, 20–21, 24–26, 29; **9**:1; **11**:7–30
point, entry **5**:31
watch **5**:22, 35
porting **11**:1–2
postloading executable files **B**:1
#pragma **9**:11, 19
preprocessor **2**:1, 3
preprocessor options **2**:3
PRINT **5**:3, 13, 19–25, 29, 35–36, 44; **7**:22; **8**:23–47
printf **5**:3–4, 18, 39–40, 42–44, 46
processes **4**:3; **5**:9, 32; **8**:1, 4, 8, 26
processing, parallel **6**:5; **8**:3–6, 5, 18, 20, 26
processor **4**:2; **5**:27; **6**:1–2, 5; **7**:25; **8**:2–3, 5–6, 8–16, 21–23, 25; **9**:3–4, 6, 13; **10**:1, 3; **11**:2; **12**:4–3
prof **4**:4; **9**:1–6
profile **9**:1–2, 5–7
profiler **4**:2; **9**:1, 4–5, 9
programs **1**:1; **2**:1; **3**:3; **4**:2; **6**:1–2, 3, 6–7, 11; **7**:1–2, 8, 22, 24, 28; **8**:1–2, 4–5, 25; **9**:1–4, 7, 9; **10**:3–11
propagation, constant **6**:8–9

R

radix **5**:12, 26, 30–31
ranlib **3**:3
re-execute **5**:33
read **5**:26, 38–39, 44; **8**:6, 21, 23; **9**:15; **12**:1–16

REAL **1**:10; **5**:13, 28; **7**:8, 24–25, 28–29, 31; **8**:3; **9**:7; **10**:2; **11**:2
recognition, reduction **6**:13
recurrence **6**:4–5, 10; **9**:13
reduction recognition **6**:13
register **4**:1; **5**:12, 27, 36, 43; **7**:12–13, 18, 26; **8**:5–27, 8; **10**:1–3
register sets **10**:1
  scalar **5**:36
  vector **5**:27, 36; **7**:13, 27; **10**:1, 3
registers, accumulator **10**:2
  CPU **10**:1
REGNAMES **5**:12
RERUN **5**:11, 18, 32
revealer **4**:2

S

save **1**:2, 6; **5**:34; **6**:9; **7**:24; **8**:7–25, 23
scalar **6**:1–3, 6, 7, 12–13, 15; **7**:8–9, 11–12, 15–17, 20, 22–24, 28, 30; **8**:20, 24; **9**:7–9, 13, 18–19; **10**:1–3
scalar expansion **6**:12
scalar register **5**:36
scatter **6**:2
scope **5**:3, 7–9, 12–14, 17, 19–21, 25, 28, 31, 38–39, 44; **6**:2; **8**:4
scope specifier **5**:8, 14, 17
semantics **6**:4; **7**:3, 15, 22
sets, register **10**:1
size **4**:3; **5**:6, 12, 14, 33, 36; **7**:19, 31; **8**:13, 25; **11**:2
SOURCE **1**:1; **2**:1; **3**:3; **4**:1–2, 3; **5**:1, 3–8, 11–12, 17–20, 24–25, 33, 36–41, 47; **6**:3, 7, 15; **7**:3; **8**:15, 17; **9**:4, 8
specifier, scope **5**:8, 14, 17
speed **3**:3; **6**:1, 3, 5, 7, 14; **7**:10–12, 22, 31; **8**:2; **9**:1, 4, 13
stack frame **10**:1
statements, compilation control **2**:2
STATUS **5**:11–12, 17–18, 20–22, 31–32, 35, 39; **8**:9; **12**:1–2, 4
STEP **3**:3; **4**:2; **5**:3–4, 11, 24–25, 32, 34, 37, 41, 44; **6**:7; **7**:4–6, 12; **8**:2, 21; **9**:2–3, 6
STOP **4**:3; **5**:11, 17, 35
stride **6**:2, 11; **7**:5–12, 21, 28; **9**:15
subexpression **7**:29
subjected **3**:2
subprogram **9**:16; **11**:2
subprograms, calling **10**:1
subroutine **5**:2–3, 7, 19, 21, 25; **6**:14; **7**:5; **8**:5–7, 9, 11, 15, 17–18, 20–23, 27; **9**:3–4, 9, 17; **12**:4–18
subscript **6**:1, 8; **7**:8, 21–22

subscripted **6**:1
synchronization **6**:6; **7**:26; **8**:2, 6, 27
SYNCHRONOUS **A**:0
system libraries **2**:2

T

techniques, compiler **6**:6
thread **6**:2; **8**:2, 5–6, 8, 11, 13, 15, 17–18, 24–26
threadlocal **8**:7, 9–13, 17, 20; **9**:17–24
TRACEBACK **5**:3, 7, 13, 25
type, data **5**:3, 21, 30; **11**:7

U

UCHAR **5**:13, 28
UINT **5**:13, 28
unambiguous **5**:27
UNCATCH **5**:12–13, 32–33
unformatted I/O **7**:30–31
unit, vector **6**:1–2, 4–5, 13; **7**:12–13, 15; **9**:13; **11**:7–14
UNLOG **5**:12, 34
unroll **7**:21, 30
Unrolling **6**:4; **7**:20
unrolling, loop **6**:4; **7**:20
USHORT **5**:13, 28

V

variable, loop induction **6**:7; **7**:4
VBEST **6**:12; **7**:27; **9**:11, 14–16
VECTOR **5**:27, 36; **6**:1; **7**:1–3, 5, 7, 9–20, 22, 24–28, 30; **8**:3–4, 23; **9**:6–14, 16, 18; **10**:1; **11**:7–3
vector register **5**:27, 36; **7**:13, 27; **10**:1, 3
Vector Reporting Facility **9**:18
vector unit **6**:1–2, 4–5, 13; **7**:12–13, 15; **9**:13; **11**:7–14
vectorization **6**:8, 10; **7**:1–12, 3, 6–8, 22, 26; **9**:10–27, 12–13, 15, 18–19
vectorize **6**:3, 6–8, 10; **7**:2–12, 7–9, 13–16, 22; **9**:1–27, 11, 14–16, 18
vectors **6**:6, 14; **7**:11, 18, 21, 30; **9**:11–31
VPROC **7**:6; **9**:11, 16
VREPORT **6**:7–8, 10–11, 14; **7**:2; **8**:23; **9**:1–15, 10, 18

W

warning messages **7**:5; **8**:19
watch point **5**:22, 35
WHERE **5**:3–4, 6–7, 9, 11, 13, 17–19, 21, 25–27, 33, 39, 43;
WINDOW **1**:7; **5**:12, 16, 20, 35–36, 47