June 1989
Order Number: 311768-001

iPSC®/2
ADA PROGRAM DEVELOPMENT GUIDE

Intel® Corporation

Copyright ©1990 by Intel Scientific Computers, Beaverton, Oregon. All rights reserved. No part of this work may be reproduced or copied in any form or by any means...graphic, electronic, or mechanical including photocopying, taping, or information storage and retrieval systems...without the express written consent of Intel Corporation. The information in this document is subject to change without notice.

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

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

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

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

286
386
4-SITE
Above
BITBUS
COMMputer
Concurrent File System
Concurrent Workbench
CREDIT
Data Pipeline
Direct-Connect Module
FASTPATH
GENIUS
i
I^2ICE
i860
ICE
iCEL
iCS
iDBP
iDIS
iLBX
im
Im
iMDDX
iMMX
Insite
int$_e$l
int$_e$lBOS
Intelevision
Intellec
int$_e$ligent Identifier
int$_e$ligent Programming
Intellink
iOSP
iPDS
iPSC
iRMX
iSBC
iSBX
iSDM
iSXM
KEPROM
Library Manager
MAP-NET
MCS
Megachassis
MICROMAINFRAME
MULTIBUS
MULTICHANNEL
MULTIMODULE
ONCE
OpenNET
OTP
PC BUBBLE
Plug-A-Bubble
PROMPT
Promware
QueX
QUEST Programming
Quick-Pulse
Ripplemode
RMX/80
RUPI
Seamless
SLD
SugarCube
UPI
VLSiCEL

Ada is a registered trademark of the U.S. Government, ADA Joint Program Office
APSO is a service mark of Verdix Corporation
Ethernet is a registrered trademark of XEROX Corporation
Excelan is a trademark of Excelan Corporation
EXOS is a trademark or equipment designator of Excelan Corporation
Green Hills Software, C-386, and FORTRAN-386 are trademarks of Green Hills Software, Inc.
GVAS is a trademark of Verdix Corporation
Lucid and Lucid Common Lisp are trademarks of Lucid, Inc.
NFS is a trademark of Sun Microsystems
Sun Microsystems and the combination of Sun and a numeric suffix are trademarks of Sun Microsystems
UNIX is a trademark of AT&T
VADS and Veridx are registered trademarks of Verdix Corporation
VAST2 is a registered trademark of Pacific-Sierra Research Corporation
VMS and VAX are trademarks of Digital Equipment Corporation
VP/ix is a trademark of INTERACTIVE Systems Corporation and Phoenix Technologies, Ltd.
XENIX is a trademark of Microsoft Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	06/89
----	Revised by Change Notice 311929-001	05/90

RESTRICTED RIGHTS

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

Contents

1 INTRODUCTION

1.1 TO THE VADS USER 1-1
1.2 VADS DOCUMENTATION 1-1
1.3 HOW TO USE THIS MANUAL 1-2
1.4 ON-LINE HELP 1-2
1.5 NOTATIONAL CONVENTIONS 1-2
1.6 GENERAL IMPLEMENTATION INFORMATION ... 1-4
1.6.1 VADS User Interface 1-4
1.7 REPORTING PROBLEMS 1-6

2 RELEASE NOTES

CHANGES IN VERSION 5.7 2-1
2.1 GENERAL IMPROVEMENTS 2-2
2.2 DEBUGGER 2-2
2.2.1 Expanded debugger commands: 2-2
2.2.2 Familiar output has been slightly changed: 2-2
2.3 PRAGMAS 2-3
2.4 TOOLS 2-3
ada 2-3
a.das 2-3
a.du 2-3
a.info 2-3
a.ld 2-4
a.ls 2-4
a.make 2-4
a.mklib 2-5
a.path 2-5
a.pr 2-5
2.5 COMPILER CHANGES 2-5
2.6 OPTIMIZATION 2-6
2.7 MACHINE_CODE 2-6
2.8 FLOATING POINT NUMBERS 2-6
2.9 FIXED POINT NUMBERS 2-6

USERS GUIDE

3 OVERVIEW OF VERDIX ADA EXL DEVELOPMENT SYSTEM

3.1 OVERVIEW OF VADS COMPONENTS 3-1
3.1.1 Library Management Tools 3-1
3.1.2 Compiler 3-2
3.1.3 Program Generation Tools 3-2
3.1.4 Analysis Tools 3-2
3.1.5 Debugger 3-2
3.1.6 Source Code Formatter 3-2
3.1.7 Additional Tools 3-2
3.1.8 Supplied Libraries 3-2
3.2 LIBRARY MANAGEMENT TOOLS 3-2
3.3 COMPILER 3-3
3.4 PROGRAM GENERATION TOOLS 3-3
3.5 ANALYSIS TOOLS 3-3
3.6 DEBUGGER 3-4
3.7 SOURCE CODE FORMATTER 3-4
3.8 ADDITIONAL TOOLS 3-4

4 LIBRARY MANAGEMENT TOOLS

4.1 OVERVIEW 4-1
4.2 SEPARATE COMPILATION INFORMATION 4-1
4.3 ADA COMPILATION UNITS 4-1
4.4 ADA SOURCE FILE STRUCTURE 4-2
4.5 UNIT DEPENDENCIES 4-2
4.6 VADS LIBRARY DIRECTORIES 4-3
4.7 LIBRARY MANAGEMENT TOOLS 4-4
4.7.1 **a.cleanlib** 4-5
4.7.2 **a.du** 4-5
4.7.3 **a.info** 4-6
4.7.4 **a.ls** 4-8
4.7.5 **a.mklib** 4-8
4.7.6 **a.vadsrc** 4-10
4.7.7 **a.path** 4-11
4.7.8 **a.rm** 4-12
4.7.9 **a.rmlib** 4-13
4.7.10 **a.tags** 4-13
4.7.11 **a.which** 4-15

5 THE COMPILER

5.1 OVERVIEW 5-1
5.2 INVOKING THE COMPILER 5-1
5.3 COMPILER PROCESSING 5-2
5.4 COMPILER ERROR MESSAGES 5-2
5.5 COMPILER ERROR MESSAGE PROCESSING 5-3
5.6 LOCK FILES 5-6
5.7 ASSEMBLER OUTPUT 5-7

6 PROGRAM GENERATION TOOLS

6.1 OVERVIEW 6-1
6.2 **ada** 6-1
6.3 **a.ld** 6-1
6.3.1 WITH*n* Directives 6-2
6.3.2 Foreign Object Files 6-2
6.3.3 Linker Options File and Runtime Library 6-3
6.4 **a.make** 6-3
6.5 EXECUTING COMPILED PROGRAMS 6-5

7 ANALYSIS TOOLS

7.1 OVERVIEW 7-1
7.2 **a.das** 7-1

8 VADS DEBUGGER

8.1 OVERVIEW 8-1
8.2 USING THE DEBUGGER AS A LEARNING TOOL . 8-1
8.3 INVOKING THE DEBUGGER 8-2
8.3.1 Command File Input: **-i** 8-3
8.3.2 Ada Library Directory: **-p** 8-3
8.3.3 Executable File 8-3
8.3.4 Display Debugger Executable: **-sh** 8-3
8.3.5 Screen-mode Invocation: **-v** 8-3
8.3.6 Redirect Debugger Input and Output 8-4
8.4 Halting the Program 8-4
8.5 Terminating the Debugger Session 8-4
8.6 On-line Debugger Help 8-4
8.7 COMMAND SYNTAX 8-5
8.8 DEBUGGER CONCEPTS 8-6
8.8.1 Call Stack, Home and Current Positions 8-7
8.8.2 Overload Resolution (Disambiguation) 8-8

8.8.3 Names, Variables, and Visibility Rules 8–8
8.9 LOADING THE PROGRAM 8–9
8.10 MODIFY DEBUGGER CONFIGURATION: **set** 8–9
8.11 Display Commands 8–10
8.11.1 Display Lines: **l, w** 8–10
8.11.2 Display Breakpoints: **lb** 8–11
8.11.3 Display Instructions: **li, wi** 8–11
8.11.4 Display Tasks: **lt** 8–13
8.11.5 Display Variables: **p** 8–16
8.11.6 Display Register Contents: **reg** 8–16
8.11.7 Display Call Stack of Task: **cs** 8–17
8.11.8 Display Task Number: **task** 8–17
8.11.9 Value Assignment 8–17
8.11.10 Display Raw Memory 8–18
8.12 EXECUTION COMMANDS 8–19
8.12.1 Step One Line/Instruction: **a, ai, s, si** 8–19
8.12.2 Continue Execution: **g, gw, gx** 8–20
8.12.3 Run Program: **r** 8–20
8.12.4 Read Debugger Commands From a File: **read** ... 8–20
8.12.5 Terminate the Debugger Session: **exit, quit** 8–21
8.13 BREAKPOINT COMMANDS 8–21
8.13.1 Set Breakpoint: **b** 8–21
8.13.2 Command Blocks 8–22
8.13.3 Set Conditional Breakpoint: **b** 8–22
8.13.4 Break Down: **bd, bD** 8–23
8.13.5 Set Breakpoint at Instruction: **bi** 8–24
8.13.6 Set Breakpoint at Exception: **bx** 8–24
8.13.7 Display Instruction: **li, wi** 8–24
8.13.8 Delete Breakpoint: **d** 8–24
8.13.9 Implicit Breakpoints 8–24
8.13.10 Tasking Breakpoints 8–25
8.14 POSITIONING COMMANDS 8–25
8.14.1 Specify New Position 8–25
8.14.2 Call Stack Bottom, Top: **cb, ct** 8–25
8.14.3 Call Stack Up, Down: **cd, cu** 8–26
8.14.4 Call Stack: **cs** 8–26
8.14.5 Move Current Position: **e, select** 8–26
8.14.6 Enter Editor: **edit** 8–27
8.15 INVOKING THE SCREEN-MODE DEBUGGER ... 8–27
8.15.1 Screen-mode Windows 8–27
8.15.2 Instruction and Source Modes 8–28
8.15.3 Screen-mode Window Commands 8–29
8.15.4 Window Command Syntax 8–29

8.16 CROSS DEBUGGING [cross compilers only] 8-29
8.17 DB_IFACE [cross compilers only] 8-30
8.17.1 Process Startup 8-31
8.18 DEBUGGING C PROGRAMS 8-31
8.18.1 Names, Variables, and Visibility Rules 8-31
8.18.2 Type Checking 8-32
8.18.3 **a.db** Stack Commands 8-32
8.18.4 **a.db** Screen Functions 8-32
8.18.5 C Operations 8-32
8.18.6 Global C Variables 8-32
8.18.7 char * Variable Display 8-32
8.18.8 Warning 8-33
8.18.9 Operators 8-33

9 SOURCE CODE FORMATTER

9.1 OVERVIEW 9-1
9.2 INVOKING **a.pr** 9-1
9.3 DEFAULT FORMAT SPECIFICATIONS 9-1
9.4 Fixed Format Specifications 9-2
9.5 Error and Warning Messages 9-2
9.6 **.prrc** FILE OPTIONS 9-3
9.7 COMMAND-LINE OPTIONS 9-3
9.8 **a.pr** OUTPUT 9-3
9.8.1 Comments 9-3
9.8.2 Pagination 9-4
9.8.3 Splitting Lines 9-5
9.8.4 Line Length Specification 9-6
9.8.5 Tabs 9-6
9.9 EXAMPLES 9-6
9.9.1 Unformatted Source File 9-6
9.9.2 The **.prrc** file 9-7
9.9.3 Output 9-7
9.9.4 Command Line Options 9-7

10 ADDITIONAL TOOLS

10.1 OVERVIEW 10-1
10.2 **a.help** 10-1
10.3 **a.list** 10-1

USING VADS

11 GETTING STARTED WITH VADS

11.1 INTRODUCTION 11-1
11.2 GETTING STARTED 11-1

12 MACHINE CODE INSERTIONS

12.1 OVERVIEW 12-1
12.2 MACHINE CODE PROCEDURES 12-1
12.3 CODE-STATEMENTS 12-2
12.3.1 Opcodes 12-2
12.3.2 Operands 12-2
12.3.3 Ada Entities as Operands 12-5
12.4 PROGRAM CONTROL 12-6
12.5 SUBPROGRAM CALL 12-7
12.6 LOCAL DATA 12-8
12.6.1 Jump Table via Offsets 12-8
12.7 PREFIX OVERRIDES 12-9
12.8 FLOATING POINT 12-9
12.9 PRAGMAS 12-10
12.9.1 **pragma** INLINE 12-10
12.9.2 **pragma** SUPPRESS 12-10
12.9.3 **pragma** IMPLICIT_CODE 12-11
12.9.4 **pragma** OPTIMIZE_CODE 12-11
12.9.5 Debugging Machine Code 12-11
12.10 **package** MACHINE_CODE 12-11

13 INTERFACE PROGRAMMING USING VADS

13.1 OVERVIEW 13-1
13.2 ADA INTERFACE TO curses 13-1
13.2.1 Step 1 — Create Parallel Data Types 13-2
13.2.2 Step 2 — Declare External Subprograms 13-6
13.2.3 Step 3 — Access Global Variables 13-10
13.2.4 Step 4 — Link With Existing Libraries 13-11
13.2.5 Step 5 — Overhead Reduction 13-14
13.3 PROGRAM CONVERSION 13-15
13.4 CALLING ADA FROM OTHER LANGUAGES 13-17
13.4.1 Finding the Right Object 13-17

13.4.2 Avoiding Elaboration 13-18
13.4.3 Linking a Non-Ada Main Program 13-18
13.4.4 Runtime Considerations 13-19
13.4.5 Summary: Calling Ada From Other Languages. . 13-19

14 SAMPLE PROJECT

14.1 OVERVIEW 14-1
14.2 THE PROJECT PLAN 14-1
14.3 USING **a.mklib** 14-1
14.4 USING **ada** AND **a.error** 14-2
14.5 USING **a.path** 14-4
14.6 USING **a.make** 14-5
14.7 USING **a.ld** 14-6
14.8 USING VADS LIBRARIES 14-6
14.9 IMPROVING THE PROGRAM 14-7
14.10 USING **a.db** 14-9
14.11 USING **a.which** 14-11
14.12 USING **a.rm** 14-12
14.13 USING **a.du** 14-12
14.14 USING **a.tags** 14-12

15 SAMPLE SCREEN-MODE DEBUGGING SESSION

15.1 OVERVIEW 15-1
15.2 THE PROCEDURE TO BE DEBUGGED 15-1
15.3 SCREEN-ORIENTED DEBUGGING SESSION 15-4

16 APPLICATION NOTES

16.1 BENCHMARKING 16-1
16.2 BENCHMARKING COMPILATION SPEED 16-1
16.3 BENCHMARKING RUNTIME PERFORMANCE ... 16-2

17 INSTALLATION GUIDE

17.1 OVERVIEW 17-1
17.2 INSTALLATION 17-1
17.2.1 Installation Requirements 17-1
17.2.2 Typical VADS Installation 17-2
17.3 CUSTOMER REPORTS 17-5
17.3.1 Submitting a Customer Report to APSO 17-5

18 IMPLEMENTATION REFERENCE

18.1 INTRODUCTION 18-1
18.2 RELEASE DIRECTORY STRUCTURE 18-2
18.2.1 The **standard** VADS Library 18-3
18.2.2 The **verdixlib** VADS library 18-6
18.2.3 The **publiclib** VADS library 18-7
18.2.4 The examples Directory 18-8
18.3 ESTIMATING DISK SPACE REQUIREMENTS FOR USING VADS 18-9
18.4 PRAGMAS AND THEIR EFFECTS 18-10
18.5 IMPLEMENTATION-DEFINED ATTRIBUTE: 'REF 18-13
18.5.1 X'REF 18-13
18.5.2 SYSTEM.ADDRESS'REF(N) 18-14
18.6 RESTRICTIONS ON 'MAIN' PROGRAMS 18-14
18.7 GENERIC DECLARATIONS 18-14
18.8 SHARED OBJECT-CODE FOR GENERIC SUBPROGRAMS 18-15
18.9 REPRESENTATION SPECIFICATIONS 18-16
18.10 SOURCE FILE STRUCTURE/RESTRICTIONS 18-22
18.11 PARAMETER PASSING 18-23
18.12 CONVERSION AND DEALLOCATION 18-24
18.13 PROCESS STACK SIZE 18-24
18.14 INTERFACE PROGRAMMING 18-25
18.15 PREDEFINED PACKAGES AND GENERICS 18-26
18.15.1 Specification Of package SYSTEM 18-27
18.15.2 package CALENDAR 18-28
18.15.3 package SEQUENTIAL_IO 18-28
18.16 TYPES, RANGES, AND ATTRIBUTES 18-28
18.17 INPUT/OUTPUT 18-30
18.18 MACHINE CODE INSERTIONS 18-31
18.19 VADS FILE FORMATS 18-32
18.19.1 Object, Executable and Archive Files 18-33
18.19.2 VADS Library Reference Files: **ada.lib** and **gnrx.lib** 18-33
18.19.3 Supported INFO and LINK Directive Names 18-35
18.19.3.1 INFO Directive Names 18-35
18.19.3.2 LINK Directive Names 18-36
18.20 RUNTIME SYSTEM 18-37
18.20.1 Overview 18-37
18.20.2 Memory Allocation 18-38

18.20.3 Semaphores 18-40
18.20.4 Tasking 18-40
18.20.4.1 Task Creation 18-41
18.20.4.2 Task Activation 18-43
18.20.4.3 Task Startup 18-44
18.20.4.4 Delay Statements 18-45
18.20.4.5 Task Entry Calls 18-46
18.20.4.6 Accept and Select Statements 18-47
18.20.4.7 Task Completion and Termination 18-49
18.21 RUNTIME SYSTEM PANIC NUMBERS AND MESSAGES 18-50
18.22 MEMORY ALLOCATION SUPPORT IN THE VADS RUNTIME 18-54
18.22.1 Allocators 18-54
18.22.2 VADS Library Memory Management Semantics . 18-56
18.22.3 VADS-Supplied Memory Management 18-58
18.22.3.1 Simple Allocation: HEAP_ALLOC 18-58
18.22.3.2 Pool-Based Allocation: POOL_ALLOC 18-59
18.22.3.3 Pool Control 18-62
18.22.3.4 Suggestions for Use of VADS Memory Allocation 18-64
18.22.3.5 Mutual Exclusion During Allocation 18-65
18.22.3.6 Memory Management and Underlying Operating Systems 18-65
18.22.3.7 Replacing User-space Memory Allocation 18-66
18.22.3.8 Memory allocation in the Kernel 18-67
18.23 VADS BASIC BIT MANIPULATION UTILITIES .. 18-68
18.23.1 The IFACE_BITS Package 18-68
18.23.2 Semantics of the IFACE_BITS Utilities 18-68
18.24 VADS SEMAPHORE SERVICES 18-68
18.24.1 The Semaphore Package 18-69
18.24.2 Semaphore Structures 18-70
18.24.3 Creating Semaphores 18-70
18.24.4 Defining a Critical Region 18-70
18.24.5 Semaphore Semantics 18-71
18.24.6 Kernel SUSPEND and RESUME 18-71
18.25 VADS TASK CONTROL SERVICES 18-72
18.25.1 The IFACE_TASKS Package 18-73
18.25.2 Controlling Tasks While Debugging 18-75
18.25.3 User Control of Tasking 18-76
18.26 VADS EXCEPTION REPORTING 18-76
18.26.1 Package IFACE_RAISE 18-77
18.26.2 Debugger Exception Output 18-81

19 VADS COMMAND REFERENCE

summary 19-1
ada 19-2
a.cleanlib 19-5
a.das 19-6
a.dbr 19-8
a.du 19-9
a.error 19-10
a.help 19-12
a.info 19-14
a.ld 19-16
a.list 19-18
a.ls 19-19
a.make 19-20
a.mklib 19-23
a.patht 19-25
a.pr 19-26
a.report 19-29
a.rm 19-32
a.rmlib 19-33
a.tags 19-34
a.vadsrc 19-36
a.which 19-37

20 DEBUGGER REFERENCE

summary 20-1
a 20-3
a.db 20-4
address 20-6
ai 20-7
:= 20-8
b 20-9
bd 20-12
bi 20-13
breakpoints 20-14
bx 20-15
cb 20-16
cd 20-17

command syntax 20-19
cs 20-21
ct 20-22
cu 20-23
current frame 20-24
current position 20-25
d 20-26
DB_IFACE 20-27
display memory 20-29
e 20-31
edit 20-33
examine 20-34
exit 20-35
expressions 20-36
files 20-37
g 20-38
gw 20-39
gx 20-40
help 20-41
home position 20-42
l 20-43
lb 20-44
li 20-45
line numbers 20-46
load 20-47
lt 20-48
overloading 20-53
p 20-54
pass 20-56
quit 20-57
r 20-58
read 20-59
reg 20-60
<RETURN> 20-61
s 20-62
screen mode 20-63
search 20-68
select 20-69
set 20-70
sh 20-73
s 20-74
stop 20-75
sync 20-76
task 20-77
terminal control 20-78
vi 20-79
visibility rules 20-80
w 20-81
wi 20-82

I INDEX

CHAPTER 1

INTRODUCTION

1.1 TO THE VADS USER

The VERDIX Ada Development System (VADS) is a toolset for the development of Ada programs. VADS processes the full Ada language as specified by the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, referred to in VADS documentation as the Ada Reference Manual or RM.

1.2 VADS DOCUMENTATION

This manual contains information on how to use VADS. It is organized in three presentations, a tutorial, references and walk-throughs. These allow each individual user the most efficient means of using the materials.

- A tutorial presentation is found in the *USER'S GUIDE*.

  The *USER'S GUIDE* (Chapters 3 through 10) presents each major concept utilized by the system and all of the tools that apply to it. This section, along with CHAPTER 19, VADS COMAND REFERENCE provides a discussion of the tools and how to use them.

- The reference sections included are the *CHAPTER 19, VADS COMMAND REFERENCE; CHAPTER 20, DEBUGGER REFERENCE; CHAPTER 18, IMPLEMENTATION REFERENCE;* and *CHAPTER 2, RELEASE NOTES*.

  The *VADS COMMAND REFERENCE* and *DEBUGGER REFERENCE* provide concise reference entries for each VADS tool and for each debugger command. Compiler limits and implementation-specific Ada language features are dealt with in the *IMPLEMENTATION REFERENCE*. Predefined types are listed with their attribute values. The *RELEASE NOTES* provide an up-to-date summary of product changes and a list of known problems.

- *USING VADS* includes walk-throughs.

  Several walk-throughs, sample applications, and scenarios for the use of VADS tools, the debugger, and compiler are presented in *USING VADS*.

- The *INSTALLATION GUIDE* covers product installation.

1.3 HOW TO USE THIS MANUAL

After the VADS system is installed using the *INSTALLATION GUIDE* section of the manual, a user can immediately begin to write and compile Ada code. *GETTING STARTED WITH VADS* in the *USING VADS* section provides a new user with the basic steps required. A more thorough understanding of the tools can then be obtained from the *USER'S GUIDE*. A study of the walk-throughs and examples in *USING VADS* is helpful in developing applications.

The reference sections, *VADS COMMAND REFERENCE* and the *DEBUGGER REFERENCE*, offer easily accessible information in a concise format. Tools and commands are listed in alphabetical order.

Operating system implementation differences affecting VADS operations or Ada programming using VADS are intended to be confined to and detailed in the *RELEASE NOTES*.

1.4 ON-LINE HELP

On-line reference manual entries are available for the compiler and all VADS tools as well as for debugger commands and concepts. Refer to the **a.help** entry in the *VADS COMMAND REFERENCE* or **help** in the *DEBUGGER REFERENCE* for an explanation of the several ways in which on-line help can be accessed.

1.5 NOTATIONAL CONVENTIONS

The name UNIX is used throughout VADS documentation to represent all operating systems derived from AT&T Bell Laboratories' UNIX.

Cross references to other sections of the manual designate unit title, chapter title, section number and title, and page number. In the example,

> For a discussion of external dependencies (WITH*n* directives), **see also** *PROGRAM GENERATION TOOLS*, 6.3 **a.ld**, page 6-1.

USER'S GUIDE is the name of the unit. *PROGRAM GENERATION TOOLS* is the name of the chapter. 6.3 **a.ld** is the number and title of the section containing the material under discussion. In this case, the subsection title is also the name of a VADS tool. The page number designates the chapter number (6) and the page.

In the example:

> **See also** *[IMPLEMENTATION REFERENCE]*, 18.19 *VADS FILE FORMATS*, page 18-31.

[IMPLEMENTATION REFERENCE], a chapter is enclosed in brackets to indicate that the reader is currently in that chapter and that the cross reference is to another section of the same chapter.

Typefaces and sizes are used to differentiate among text, code, meta names (user supplied names), commands, actual file names, etc. as shown in Table 1-1.

Table 1-1. Documentation notation.

Construct	Typeface	Example
Meta names	italics	*ada_source.a* [*options*]
Actual file names and directory names	bold face in text and typewriter font in examples	**GVAS_table** `/usr/babbage/adaproject`
Environment variables	capitals	$EDITOR
Commands	bold face in text and typewriter font in examples	**a.mklib** `a.mklib`
Operating system commands and utilities	bold face in text	**vi**
Ada keywords	bold face in text	**pragma**
Ada identifiers	capitals	INTERFACE

1.6 GENERAL IMPLEMENTATION INFORMATION

1.6.1 VADS User Interface

The VADS user interface is designed to provide a consistent set of tools that will be easily usable by programmers experienced with the host operating system and easily learned by others. The VADS toolset and operating system utilities can be considered an extended APSE and together provide an extremely rich programming environment for Ada software development.

Two notable extensions over standard tools are in the debugger and the compiler front end. The debugger supports high-level language constructs and does not require any special compiler qualifiers to produce debuggable objects. The compiler itself offers extensive error recovery and diagnostic messages.

A compilation excerpt showing error messages from the compiler is shown in Figure 1-1. Both the compiler and debugger have knowledge of the appropriate Ada RM sections explaining syntax, semantic, and runtime errors and exceptions.

An example of the debugger's presentation and explanation of a runtime error is shown in Figure 1-2 in screen mode format.

```
PROCEDURE B74301B IS

   GENERIC
      TYPE T IS PRIVATE;
      INP :IN T;
   PACKAGE P1 IS END P1;

   PACKAGE PKG IS
      TYPE PRIV IS PRIVATE;
      DC: CONSTANT PRIV;

      PACKAGE I1 IS NEW P1 (PRIV,DC);
---------------------------^A                                    ###
--------------------------------^B                               ###
--### A:error:RM 7.4.1(4):type is not yet fully defined
--### B:error:RM 7.4.3(2):illegal use of a deferred constant

   PRIVATE
      TYPE PRIV IS (X);

      PACKAGE I2 IS NEW P1 (PRIV,DC);
--------------------------------^A                               ###
--### A:error:RM 7.4.3(2):illegal use of a deferred constant
```

Figure 1-1. Compilation Excerpt

```
 68       configuration (col):=row;
 69    end;
 70
 71    Procedure remove_queen (row, col:in integer) is
 72       vacant:constant integer:=0;
 73    begin
 74*      safe_row (row):=true;
 75       safe_up_diag (row + col):=true;
 76       safe_down_diag (row - col):=true;
 77       configuration (col):=vacant;
 78    end;
 79
 80    function is_safe (row, col:in integer) return boolean is
 81    begin
-*-------------------------------------------------------queens.a--
**MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error" RAISED
(program is now stopped on line where exception was raised)
stopped 8 instructions after "/vc/mike/hdemo/queens.a":74 in
remove_queen
74        safe_row (row):=true
  --------^
=> RM 4.1.1(4):index value greater than upper bound of index subtype
```

Figure 1-2. Debugger Output

1.7 REPORTING PROBLEMS

Suggestions for improving software, documentation, or service are encouraged and should be submitted to the Verdix Ada Product Support Office (APSO) at the address below. Information to be included with each suggestion is outlined under **a.report** in the *VADS COMMAND REFERENCE.*

Ada Product Support Office
Verdix Corporation
Sullyfield Business Park
14130-A Sullyfield Circle
Chantilly, Virginia 22021
(703) 378-7600

Customers contracting with Verdix for additional product support can use the automated reporting tool **a.report** to submit customer reports to APSO as described under **a.report** in the *VADS COMMAND REFERENCE.*

CHAPTER 2

RELEASE NOTES

This compiler is validated under Suite_1.10 of the ACVC tests. This release is accompanied by new documentation. If you find errors in either the documentation or the VADS software components, please report them to Verdix's Customer Support.

Changes and improvements since Version 5.5 products are described below.

CHANGES IN VERSION 5.7

IMPORTANT

The Ada Validation Office has disallowed the optional third parameter to **pragma** INTERFACE. This ruling forces us to change our long-standing implementation of that pragma as follows:

- PRAGMA INTERFACE_NAME replaces PRAGMA INTERFACE_OBJECT. PRAGMA INTERFACE_NAME is more general by allowing either objects or subprograms as the first argument.

- The optional third parameter has been deleted from the PRAGMA INTERFACE. Now, PRAGMA INTERFACE_NAME(*Ada_subprogram_name*, "link_name"), must be used in conjunction with an PRAGMA INTERFACE for subprograms.

For example:

pragma INTERFACE(*language_name*, *Ada_subprogram_name*, *"link_name"*);

Is now:

pragma INTERFACE(*language_name*, *Ada_subprogram_name*);
pragma INTERFACE_NAME(*Ada_subprogram_name*, *"link_name"*);

- **a.pragma** is a tool that will assist in converting code that contains the old usage of **pragma** INTERFACE to the new legal pragmas. Its location is: *VADS_location*/**sup/diag/a.pragma**.

  Syntax: `a.pragma` *ada_source.a*

  The File *ada_source.a* will be overwritten.

2.1 GENERAL IMPROVEMENTS

- Available VADS releases are listed and expected in a file called /etc/VADS, previously it was called /usr/lib/VADS.
- Report package has been added to **verdixlib**.
- The files in the **.lines** directory have information on register use.
- The physical address functions previously found in **system.a** are no longer supported (PHYSICAL_ADDR, ADDR_GT, ADDR_LT, ADDR_GE, ADDR_LE, INCR_ADDR, DECR_ADDR).
- A file listing the legal INFO directives is called LEGAL.*target*, where *target* is the value of TARGET:INFO: directive found in **standard/ada.lib**, (e.g., SELF_TARGET, mc68881, mc68020, IAPX80386).

2.2 DEBUGGER

2.2.1 Expanded debugger commands:

- The debugger recognizes an "e" in a floating point number (e.g. 1.2e35)
- The debugger may be run in background with input from standard input, as long as your program gets its input from a separate input file. Along with this change, we added "set input |tty|pty" and "set output |tty|pty" to redirect input and output for your program. The old "set catchio on" has been replaced by "set input|output pty", and "set catchio off" is "set input|output tty".
- The old "set output 8|10|16" command is "set obase 8|10|16".
- Program parameters with embedded environment variables are properly expanded.
- You may modify floating point registers.
- You may display task variables (and thus arrays of tasks).
- The debugger displays the values of an Ada variable that the optimizer has placed in a register, but only while it is active in that register (i.e., not spilled to memory to make way for another variable).
- The debugger supports correct handling of objects declared by a renames declaration.

2.2.2 Familiar output has been slightly changed:

- The debugger can recognize many cases when your program is stopped in an interrupt handler, and can follow the chain of calls to the subroutine which was executing when the interrupt occurred.

- The response to a **cs** command will include a "!" on each line which corresponds to an interrupt handler.
- When displayed by the **lt** command, tasks that are accepted for immediate rendezvous show status "in rendezvous", where before they showed "ready".

2.3 PRAGMAS

- Calls to C interface procedures will no longer generate a warning for parameters whose type derive from ADDRESS.
- **pragma** OPTIMIZE_CODE(OFF) can now be used in any subprogram to prevent optimization. Previously, it was allowed only in machine code subprograms.
- If **pragma** NO_IMAGE is applied to an enumeration type, a warning will be issued when the IMAGE, WIDTH, VALUE attribute are used.

2.4 TOOLS

ada

- **ada -d** will correctly compile subprograms instantiations.
- **ada -d** correctly determines the dependencies of generic subprograms with explicit specs in the same file as the body.

a.das

- **a.das** correctly disassembles object files in non-Ada libraries.
- **a.das** disassembles source files having their units out of order in the **ada.lib**. This feature does not handle the disassembly of programs with separates or generics. If there are units outside the specified source file with names starting with one of the sorce file unit names, those outside units will incorrectly be brought in.

a.du

- **a.du** now works correcly in a library with INFO and LINK directives.

a.info

- When adding or removing a LINK or INFO directive, it is no longer necessary for the directive name to be in lower case. Relative file names are expanded when added through **a.info** in the **ada.lib**.

a.ld

- **a.ld** can link programs which contain over three hundred units.
- **a.ld** links with WITH*n* directives for archive libraries requiring multiple links.
- **a.ld -DX** can be used when linking a large number of units causes the error message "local symbol-overflow" to be encountered.

a.ls

- **a.ls** correctly handles libraries with units having more than 80 characters in their names.

a.make

There are several interface and operation general improvements:

- With only subprogram instantiations, **a.make** can correctly compile files and give dependency information
- **a.make** can correctly compile files containing only subprogram instantiations.
- The verbose option (**-v**) lists the units in a file in the order that they appear in the file.
- **a.make** no longer gives an assertion error when neither a *unit_name* nor the **-f** option have been specified and an INFO or LINK directive is in the current **ada.lib**. For example, an assertion error would no longer occur with **a.make -v** or **a.make -I** *file_name*.
- **a.make -I** *source_file* now correctly prints out the compilation and linking commands that would be invoked if *source_file* was changed, even if the qualifier **-f** *file_name* is not specified.
- **a.make -A** *VADS_library* now correctly compiles out-of-date files in *VADS_library*, even if *VADS_library* was specified as a relative path name.
- There is an additional new option:

  a.make -F *file_name*

  where *file_name* is a file which itself is a list of files to be considered for compilation. This should be used instead of the VADS command:

  ```
  a.make  -F *.a
  ```

  when "***.a**" expansion exceeds the UNIX command line length limitation.

a.mklib

- **a.mklib** has better error condition handling. If no parent library is specified through use of a **.vadsrc** file or at invocation time, and a single entry in **/etc/VADS** exists, it is used as the default parent library.

a.path

- **a.path** is no longer sensitive to the order of the directives in the **ada.lib**.

a.pr

- **a.pr** works correctly on statements split over several lines.
- **a.pr** directs error messages to **stderr**.
- **a.pr** correctly indents files with subprograms using named parameter association that followed exception handlers, having multiple exception selectors.
- **a.pr** handles the situation when the number of leading blanks exceeded the number of characters to be placed.
- **a.pr** can correctly indent consecutive begin statements.
- **a.pr** correctly handles **pragma** PAGE.

2.5 COMPILER CHANGES

- 32 bit address types are supported.
- 32 bit integer types are supported.
- Aggregates compile faster.
- The allowable path length in **ada.lib** is extended to 1024 chars.
- Building of the DAG structure in the optimizer is simplified and improved.
- Look-ups of index registers and address operands are improved.
- Compilation no longer exits with a bus error when the GVAS_table is unwriteable.
- Checking for forcing occurrences as defined by the Ada RM Chapt. 13 is implemented.
- Length clauses applied to scalar types are now recognized and used for the allocation of components of the scalar type nested in a composite type.
- Alignment clauses (Ada RM 13.4, **at mod**) except for **at mod 1** and **at mod 2** are not supported.

2.6 OPTIMIZATION

- Many bugs have been fixed and overall performance is faster.
- Register variables are supported by the compiler, but are not supported by the debugger.

2.7 MACHINE_CODE

- The following sized jumping/transfer instruction forms with a size suffix are no longer supported: BRA,Bcc,BSR,DBcc,FBcc. For example, a previous use of "bra_1' must be replaced iwht "bra". The compiler will choose the optimal sized branch.

2.8 FLOATING POINT NUMBERS

- Floating point to integer conversion has been fixed
- Floating point rounding has been corrected.

2.9 FIXED POINT NUMBERS

- Fixed Point values are read and printed correctly from memory.
- The 'small length clause can be applied to fixed point types.
- The size of a fixed point type may be altered with 'size and 'small representation specifications.

CHAPTER 3

OVERVIEW OF THE VERDIX ADA DEVELOPMENT SYSTEM

3.1 OVERVIEW OF VADS COMPONENTS

The Verdix Ada Development System (VADS) is centered on a high-performance Ada compiler that is fully compliant with ANSI/MIL-STD-1815A. The system includes the software components for library management, compilation, program generation, object file analysis, debugging, source code formatting, as well as additional tools and libraries.

First, in this overview, the tools are listed in groups according to their functional classification. A general introduction to each group and to the tools themselves follows.

The next sections of the *USER'S GUIDE* present a detailed description of each tool and its use, organized according to the functional classifications of the tools as listed. Some tools have multiple functions and are mentioned where appropriate.

3.1.1 Library Management Tools

a.cleanlib	reinitialize library directory
a.du	summarize disk usage for Ada libraries
a.info	list or change VADS library directives
a.ls	list compiled units
a.mklib	create a library directory
a.vadsrc	display available VADS versions and create a default library configuration file
a.path	report or change a VADS library search list
a.rm	remove an Ada source file or unit from a library
a.rmlib	remove a compilation library
a.tags	create a source file cross reference of units
a.which	find a compiled unit

3.1.2 Compiler

ada	invoke the VADS Ada compiler
a.error	analyze and disperse error messages

3.1.3 Program Generation Tools

a.ld	build an executable program from previously compiled units
a.make	recompile source files in dependency order

3.1.4 Analysis Tools

a.das	disassemble object files

3.1.5 Debugger

a.db	debug Ada source code programs

3.1.6 Source Code Formatter

a.pr	format source code

3.1.7 Additional Tools

a.help	interactive help utility for VADS
a.list	produce a program listing
a.report	report VADS deficiencies

3.1.8 Supplied Libraries

VADS provides the Ada standard library (**standard**), as defined in the Ada Reference Manual (Ada RM), including **package** MACHINE_CODE and runtime support. Additional packages and libraries are specific to product type, target processor, and version.

See also *Implementation Reference, 18.2 Release Directory Structure,* page 18–2.

3.2 LIBRARY MANAGEMENT TOOLS

VADS requires all compilations to take place in an Ada library. Any directory may become such a library, called a VADS library in this manual. Ada program libraries may reference other libraries so that units not local to a VADS library can be accessed during compilation. This organization enables programmers to work on local versions of individual program units while retrieving the remainder of the program from previously developed libraries. The tools **a.cleanlib**, **a.info**, **a.mklib**, **a.path**, **a.rm**, **a.rmlib**, and **a.vadsrc** are provided to maintain these structures.

Effective library management and disk resource use requires reporting and referencing tools. The tools **a.du**, **a.info**, **a.ls**, **a.tags**, and **a.which** can be used to ease source management on large projects. These tools help programmers use the separate compilation facilities that VADS and Ada offer while generally making it unnecessary to remember the source file or library in which a particular unit specification or body resides. Directives can be inserted into the library that denote dependencies upon object files compiled from other languages or change default compiling or linking options in the library.

3.3 COMPILER

The optimizing Ada compiler, **ada**, uses several different error-recovery techniques to maximize the number of errors found per compilation while reducing spurious error messages. Users can request that error messages be placed as comments into the source file so that each error and its location can be identified exactly. For semantic and most general errors, messages identify the relevant section and paragraph of the Ada RM.

Since the normal software development process involves iterative editing, compiling, testing, and debugging, various options in the compiler and debugger provide for easy transition from one phase to another. For example, a single command can compile a program, invoke the error processor, and call the editor with error messages interposed in the program.

3.4 PROGRAM GENERATION TOOLS

VADS includes several methods for generating executable programs from compiled sources. A prelinker, **a.ld**, is provided to verify and create an executable image of all component units required for a given main unit. **a.ld** can be invoked automatically by either the compiler or the dependency recompilation tool **a.make**.

In compiling programs consisting of more than a few files, changing a unit upon which another unit depends can often lead to inconsistent results, and maintaining compilation order tables by hand for proper reconstruction of the latest program version is difficult and error-prone. **a.make** brings all specified units and their dependents up to date by recompiling them in the required order.

3.5 ANALYSIS TOOLS

a.das provides disassembly of object files with source code interspersed.

3.6 DEBUGGER

The debugger, **a.db**, provides a fully symbolic debugging facility with both screen-oriented and line-oriented presentation modes. The user can access entities by name. The debugger displays objects in their Ada source representations; for example, it displays enumeration values in symbolic form and recognizes the subcomponents of aggregates such as records and arrays.

The debugger has breakpoint capabilities that permit the user to suspend program execution at a specific statement, at the beginning of a subprogram, or at the return from a call. Single-step execution is provided with options for stepping *over* or *into* subprogram invocations. Breakpoints can also be made conditional on variable values, and complex debugger commands can be executed from command files. Variables, memory, or machine registers can be examined and modified, and programs can be executed any number of times without leaving the debugger.

During a debugging session, the debugger provides access to both Ada source code and machine instructions. These facilities are integrated with the breakpoint and stepping capabilities so the user has a 'window' into the program source and machine code at all times. Although a user program may be disassembled using the debugger, it is often desirable to view program disassembly without requiring the target processor board. VADS includes **a.das**, a stand-alone disassembler.

Since much of debugging involves alternating attention between the executing program and the program source code, **a.db** supports a screen oriented interface that keeps the relevant source code always visible without the need to continuously type list commands.

On-line help facilities are available while using the debugger.

3.7 SOURCE CODE FORMATTER

VADS includes a source code formatter or 'pretty printer', **a.pr**, with options for tailoring code to individual Ada coding standards.

a.error and **a.list** provide additional listing facilities for Ada source code.

3.8 ADDITIONAL TOOLS

Online help is available for VADS tools and debugger commands using **a.help**.

a.list provides a program listing with or without numbers for programs containing no errors. (**a.error** produces listings for programs with errors.)

a.report interactively helps in submitting a problem report description to Verdix.

CHAPTER 4

LIBRARY MANAGEMENT TOOLS

4.1 OVERVIEW

This chapter describes general Ada characteristics relating to libraries, introduces the VADS library directory structure, and describes the VADS library management tools that enable the user to maintain library directories and source code. Libraries supplied with VADS are then introduced.

4.2 SEPARATE COMPILATION INFORMATION

An Ada compiler is different from compilers in most other languages in that each Ada compilation must make direct use of information that has been collected from previous compilations. This information is called 'separate compilation information' and is collected and maintained in an Ada library.

VADS maximizes the economy that the Ada separate compilation capability brings to a large programming effort. Verdix Ada allows a large program to be subdivided into and developed as discrete units and enables each unit to be compiled alone. Individual units that are ready can be compiled before the project is complete, and recompilation of individual units can be done without affecting non-dependent units. Separate compilation can significantly reduce software development time.

Ada encourages separate compilation and defines library mechanisms to deal with unit dependencies and subprogram specifications and bodies in separate files. VADS provides simple tools for referencing previously compiled units in other libraries and allows large programs to be designed with a modular approach. Programmers can each work on one facet of the project using previously developed units from libraries other than their own. When the local portion of the project is completed, others can be given access to the current library with VADS library tools.

VADS builds on these Ada features by providing a program library system that allows programmers to access units that are not contained in their local library. The Ada rules governing unit dependencies still apply. Each VADS library has a list of other VADS libraries to be searched whenever a unit cannot be found locally. This information is maintained in the library search list of the **ada.lib** file. The specified libraries are searched in the order listed in the library search list.

4.3 ADA COMPILATION UNITS

Each Ada source file contains the text for a single Ada compilation which, by definition, consists of one or more compilation units.

A compilation unit can be any of the following.

subprogram declaration
generic declaration
package declaration
generic instantiation
subprogram body
library unit body (subprogram body or package body)
subunit (subprogram body, package body, or task body)

Compilation units in an Ada source file are compiled in the order in which they appear in the file.

4.4 ADA SOURCE FILE STRUCTURE

Ada source files are ordinary text files and can be edited using any text editor.

The VADS compiler requires Ada source file names to end in the **.a** suffix as shown in the following examples.

factorial.a
Math_Package.a

VADS imposes no restriction on the location of subprogram specifications and bodies. It is often useful to place them in separate files or even distributed among several program libraries.

Source files must be compiled within a VADS library directory. A VADS library directory can be created anywhere in the file system by using the **a.mklib** utility discussed later in this chapter.

4.5 UNIT DEPENDENCIES

When compiling a unit that references other units, the compiler will automatically check that each reference is valid by checking actual data types in calling units against specified data types in the called subprogram.

To permit this checking, Ada program units must be compiled in proper order. This means that if one unit depends on specifications or definitions provided in another unit, the unit containing those specifications or definitions must have been compiled first.

See also Ada RM, *10.3 Order of Compilation.*

The files **fact.a** and **prob.a** shown below illustrate a simple case of such a dependency. When the file containing FACTORIAL (**fact.a**) is compiled, a record is made in a library that a function named FACTORIAL can now be called and that FACTORIAL requires an INTEGER argument and returns an INTEGER result.

```
-- file: fact.a --
function FACTORIAL ( N: INTEGER ) return INTEGER is
begin
   if N <= 0 then return 1;
   else return N * FACTORIAL ( N - 1 );
   end if;
end FACTORIAL;
```

Another compilation, the file **prob.a** that uses FACTORIAL, is shown below. The separate compilation information from the previously compiled unit FACTORIAL will be used to check that it is called with parameters of the correct number and type in the file **prob.a**.

```
-- file: prob.a --
with FACTORIAL;
function PROBABILITY ( NUM_ITEMS : INTEGER ) return FLOAT is
begin
   return 1.0 / FLOAT ( FACTORIAL ( NUM_ITEMS ));
end PROBABILITY;
```

4.6 VADS LIBRARY DIRECTORIES

VADS maintains separate compilation information in ordinary directories and files. Ada program sources are standard ASCII text and can be created with any available editor. Binary files are used for the separate compilation information and the resultant object and intermediate files. All files associated with a particular library are placed in a VADS library directory containing the elements shown in the Figure 4–1.

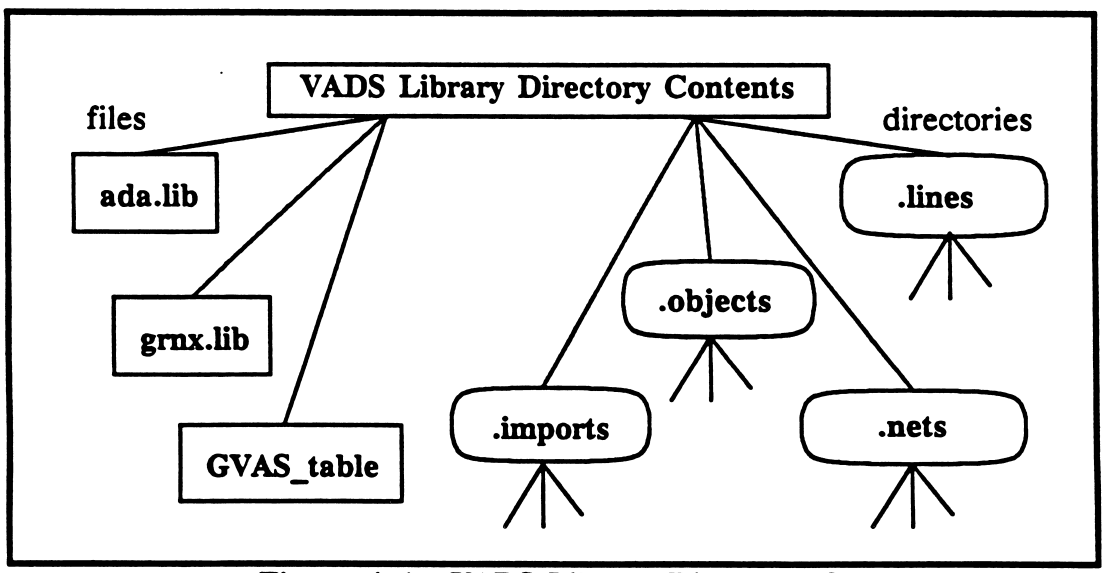


Figure 4–1. VADS Library Directory Contents

The terms *library* and *library directory* are interchangeable; however, in the following discussion, *library* will be used to emphasize the Ada aspects, and *library directory* will be used to emphasize the file system aspects of the concept.

The **ada.lib** file contains the mappings from Ada library unit names to file names for the corresponding separate compilation information. VADS maintains this information in a virtual area called the *Global Virtual Address Space* (GVAS). The **GVAS_table** file keeps track of which virtual addresses are free. The attachment of a virtual address to each compiled unit allows the VADS compiler to reuse information about previously compiled units rapidly. **ada.lib** also contains a library search list giving libraries besides the current one that will be searched for Ada units not found locally.

Many VADS tools use the library search list, for example **ada**, **a.ld**, and **a.db**. The VADS tool, **a.path**, will print out the library search list of the current library. The current Ada library will not be printed as part of the search path; however, it is understood that the current library will always be searched first.

An example search list printed by **a.path** is shown here.

```
library search list:
        /usr2/babbage/code
        /usr2/vads5/verdixlib
        /usr2/vads5/standard
```

The **gnrx.lib** file contains a descriptor for each generic body, for each request for a generic instantiation, and for each actual generic instantiation. This file, like **GVAS_table** and **ada.lib**, is maintained and used by the compiler.

The **.imports** directory contains files for units that have been imported from other libraries. The **.nets** directory contains files for each compiled unit holding the separate compilation information itself. VADS uses the DIANA intermediate representation for each unit and stores this representation in the directory **.nets**. These files are referred to as 'net' files in this document. The **.objects** directory contains the generated object files, and the **.lines** directory contains line number reference files for use by the debugger.

VADS libraries sometimes contain an additional component, **name_lib**. This component is used to map the longer object names to names of a length acceptable to the system linker. **name_lib** is created, when necessary, by the compiler front end.

4.7 LIBRARY MANAGEMENT TOOLS

Several VADS utilities are provided to manipulate VADS library directories.

a.cleanlib	reinitialize library directory
a.du	summarize disk usage for Ada libraries

a.info	list or change VADS library options
a.ls	list compiled units
a.mklib	create a VADS library directory
a.vadsrc	display available VADS versions and create a default library configuration file
a.path	list or change VADS library search list
a.rm	remove an Ada source file or unit from a library
a.rmlib	remove a compilation library
a.tags	create source file cross reference of units
a.which	find a compiled unit

4.7.1 a.cleanlib

a.cleanlib preserves all non-compilation information contained in **ada.lib**, including any additional libraries contained in the library search list and any other directives found in **ada.lib**.

a.cleanlib empties the files **ada.lib**, **gnrx.lib**, and **GVAS_table** of all separate compilation information and removes the contents of the directories **.imports**, **.nets**, **.lines**, and **.objects** from the named library or, if no library is specified, from the current library directory. All library directives in the **ada.lib** are preserved by **a.cleanlib**. It removes the file **name_lib** if present.

The command is invoked as shown below.

```
a.cleanlib [-f -F] [VADS_library]
```

If any library component is missing, **a.cleanlib** will terminate without removing any information unless the **-f** option is given.

a.cleanlib normally prohibits the cleaning of libraries named **standard**, **verdixlib**, or **publiclib**. The **-F** option overrides this restriction.

The command *must* include the **-f** option to reinitialize a VADS library if its structure has become corrupted because of some operational disruption (e.g., a system crash).

4.7.2 a.du

a.du lists sizes in bytes of files associated with each library unit (files in the **.nets**, **.lines**, and **.objects**) in the named library or the current library if unspecified. Each net file is checked for validity.

The invocation of the command is as follows.

```
a.du [options] [VADS_library]
```

Each option (and possible associated arguments) should be separated on the command line with spaces.

The **-e** option is useful for providing disk usage information when the library net files have become corrupted, and the **-i** option provides information about units imported from other libraries for use in local compilation.

Sample output of the command

```
a.du
```

is shown below for a library containing a 'Towers of Hanoi' program in three files. The numbers represent the sizes in bytes of the indicated files in each subdirectory.

```
.nets .objects .lines

   29952 4790  413 body of hanoi in ./hanoi.a

   9241  1910  197 body of terminal in ./termbody.a

   9084  666 52 spec of terminal in ./termspec.a
```

4.7.3 a.info

a.info is a tool for reporting or altering INFO and LINK directives in the current VADS library.

LINK and INFO directives are used primarily to add additional dependent files to links performed in this or a child VADS library. The additional files to be linked can be files compiled from another language, archive (ar) library files, or Ada objects whose source is not mentioned in a context clause in the local Ada source files.

a.info is invoked as shown below.

```
a.info [options]
```

a.info may be used interactively or with all information passed to it via the command line.

Directives have the format

name:*type*:*value*:

where *type* can be the word LINK or INFO, and *value* is usually a filename. The **ada.lib** example shown below contains six WITH (in the WITH*n* format) and one INFO directive.

```
!ada library
ADAPATH= /rc/vads/verdixlib /rc/vads/standard
TARGET:INFO:SELF_TARGET:
VERSION:INFO:5.7:
WITH1:LINK:/vc/config/.objects/config01:
WITH2:LINK:/vc/config/.objects/config_b01:
WITH3:LINK:/vc/config/.objects/intr_sup01:
WITH4:LINK:/vc/config/.objects/intr_sup_b01:
WITH5:LINK:/vc/config/.objects/serial_sup01:
WITH6:LINK:/vc/config/.objects/serial_sup_b01:
configuration:NLPS:config01:
configuration:NLPB:config_b01:
serial_support:NLPS:serial_sup01:
intr_support:NLPS:intr_sup01:
intr_support:NLPB:intr_sup_b01:
serial_support:NLPB:serial_sup_b01:
```

With the *regular_expression* and **-value** *regular_expression* options, regular expressions are used to match *name* and *value* by **a.info** using **regex(3)**. Specifications for regular expressions are listed in the *UNIX Programmers Manual* under **ed(1)**.

For example, to add the LINK directive

```
WITH7:LINK:/vc/config/.objects/example
```

use the command:

```
a.info +link with7 /vc/config/.objects/example
```

and the command

```
a.info -info version
```

will delete the INFO directive

```
VERSION:INFO:5.7:
```

Without options, **a.info** displays all INFO and LINK directives and can be used to alter them with the + and - (add and delete) options.

With the **-l** (long) option, **a.info** gives the following type of information.

```
INFO Directive     TARGET -> SELF_TARGET
INFO Directive     VERSION -> 5.7
```

The verbose option **-v** supplies information of the following type.

```
TARGET
   INFO Directive
   value: SELF_TARGET
VERSION
   INFO Directive
   value: 5.7
```

For a complete list of directive names, see also *IMPLEMENTATION REFERENCE, 18.19.3 Supported INFO and LINK Directive Names*, page 18-35.

For a discussion of WITH*n* directives used with the prelinker, see also *PROGRAM GENERATION TOOLS, 6.3 a.ld*, page 6-1 and *VADS COMMAND REFERENCE*, **a.ld**, page 19-16.

4.7.4 a.ls

a.ls provides a list of the units compiled in the current VADS directory.

Options are provided to give more or less extensive information, to change the format of the list, or to provide a list of compiled units occurring in specified source files.

Additionally, *unit_name* can be specified as a regular expression to match groups of units. If the regular expression contains any of the shell's meta characters, the expression must be quoted.

a.ls is invoked as shown below.

```
a.ls [options] [unit_name] ... [-f ada_source.a ...]
```

With the **-l** or **-v** options, **a.ls** prints output in multiple columns. Use the **-1** (single) option to override this feature.

The **-F**, **-l**, and **-v** options, respectively, provide increasing amounts of information and are not to be used in combination. If more than one is given, the one providing the most information is used.

Output from the command

```
a.ls -l "*float*"
```

is shown below.

```
Sep 10 19:40                 Pack Spec (Shared)   float_io
Sep 10 19:40                 Pack Spec (Shared)   float_io
Sep 10 19:40  Aug 10 17:59   Pack Spec (Instan.)  float_io
Sep 10 19:47                 Pack Body (Shared)   float_io
Sep 10 19:47                 Pack Body (Shared)   float_io
Sep 15 18:20  Sep 18 20:05   Proc Body (Main)     my_float
```

For specifications of regular expressions, see also *UNIX Programmers Manual*, **ed(1)**.

4.7.5 a.mklib

Use of VADS begins with the creation of a VADS library directory using the **a.mklib** utility. As a rule, a new library directory should be created for each program or family of programs being developed. A VADS library directory is created using the command as shown below.

```
a.mklib [-F -f -v -i ] [new_VADS_library [parent_VADS_library]]
```

A directory with the name specified by *new_VADS_library* is created and referenced to the named *parent_VADS_library* so that units in the new library can reference all Ada library units defined in the parent. Because there may be more than one VADS compiler, **a.mklib** must be executed in a way that a *parent_VADS_library* is specified or implied. Here are some alternatives:

- the recommended way to use **a.mklib** is with the **-i** option to choose a VADS release interactively
- specify both *new_VADS_library* and *parent_VADS_library*
- have a **.vadsrc** file present in the current directory or in HOME

See also *[LIBRARY TOOLS], 4.7.6* **a.vadsrc** page 4–10.

The library hierarchies created through the repeated use of **a.mklib** are distinct from the operating system directory hierarchy, but use directory path names. Ada libraries can reside anywhere in the file system and need not stand in any particular relationship to one another. Thus, our use of the term *parent_VADS_library* refers only to the VADS perception of the library hierarchy.

A directory that is not a VADS library directory (e.g., **/usr2/babbage/code**) can be turned into a VADS library directory by the commands

```
a.mklib -i /usr2/babbage/code
a.mklib . parent_VADS_library
```

or, alternatively,

```
a.mklib -i
```

when /usr2/babbage/code is the current working directory.

If **/usr2/vads5/verdixlib** is specified as the *parent_VADS_library* or **/usr2/vads5** is chosen as the VADS release when using the **-i** (interactive) option, the library search list would be similar to that shown below.

```
library search list:
        /usr2/vads5/verdixlib
        /usr2/vads5/standard
```

Create a subdirectory, **/usr2/babbage/code/newdir**, with **/usr2/babbage/code** as the parent library directory by using the following command without changing directories.

```
a.mklib newdir /usr2/babbage/code
```

The library search list for **newdir** is shown below.

```
library search list:
        /usr2/babbage/code
        /usr2/vads5/verdixlib
        /usr2/vads5/standard
```

The current library is always understood to be searched first even though its name does not appear in the list. For any compilation executing in the library **/usr2/babbage/newdir** from the example above, that library is searched first for separate compilation information, then **/usr2/babbage/code**, then **/usr2/vads5/verdixlib**, and finally **/usr2/vads5/standard**.

Ada and VADS require that all library units have distinct names. A new unit will hide an existing unit in an ancestor library (i.e., a library on the ADAPATH) if it is given the same name. This should be avoided unless either a replacement of the existing unit or an temporary local version is intended. This feature can be used intentionally to test local modifications to code in ancestor libraries. Compilations explicitly requiring access to two units of the same simple name will terminate with the following fatal error message

```
RM 10.1(3): definitions of unit_name conflict: list
```

or will tell the user to recompile the sources so that all use the newer definition of *unit_name*.

The **-f** option forces initialization of the VADS library structure, overwriting any existing components. This option is useful when reinitializing a library following a system crash or some other experience that corrupts a library.

a.mklib normally prohibits the creation of libraries named **standard**, **verdixlib**, or **publiclib**. The **-F** option overrides this restriction.

The **-i** (interactive) option provides a list of all versions available on the system and prompts for a selection as shown below.

```
    Target Name        Version   VADS Location
1   SELF-TARGET        5.7       /usr2/vads
2   IAPX80386          5.5       /usr/vads_386
      Selection (q to quit):
```

The **-v** (verbose) option displays the library search list.

```
!ada library
ADAPATH= /usr2/vads/verdixlib /usr2/vads/standard
```

4.7.6 a.vadsrc

a.vadsrc is used to show available VADS versions or to create a default library configuration file and uses the following syntax.

```
a.vadsrc [-i]
```

a.vadsrc shows all versions of VADS available on the system, whether there is a **.vadsrc** file in HOME or in the current directory, and the contents of either or both such files. Sample output is shown below.

```
VADS versions available
   Target Name        Version   VADS Location
1  SELF-TARGET        5.7       /usr2/vads
2  IAPX80386          5.7       /usr2/vads_386
      Selection (q to quit):

/vc/sbq/.vadsrc:
No .vadsrc file present

./vadsrc:
   Target Name        Version   VADS Location
   SELF-TARGET        5.7       /usr2/vads
```

If the **-i** (interactive) option is given, the tool prompts for selection of a VADS version and creates a **.vadsrc** file in the current directory. This provides future invocations of **a.mklib** with a default version to use when no parent library is given. A **.vadsrc** file in the current directory overrides one in the user's home directory.

4.7.7 a.path

A list of libraries to be searched during compilation is maintained in the current VADS library directory in the file **ada.lib**. These libraries are sometimes referred to as remote libraries. **a.path** changes or reports the list of library names contained there. During compilation, any program units not found in the current library will be searched for in the VADS libraries on the search list. If the unit is not found in the first VADS library, it is searched for in the second, and so on in listed order.

The command

```
a.path [options] [VADS_library1 [VADS_library2]]
```

is used to display or change a VADS library search path.

When compiling a unit that references an externally compiled unit, the separate compilation information for the latter unit must be found and 'imported' into the current library. The ordered search list for these other units is maintained in the **ada.lib** file. The library search list is initialized by **a.mklib**, but it can be subsequently modified or examined with **a.path**.

Using **a.path** with no options will report the current library search path on the standard output. The command

```
a.path
```

produces output of the following type.

```
library search list:
   /usr2/vads5/verdixlib
   /usr2/vads5/standard
```

The **-a**, and **-i** options are used to specify changes to the search list of directory paths. For example,

```
a.path -i /usr2/babbage/adalibrary
```

will insert **/usr2/babbage/adalibrary** at the beginning of the list.

Using the **-i** option with two library names will insert the new library before the second one already in the list, as illustrated below.

```
a.path -i /usr2/babbage/adalibrary /usr2/vads5/standard
```

Using the **-v** option with one of the options that alter the library search list reports the list after it has been changed.

Removing a library name from the list does not remove compilation information from the referenced libraries. It only changes the search path for later compilations.

The **-t** (target) option produces information of the following type.

```
library search list:
        /usr2/vads5/verdixlib
        /usr2/babbage/adalibrary
        /usr2/vads5/standard
target_name         version      vads_location
  SELF-TARGET         5.7          /usr2/vads
```

The current directory is always understood to be first in the library searching order, but is not displayed and thus cannot be removed or changed.

4.7.8 a.rm

a.rm removes all information associated with a given unit or source file *including the information for all dependent units*. The command can take any of the forms shown below.

```
a.rm [options] ada_source.a
a.rm [options] unit_name
a.rm [options] path/unit_name
```

Each option (and possible associated arguments) should be separated on the command line with spaces as shown below.

```
a.rm -v -b myfile.a
```

Executing **a.rm** removes all information associated with the named unit(s) or file(s) and all dependent units. When a unit is specified, the corresponding **.nets**, **.lines**, and **.objects** files are removed and the **ada.lib** entries are deleted for the affected unit(s). When a file name is specified, all net, object, and line number files are removed for each unit defined in the file, and the appropriate entries in **ada.lib** are removed.

A variety of options are provided to deal with the deletion of specifications and bodies separately. The **-u** option is provided for the single case that a unit A has been defined and is referenced in the source file with the selected component notation, such as MY_PACKAGE.A.

See also Ada RM 4.1.3

When **a.rm** removes all information associated with a given unit or source file including the information for all dependent units, it 'uncompiles' a unit or file and allows non-local units of the same name (if any) to be visible in the current library. It is also useful when changing a particular unit's dependency or code portion declared **separate**.

The **-i** (interactive) option, which prompts for confirmation before deleting information for any units, removes the **ada.lib** entry and all files associated with a unit with a single response.

4.7.9 a.rmlib

The **a.rmlib** utility

```
a.rmlib [-f -F] [VADS_library]
```

is used to remove all library components from *VADS_library* or the current library if no argument is given.

a.rmlib removes the **ada.lib**, **gnrx.lib**, and **GVAS_table** files and the directories **.imports**, **.nets**, **.lines**, and **.objects** from the named library. The library directory itself, all Ada source files, and other files and directories are left untouched. However, the Ada compiler can no longer be executed in this directory because the VADS library directory components created with **a.mklib** are gone.

If any library component is not present, **a.rmlib** will terminate without removing any files unless the **-f** option is given.

The **-F** option allows **a.rmlib** to remove library components from library directories that have a restricted name (**standard**, **primelib**, or **publiclib**).

4.7.10 a.tags

The **a.tags** command is analogous to the UNIX **ctags**(1) command. It provides cross-referencing of Ada units and, optionally, of types as well. **a.tags** creates a file named **tags** in the current library that contains an index of the locations of particular units. This file is used by **ex**(1) or **vi**(1) and allows a unit to be accessed for editing without knowing the file in which it is defined. **a.tags** is also used to provide similar listings on standard output.

a.tags takes the format shown below.

```
a.tags [options] ada_source.a ...
```

Using **a.tags** will replace any existing tags file unless the **-a** option is used.

Each option (and possible associated arguments) should be separated on the command line with spaces as shown below.

```
a.tags -t -a *.a
```

The **-x** and **-v** options provide indexed listings on the standard output. All other options refer to the file **tags** generated for use by **ex** or **vi**.

Execution of the command

```
a.tags -x *.a
```

while positioned in a VADS library containing Ada source code will produce a listing like the one below.

```
HANOI                 10 hanoi.a          procedure HANOI is
HANOI.DRAW_RING       38 hanoi.a          procedure DRAW_RING
HANOI.DRAW_START      72 hanoi.a          procedure DRAW_START is
HANOI.SOLVE           93 hanoi.a          procedure SOLVE
TERMINAL               2 termbody.a       package body TERMINAL is
s#TERMINAL             2 termspec.a       package TERMINAL is
```

The output gives the locations of packages, subprograms, generics, and task definitions in a group of files. The **-t** option adds types to the above list. Each line contains the name of the construct, line number, file in which it is defined, and the program text at that line.

The command

```
a.tags -t *.a
```

executed in the same VADS library as the previous example will create a file **tags** containing the same type of information, but in a form readable by **ex** or **vi** and containing entries for types.

The command

```
vi -t HANOI.SOLVE
```

will then call the **vi** text editor, locate the proper file, and place the cursor at the definition of the named type or subprogram ready for editing. While in **vi**, the command

```
:tag TERMINAL
```

searches for the file in which TERMINAL is defined, enters the file, and again places the cursor at the named type or subprogram. Alternatively, place the cursor on the first character of a subprogram and press <CONTROL>]. Some editors other than those supplied with the UNIX operating system support the initial **-t** option and similar commands as well.

Ada specifications are named by prefixing the Ada name with **s#** and bodies are named with the unmodified Ada name. Stubs for separates are named by prefixing the Ada name with **stub#**.

Nested packages, subprograms, types, generics, and task definitions are always listed with their full, Ada expanded name with any tag prefaces added to the simple name. Simple names for nested units are listed only if the simple name is unique across all other tags.

Overloaded subprograms within a file are not differentiated. However, the tag can identify the correct file, and repeated application of the search pattern (the 'n' key in vi) will find the desired subprogram.

The search pattern is generalized to match all versions of the overloaded subprogram; this generalization can cause the pattern to match things other than the desired unit. When initially starting **ex** or **vi** with the **-t** (tags) option or **:tag** command, the desired unit or type *must* be in the same case (upper or lower) as it appeared in the source file.

4.7.11 a.which

The command

```
a.which [options] [unit_name]
```

or

```
a.which [options] [path/unit_name]
```

gives the name of the file that defines the version of the referenced unit that is visible in the current library or the library indicated in *path*.

When positioned in a VADS library, the command

```
a.which text_io
```

produces the following sort of information.

```
/usr2/vads5/standard/text_io_spec.a
```

The command

```
a.which -b terminal
```

shows the location of **package body** TERMINAL.

```
./termbody.a
```

The library search sequence used to locate the unit can also be displayed with the -v option. For example, the command

```
a.which -v hanoi
```

produces the following results.

```
library search list:
.
/usr2/vads5/verdixlib
/usr2/vads5/standard
unit hanoi (spec of hanoi) is defined in ./hanoi.a
```

CHAPTER 5

THE COMPILER

5.1 OVERVIEW

This chapter describes the use of the VADS compiler, and compiler processing. It also describes use of the **a.error** utility to examine the different types of error messages produced by the compiler. Lock files and assembler output are discussed at the end of the chapter.

5.2 INVOKING THE COMPILER

The Ada compiler can be invoked once a VADS library directory has been established and Ada source files are present in the library directory.

The basic command for invoking the compiler is

```
ada [options] ada_source.a ... [linker_options] [object_file.o] ...
```

where ... may represent multiple files.

Each option (and possible associated arguments) should be separated on the command line with spaces as shown below.

```
ada -v -w myfile.a
```

If *ada_source.a* is a simple file name, the compiler expects the current directory to be an Ada library directory. For example, if the current directory is **/usr2/babbage/adaproject**, the command

```
ada sample.a
```

will ask the compiler to assume that **/usr2/babbage/adaproject** is a VADS library directory, and the compiler will look in that directory for a file named **sample.a**.

If a full pathname is provided, the named program can be compiled regardless of the current working directory. For example, the command

```
ada /usr2/babbage/adaproject/sample.a
```

will cause the compiler to search for and compile **sample.a** in the library directory **/usr2/babbage/adaproject**.

Since certain other VADS commands are useful only while positioned in the desired VADS library directory, the compiler is typically used in the same fashion.

5.3 COMPILER PROCESSING

For each unit of the compilation, the compiler first analyzes the source text to ensure that Ada's lexical and syntax rules have been observed. The compiler then determines which additional units are needed from this or other VADS library directories to supply any required separate compilation information.

For instance, in the **fact.a** and **prob.a** examples used in *LIBRARY MANAGEMENT TOOLS, Unit Dependencies*, the file **fact.a** containing FACTORIAL is compiled. A record is made in the containing library that a function named FACTORIAL can now be called and that FACTORIAL expects an INTEGER argument and returns an INTEGER result. A compilation of the file **prob.a** makes use of FACTORIAL. Since the Ada source file **prob.a** references a previously compiled Ada unit FACTORIAL, the command

```
ada prob.a
```

will require the compiler to find the unit named FACTORIAL to complete the compilation of PROBABILITY.

The compiler will first look in the current Ada library directory for an entry for FACTORIAL in **ada.lib**. If the entry is not found, the compiler will search, in turn, each of the additional Ada library directories identified in the library search list. If it cannot find FACTORIAL, the compiler will report an error. If an entry is found, the compiler will identify the file in the **.nets** subdirectory containing the corresponding separate compilation information for FACTORIAL. If this file does not exist or has been damaged, the compiler will report an error.

The compiler then proceeds with a semantic analysis of the current unit and reports semantic errors as appropriate. If there are no errors, the compiler generates code, and the library directory is updated. Note that the library is updated both with the separate compilation information and the linkable object code on a unit-by-unit basis. Thus, as Ada requires, even if some units in a multiunit compilation have errors, those units without errors will still be compiled and placed in the library.

5.4 COMPILER ERROR MESSAGES

The compiler writes all error messages to the standard error stream, **stderr**. The categories of error messages are listed below, followed by descriptions of each.

lexical errors
syntax errors
semantic errors
general errors
warnings
fatal errors
internal errors

Lexical errors are errors in the formation of literals, identifiers, and delimiters. The compiler will perform no semantic analysis on a unit containing lexical errors, but will attempt to correct the error internally to minimize its impact on the discovery of further lexical and syntax errors.

Syntax errors are errors in the formation of grammatical constructs. The compiler will perform no semantic analysis on a unit containing syntax errors, but will attempt to correct an error to minimize its impact on the discovery of further lexical and syntax errors.

Semantic errors are those made in the usage of Ada language constructs. The compiler will generate no code for units with semantic errors and will not update the library. It will generate code and update the library for a unit that is error-free even if other units in the file have errors. All semantic error messages refer to the specific section, subsection, or paragraph within the Ada RM.

General errors are semantic in nature, but do not fall within a specific Ada RM reference. However, it will generate code and update the library for a unit that is error-free, even if other units in the file have errors.

Warnings are errors not sufficiently serious to prevent code generation or that indicate questionable use of a construct or dangerous programming practice.

Fatal errors are those of such severity that meaningful recovery is impossible and compilation of the file stops. After checking that the fatal error is not due to accidental misuse of VADS tools (e.g., attempting a compilation in a non-VADS library directory), fatal errors should be reported to Verdix Ada Support Office (APSO) as described in the *Installation Guide, 17.3 Customer Reports*, page 17-5.

Internal errors are occasionally produced by Ada programs containing unusual semantic errors, but may also occasionally be due to faults within the compiler. Internal errors are most often of the form

```
internal assertion error at file p_sem.c, line 394
```

or

```
internal case error at file p_sem.c, line 546
```

When an internal error occurs, its complete text and the program construct where it occurred should be reported to APSO.

5.5 COMPILER ERROR MESSAGE PROCESSING

a.error is usually called directly from the compiler using the options provided for **ada**. It reads the specified file or the standard input, determining the source file(s) containing errors and processing the errors according to the options specified in a command of the following format.

```
a.error [option] [error_file]
```

The **-e** and **-ev** options of the **ada** command automatically call **a.error** to process any compiler error messages resulting from the current compilation.

Note that **a.list** produces a program listing closely resembling the output of **a.error** with the **-l** option for programs containing no errors.

Compiler output can be redirected to a file and examined with the aid of the **a.error** utility or be piped directly to **a.error**. Options for **a.error** are shown below.

To illustrate, a file containing errors, **badtry.a**, is compiled.

```
-- file: badtry.a --
with TEXT_IO;
procedure BADTRY is
   subtype T is range 1..1f;
   COUNT : T;
   SUM : INTEGER;
   type REAL is digits 6;
   AVG : REAL
begin
   for COUNT in T loop
   SUM := SUM + I;
   end loop;
   AVG := SUM / COUNT;
   TEXT_IO.PUT(INTEGER'IMAGE(SUM));
   TEXT_IO.PUT(REAL'IMAGE(AVG));
end MAIN
```

The output from the compilation of **badtry.a** is redirected using the following command from **sh**(1).

```
ada badtry.a > badtry.errors 2>&1
```

The file **badtry.errors** contains the following messages. Each line that contains an error is listed and followed with a description of the errors that were found.

```
/usr2/babbage/vads/badtry.a, line 5, char 28: lexical error: deleted
/usr2/babbage/vads/badtry.a, line 5, char 18: syntax error:
"identifier"
  inserted
/usr2/babbage/vads/badtry.a, line 10, char 3: syntax error: ";"
inserted
/usr2/babbage/vads/badtry.a, line 17, char 7: syntax error: RM 6.3(4):
  subprogram was given a different name
```

With the **-v** option, **a.error** will write the error messages directly into the original source file as comments and call the vi text editor. Line numbers will be suppressed, error messages marked with the pattern ###, and the editor positioned in the file with the cursor at the point of the first error. The error description includes one or more error messages that begin with a capital letter and point to the place in the program where the error was detected. Subsequent lines beginning with corresponding letters provide brief synopses of the errors encountered as shown below with the file **badtry.a**.

```
   -- file: badtry.a --
      with TEXT_IO;
   procedure BADTRY is
      subtype T is range 1..1f;
------------------^A                                          ###
----------------------------^B                               ###
--### A:syntax error: "identifier" inserted
--### B:lexical error: deleted
      COUNT : T;
      SUM : INTEGER;
      type REAL is digits 6;
      AVG : REAL
   begin
--^A                                                        ###
--### A:syntax error: ";" inserted
      for COUNT in T loop
      SUM := SUM + I;
      end loop;
      AVG: = SUM / COUNT;
      TEXT_IO.PUT(INTEGER'IMAGE(SUM));
      TEXT_IO.PUT(REAL'IMAGE(AVG));
   end MAIN;
------^A                                                     ###
--### A:syntax error: RM 6.3(4): subprogram was given a
--### different name
```

The file contains four lexical and syntax errors. First, an identifier naming a type was omitted before the keyword **range**. The compiler continues as though this identifier were inserted, but does not, of course, edit the original source file. The next error is a lexical error, resulting from the malformed integer literal '1f'. The compiler continues as though the 'f' were deleted. The remaining error messages show that a semicolon should have preceded **begin** and that the designator after **end** has a different name than was given to the subprogram.

Error messages are marked with ### so that errors can be located easily and subsequently deleted. It should also be noted that all error message lines are prefixed with —, which denotes an Ada comment. Thus even if **a.error -v** has been used to intersperse error messages into a file, the compiler can still process that file without deleting the error messages. Since -v places the error messages directly in the source file, if **a.error -v** is called again before the messages are deleted or the error corrected, a second copy of the same messages will appear.

Since all error lines are flagged with ###, the **vi** command

```
:g/###/d
```

can be used to delete them. Any source lines with ### in them will also be deleted; consequently, it is advisable not to use ### in source code.

The file **badtry.a** can now be edited to repair the lexical and syntax errors and resubmitted to the compiler. If those errors are fixed correctly, semantic analysis can proceed.

5.6 LOCK FILES

Since the compiler updates libraries as part of the compilation process, it must prevent other compilations from accessing a library during certain portions of the procedure. This feature will usually be invisible to the user although it can cause longer compilation times if several compilations are sharing libraries.

System failures or other operational disruptions can cause the locking mechanism to remain in effect. The mechanism consists of advisory files whose presence prevents some part of the compiler from writing to a specific library file. If the compiler finds a lock file (files ending in **.lock**) in the required VADS library, it will wait and look again. If after several such tries the lock file remains, the compiler will issue a message such as the one shown below.

```
/usr2/babbage/han.a/termspec.a, line 170, char 4: fatal: timeout.
Library ada.lib is locked (lock file is /usr2/babbage/han.a/ada.lock)
```

The compiler message names the specific lock file at the end of the message. To proceed, remove the lock file and restart the compilation. For the preceding above, typing

```
rm ada.lock
ada termspec.a
```

will restart the compilation process.

5.7 ASSEMBLER OUTPUT

Assembly-code listings are not produced by the compiler. The utility **a.das** is provided for producing an assembler listing of a compiled object. A similar listing can be produced using the debugger, **a.db**.

See also *ANALYSIS TOOLS, 7.2* **a.das**, page 7-1 and *DEBUGGING, 8.11.3 Display Instructions:* **li, wi**, page 8-11.

CHAPTER 6

PROGRAM GENERATION TOOLS

6.1 OVERVIEW

Three tools — **ada**, **a.ld**, and **a.make** — are available to build executable programs and are described in this chapter. The procedure for executing compiled programs is also introduced.

6.2 ada

As discussed in *USER'S GUIDE, VADS Compiler*, **ada** invokes the compiler front end and code generator by default. The **-M** option can be used to call **a.ld** to link an Ada program. The command

```
ada -M main hello.a -o hello
```

will, if executed in the directory containing **hello.a**, call **a.ld** on the procedure and produce an executable file **hello**.

```
--file: hello.a --
with TEXT_IO; use TEXT_IO;
procedure MAIN is
begin
   put ("Hello world!");
   NEW_LINE;
end MAIN;
```

For a full description of the options available, **see also** *VADS COMMAND REFERENCE*, **ada**, page 19-2.

6.3 a.ld

a.ld is a prelinker that gathers information for each unit dependent on the 'main' unit. It verifies that all units are up to date, creates a list of commands, names, and options with which it invokes the linker and gathers elaboration and exception information into an object file.

In the command

```
a.ld [options] unit_name [ld_options]
```

unit_name is the main program and must name a non-generic subprogram. If *unit_name* is a function, it must return a value of **type** STANDARD.INTEGER.

Options intended for the UNIX linker, **ld**(1), should be placed after *unit_name*. *ld_options* can refer to archive libraries, option switches, object files, or library abbreviations. For example, the **a.ld** line for a compiled Ada unit that makes use of **pragma** INTERFACE and references mathematics functions might look like the following.

```
a.ld my_unit c_function.o  lscr.a -o my unit
```

6.3.1 WITHn Directives

a.ld reads instructions for generating executables from the **ada.lib** file in the VADS libraries on the search list. Besides information generated by the compiler, these directives also include WITH*n* directives that allow the automatic linking of object modules compiled from other languages or Ada object modules not named in context clauses in the Ada source. Any number of WITH*n* directives may be placed into a library, but they must be numbered contiguously beginning at WITH1. The directives are recorded in the library's **ada.lib** file and have the following form.

```
WITH1:LINK:object_file:
WITH2:LINK:archive_file:
```

WITH*n* directives may be placed in the local Ada libraries or in any VADS library on the search list.

A WITH*n* directive in a local VADS library or earlier on the library search list will hide the same numbered WITH*n* directive in a library later in the library search list.

Use **a.info** to change or report library directives in the current library.

For more information on using WITH*n* library directives, see also *LIBRARY MANAGEMENT TOOLS*, **a.info**, page 4-6.

6.3.2 Foreign Object Files

Object files created by other non-VADS compilers or assemblers can be linked with an Ada program. They can be introduced into the linking process using a WITH*n* directive.

Note that **a.ld** does not automatically link object files containing subprograms or objects referenced by **pragma** INTERFACE. The user is responsible for including required foreign object files in the link.

Ada modules that are included via a WITH*n* directive in the **ada.lib** should contain **pragma** SUPPRESS (ELABORATION_CHECK). Ada modules that are linked via a WITH*n* directive do not get elaborated and therefore must not depend on elaboration

code. **pragma** SUPPRESS suppresses the generation of elaboration checks for subprograms.

See also *IMPLEMENTATION REFERENCE, 18.4 PRAGMAS AND THEIR EFFECTS*, page 18-10 and *IMPLEMENTATION REFERENCE, 18.14 INTERFACE PROGRAMMING*, page 18-25.

6.3.3 Linker Options File and Runtime Library

a.ld searches for three LINK directives, TASKING, LIBRARY, and WITH*n*. These directives identify the Ada Runtime System library and additional object files to include in the executable.

The **standard** library provides default directives similar to the following

```
TASKING:LINK:/vc/release/standard/objects/tasking.a:
LIBRARY:LINK:/vc/release/standard/objects/library.a:
```

6.4 a.make

In compiling programs consisting of more than a few files, changing a unit on which another unit depends can often lead to inconsistent results, and maintaining compilation order tables by hand for proper reconstruction of the latest program version is difficult and error-prone. **a.make** automates this reconstruction by performing the minimal number of recompilations necessary to bring a unit, program, library, or group of libraries up to date after source changes.

The most general forms of the command are shown below.

a.make [*options*] *unit_name*

a.make [*options*] *path/unit_name*

For information on additional forms of the command including options passed to the system linker, **see also** *VADS COMMAND REFERENCE*, **a.make**, page 19-20.

unit_name can be prefaced by a VADS directory name (*path*) as though it were a file in that directory. This will result in **a.ld** setting its current library to that directory.

This utility determines which files must be recompiled to produce a current version of *unit_name*. It also calls **a.ld** to create the appropriate executable file if *unit_name* is capable of being a 'main' unit; otherwise, it just ensures that the named unit is up-to-date, recompiling any dependencies if necessary.

The VADS utility **a.make** is similar to the UNIX utility **make**(1). Both process dependency information that allows the recompilation of only those files containing

out-of-date objects. **a.make** generates such information automatically from the compilation of source files using Ada's **with** statements. In the Ada compiler, this dependency information can be used to determine which units need to be recompiled when some units are changed. For example, if unit IMAGE depends on PIXEL, IMAGE cannot be compiled before PIXEL, and IMAGE must be recompiled if PIXEL is changed. As long as the dependency information has been recorded by a previous compilation of each unit, the command

```
a.make image
```

will examine the current library and determine and automatically execute the minimal sequence of compilations necessary to build a new version of IMAGE that is up to date with the Ada units it depends on.

a.make uses the files in the **.nets** subdirectory to determine the correct order of compilation and elaboration. Unless the **-f** option is used, the utility will have no knowledge of any source file until that file has been compiled in a way that changes the program library. The **-f** option can be used to compile groups of files not previously entered into the library.

Unless the **-U** or **-D** option is used, the file must compile successfully or else the program library will remain unchanged. A single compilation is sufficient (unless syntax errors are present) if the **-U** option is used to force changes to the program library. In any case, syntax errors must be corrected before **a.make** will properly treat the source file.

The **-C** option is normally used to provide specific options to the compiler. For example, use a command of the following type to have **a.make** call the compiler with the optimizing option and invoke **vi** on compilation errors.

```
a.make -C "ada -O -ev" [other commands]
```

The **-f** option is useful when compiling Ada source files originally developed elsewhere. For example, assume the main unit of the project to be CALCULATOR. Simply create a new VADS library with **a.mklib**, copy in the source files (changing their suffixes to **.a** if necessary) and issue the following command.

```
a.make calculator -o calculator -f *.a
```

All command line arguments following **-f** are considered by **a.make** to represent source file names; consequently, it must be the last argument on the command line. This will generate the executable file **calculator** in the current directory, all dependencies having been properly interpreted by **a.make**.

Options unknown to **a.make**, such as the **-o** option in the CALCULATOR example above, are passed to **a.ld**.

6.5 EXECUTING COMPILED PROGRAMS

A compiled program can be executed under UNIX by typing its name. If the executable image is not in the current directory, it can be invoked by listing the directory containing the executable in the $PATH variable or by typing its full pathname,

```
/usr/babbage/adaprogs/a.out
```

CHAPTER 7

ANALYSIS TOOLS

7.1 OVERVIEW

An additional tool for program analysis is provided. **a.das** disassembles a compiled Ada object into machine instructions.

Without the **-s** option, **a.das** disassembles the body of the indicated unit.

7.2 a.das

a.das, the stand-alone object disassembler, is often used in development as part of the edit-compile cycle to examine the compiler output particularly for machine code procedures.

```
a.das [options] [path/]unit_name
a.das [options] -n object_file
```

CHAPTER 8

VADS DEBUGGER

8.1 OVERVIEW

The VADS symbolic debugger is used to debug Ada programs at the source level. It can also be used as an aid to understanding code written by others. Basic **a.db** features, concepts, and commands are presented.

a.db includes a line-mode and a screen-mode interface. The screen-mode debugger provides a more convenient interface for most program debugging at the source level. All commands discussed for the line-mode debugger are available in screen mode by prefacing them with a colon. The colon serves as a line-mode escape from screen mode. Also, many line-mode commands are directly available in screen mode.

Commands are discussed in the following order.

display commands
execution commands
breakpoint commands
positioning commands
screen-mode commands

Documentation for **a.db** refers frequently to the INTERRUPT key (<INTR>). The system command **stty** allows this function to be assigned to any convenient key (often <CONTROL>**c**).

See also Operating system documentation, **stty**.

This chapter provides a functional introduction and an explanation of most commands; detailed information is provided in the *DEBUGGER REFERENCE*.

Debugger register value output is shown in some examples. The actual registers shown and their names will vary depending on the target processor.

8.2 USING THE DEBUGGER AS A LEARNING TOOL

The VADS symbolic debugger is primarily viewed as a tool for tracking down runtime errors in programs, but its ability to relate the source code to the flow of execution also makes it a useful tool when modifying or debugging code that someone else has written. The following paragraphs outline an approach to using the debugger in screen mode that provides a 'window' through which to view the operation of another programmer's code.

Provided that the Ada libraries containing the source code are on the library search list, the debugger's **e** (enter) command can be used to view selected routines without having to determine the location of their source files. Furthermore, the **s** (step) and **a** (advance) commands can be used to walk through the code as it executes. This lets the programmer read through the source code in execution order and limits the code to be read to that of special interest. If a subprogram is accidentally stepped into, the **:bd** (break down) command can be used to set a breakpoint immediately after call to the subprogram, followed by the **g** (go) command to reach the breakpoint.

As the programmer begins to see how the code works, there are portions that require more careful examination. The **b** (set break) and **d** (delete break) commands allow selection of areas that can then be quickly reached using either the **g** (go) or **r** (run) command.

When a breakpoint is reached, the current state of the program can be examined. The print commands **p** and **P** print the values of entire variables or selected components of variables. The **cs** (call stack) command can be used to display the current call stack and the values of the arguments passed to each routine. Furthermore, the **cd** (call down) and **cu** (call up) commands provide the ability to move the entire debugging context to another routine on the call stack so that the source code and variables in those routines can be examined. After moving along the call stack, the **ct** (call top) command can be used to return to the top of the stack.

8.3 INVOKING THE DEBUGGER

Invoke the debugger with a command of the form shown below.

```
a.db [options] [executable_file]
```

If no options are specified, **a.db** begins in line mode and searches for an executable file **a.out**.

In addition to the invocation line, parameters to **a.db** can be supplied through **set** commands in the **.dbrc** (debugger runtime configuration) file.

See also *[VADS DEBUGGER], 8.10 Modify Debugger Configuration:* **set**, page 8-9.

VADS_location/**bin/a.db** is a 'wrapper' program that executes the correct debugger executable based upon directives visible in the **ada.lib** file. This permits multiple VADS compilers to exist on the same host. Only one VADS **bin** directory need be listed in the user's $PATH variable and changing from one to another VADS version on the same host is accomplished simply by executing **a.db** in a library created for a particular release.

8.3.1 Command File Input: -i

The **-i** option tells the debugger that its initial input will not be from the keyboard, but will instead be from a specified file. When the debugger commands in this file have all been read and executed, the debugger switches to reading from the keyboard unless the file contains a **quit** or **exit** command. The commands can be any line-mode debugger commands.

8.3.2 Ada Library Directory: -p

If the executable program is an Ada program, the debugger needs the name of the Ada library directory where the program was built. Normally programs are debugged from within the Ada library where they were built, so the debugger checks the current working directory to see if it is an Ada library directory. If the current working directory is not an Ada library directory or if it is not the Ada library directory where the program was built, the **-p** option can be used to supply the debugger with that information.

If the current working directory is not an Ada library and no **-p** option is used, the debugger will search the directories in a **set source** command, obtained by reading the **.dbrc** file, for the first directory that contains an **ada.lib** file. That directory is used by the debugger to obtain information from the **ada.lib** file, the separate compilation information produced by the compiler.

8.3.3 Executable File

If an executable file is given, the debugger uses the $PATH environment variable to find its full path name if the full path is not specified. Note that if the current directory is not in $PATH, executables in it cannot be found without specifying a full or relative pathname. If no executable file name is given at invocation, **a.db** will debug **a.out** by default.

8.3.4 Display Debugger Executable: -sh

The **-sh** option prints the name of the actual debugger executable file. Based on directives in the **ada.lib** file, the wrapper program, *VADS_location*/**bin/a.db**, finds the correct debugger for the Ada library.

8.3.5 Screen-mode Invocation: -v

Use the **-v** option to invoke the debugger in screen mode. When in screen mode, Q returns to the line mode.

The screen-mode debugger provides a more convenient interface than the line-mode interface for most program debugging at the source level. All commands discussed for the line-mode debugger are available in screen mode as well by prefacing them with a colon.

The screen-mode interface includes additional commands that do not require a preceding colon.

See also *[VADS DEBUGGER], 8.15.3 Screen-mode Window Commands, page 8.15.3* and *DEBUGGER REFERENCE, screen mode,* page 20-63.

8.3.6 Redirect Debugger Input and Output

The debugger will need input from a file rather than the terminal. This option allows you to run the debugger in background. If you like, the output which would normally be written to the screen may be directed to a file.

A sample invocaton line is:

```
a.db my_prog < debug.in >& debug.out &
```

There are two restrictions:

1. You cannot use screen mode when debugger input is a file.
2. Your program cannot share the debugger's input file. Use **set input** *filename* or **set run >** *filename* to set the input file for your program. A sample *debug_in* file is:

```
set input my_prog.in
r
quit
```

8.4 Halting the Program

The program being debugged can be stopped by setting a breakpoint or pressing the key mapped to <INTR>.

8.5 Terminating the Debugger Session

Type **quit** or **exit** in line mode or **:quit** or **:exit** in screen mode to leave the debugger.

8.6 On-line Debugger Help

Access on-line help for **a.db** and VADS utilities during a debugging session by typing the following command.

```
help [subject]
```

If the subject is omitted, a list of debugger commands for which help is available will be printed. This overview can also be obtained by typing **intro** after a help prompt. Otherwise, information on the specified subject is shown.

If the pager prompt

```
--More--
```

appears in the lower left corner of the screen, the valid responses are the following. Type

```
<RETURN>
```

to display another *line* of information. Type

```
<SPACE>
```

to display another *screen* of information. Type

```
q
```

to exit help.

Additionally, to substitute for the default 'paging program', the PAGER environment variable may be set to the name of an alternative paging program. (The default for System V UNIX is **pg**.)

When using the debugger in screen mode, the command

```
:help [subject]
```

will provide the same aid.

An additional help tool is available in screen mode by typing

```
H
```

which prints a line of information at the bottom of the screen such as the one below.

```
b|B(break) d(delete break) a(advance) s(step) g(go) r(run)          --H
```

This is the first line of several single lines of help text that provide a brief summary of the screen-mode commands. Press **H** again for the next line of help text.

See also *DEBUGGER REFERENCE*, **help**, page 20–41.

8.7 COMMAND SYNTAX

Most debugger commands are of the form

```
keyword [parameters]
```

In line mode a list of commands can be placed on a single line if separated by semicolons as illustrated below.

```
bd; g
```

In screen mode, this command line must begin with a colon.

Commands are executed when <RETURN> is pressed. An exception is a breakpoint command followed by a block of commands where the breakpoint is set after the final <RETURN>.

The debugger uses Ada syntax for comments, that is, characters between the double hyphen (--) and <RETURN> are ignored.

While in line mode, <RETURN> can be used to repeat the most recent of several commands (**a**, **ai**, **s**, **si**, **l**, **li**, /, or **?**). Debugging a program with a **r** (run) or **g** (go) command clears <RETURN> until one of the repeatable commands is used again. Each command that can be repeated this way is marked with the phrase '<RETURN> repeats' in the documentation. In screen mode, type a period to repeat the previous command line.

The debugger expects data and instruction addresses to be hexadecimal numbers. When a hexadecimal number could be interpreted as a decimal number or as an identifier, use a leading zero.

Register values can be printed or modified. To specify a register, precede the register name with a dollar sign, i.e., **$eax**, **$ebp**.

See also *[VADS DEBUGGER], 8.11.6 Display Register Contents:* **reg**, page 8–16.

File names used with the debugger commands **e**, **vi**, **edit**, **r**, and **set** may be enclosed in double quotes (**"/usr2/jack/src/foo.c"**). Quotes are required around simple file names that contain characters having some special meaning in the shell. The debugger interprets shell environment (exported) variables for file names.

When a name could refer either to a debugger keyword or a variable name, the debugger interprets it as a keyword. To force interpretation as a variable, precede the name with a backslash.

Control characters have their usual meaning. In addition, the UNIX interrupt signal <INTR> will interrupt the program being debugged if it is running or it will interrupt the debugger's current operation. The debugger immediately responds to interruption with a prompt for the next command.

For more information about control characters, **see also** the Operating system documentation, **stty**(1) and **tty**(4).

8.8 DEBUGGER CONCEPTS

Using a symbolic debugger effectively depends on understanding several language and implementation concepts, including the notion of a debugger call stack, name overload resolution, and visibility rules for language objects.

8.8.1 Call Stack, Home and Current Positions

To represent the current state of the program being debugged, **a.db** uses a model known as the *call stack*. The call stack represents all currently active subprograms in the program being debugged. These are subprograms that have been called, but have not yet returned to their caller. When the program is executing, the subprogram that is currently executing is at the top of the call stack. A subprogram call 'pushes' the called subprogram on top of the stack. When a subprogram returns to its caller, the returning subprogram is 'popped' from the top of the stack.

When the program being debugged is halted at a breakpoint or after a single step, the subprogram containing the point of execution where the program is stopped is the top of the call stack. The debugger provides a command **cd** (call down) that lets the user 'move' down the call stack, that is, from the current subprogram to the subprogram that called the current one. There are also commands for moving up (**cu**), to the top (**ct**), to the bottom (**cb**), and displaying the call stack (**cs**). As will be explained, changing the current level on the call stack changes the variables that are directly visible.

When the program stops, the debugger initializes two 'positions' — the 'home' position and the 'current' position. The home position represents where the program will execute next (or where the program is executing) at the current level of the call stack. The current position represents that part of the source code seen on the terminal at this moment. The current position will change as debugger commands display different source files or disassemble part of the program. The home position only changes when moving up and down on the call stack. Whenever the home position changes, the current position changes to match it.

The purpose of the home position is to show exactly where execution is at the current level of the call stack. When displaying a line of source code or machine instructions containing the home position, the debugger shows a * in the left margin next to the line. In the top subprogram of the call stack, the home position is the next instruction to be executed. In all other subprograms on the call stack, the home position will be the source line (or machine instruction) that called the next higher subprogram.

The purpose of the current position concept is to make commands that operate on line numbers more convenient to use. The current position is marked with >. Certain listing commands (**l**, **li**, **w**, and **wi**) use the current position as their default starting point. In addition, the **l** and **li** commands will change the current position to be the last item listed. In this way, successive **l** or **li** commands provide a way of conveniently displaying successive sections of text or disassembled instructions. Another command using a line number is the **b**, set breakpoint, command. If no explicit line number is given, the current position will be used.

8.8.2 Overload Resolution (Disambiguation)

a.db supports a subprogram, task, or package name as the object of the **e** (enter subprogram or file), **vi** (enter screen-oriented mode), and **b** (set breakpoint) commands. When debugging Ada programs, the name supplied to these commands need not be fully qualified (the simple name **sin** could be used in place of **standard.math.sin**).

When a simple name is used for one of these commands, the debugger searches 'program-wide' to find the name. This search allows the user to set a breakpoint or enter any subprogram regardless of the current visibility rules.

If the search finds multiple definitions for the same name, the debugger issues a message displaying each of the visible alternatives. The notation is *simple_name'n*, where *n* is a number (e.g., **sin'2**). An example of the debugger message follows. A breakpoint is being set at a subprogram with the simple name **add** which is overloaded.

```
>b add
=> add is overloaded. Use 'n to select one:
add'1 (integer, integer) return integer
add'2 (integer, integer) return real_a
add'3 (integer, integer) return float
add'4 (integer)
add'5 (float)
>b add'3
```

The suffix *'n* must be appended to the subprogram name to indicate precisely the name to be referenced, where *n* is one of the numbers listed in the overloading message (**b add'3**).

The debugger assigns a number to each occurrence of a simple name, starting with 1. These numbers remain constant throughout the debugging session. When the debugger finds overloading, all of the overloadings will be listed.

8.8.3 Names, Variables, and Visibility Rules

a.db uses Ada scope and visibility rules similar to Ada for looking up variable names. The debugger takes the current home position as the basis for looking up a variable name for the **p** or other commands. If an Ada subprogram, package, or task contains the current home position, the debugger uses Ada visibility rules in looking up the name. The debugger will find the same definition for the name as the Ada compiler would find if the name were part of the program text corresponding to the current home position.

See also Ada RM 8.2, 8.3

In addition to names defined in the program being debugged, the debugger recognizes the names of hardware registers when prefixed by a dollar sign. These names can be used anywhere a program variable name can be used, including as the left hand side of the assignment operator (:=).

8.9 LOADING THE PROGRAM

Self-target debuggers automatically load the desired executable and prepare it for execution.

In cross-target debuggers, a **load** command must be given to instruct the debugger to download the executable. Since the runtime system kernel is an independent executable, it must be downloaded separately to the target using **a.run** before debugging or running a user program. Cross-target debuggers can also be used with an image already in target memory by giving the **sync** command immediately after invocation.

See also *VADS Debugger Reference;* **load**, page 20-47, and **sync**, page 20-76.

8.10 MODIFY DEBUGGER CONFIGURATION: set

set commands are used to modify the current debugger parameters. They can be used at any time during normal operation of **a.db**. The commands can also appear in the debugger runtime configuration, **.dbrc**, file which is read during invocation. This file is normally placed in the user's home directory. In addition to normal file names, **set** interprets environment variables.

The **set** command uses the following format.

```
set [option [value]]
```

Current settings of all the parameters are available in line mode by typing

```
set
set all
```

or in screen mode

```
:set
:set all
```

All **set** commands are valid within the **.dbrc** file, from a command input (**-i**) file, or typed directly from the keyboard.

When debugging in line mode, or in screen mode with **set output tty**, no processing is done on characters written to **stdout** by the program being debugged. When debugging in screen mode with **set output pty**, lines of characters written to **stdout** are processed slightly to avoid having a blank line at the bottom of the screen. The line-feed character at the end of a line or partial line is written. Then the window scrolls up and the line is written, but the line-feed character is again delayed until the next output from the program. For a group of lines of output numbering more than the current size of the lower window, some lines are written and the prompt --More-- appears on the bottom of the window. No further output appears until the user replies to the prompt.

This behavior may be undesirable when debugging programs that use cursor-positioning or other special character sequences to affect output to the video terminal. For debugging programs of this type, toggling between screen mode and line mode or between **set output pty** and **set output tty** is often helpful.

8.11 Display Commands

The following debugger commands are available to display variables, source code, tasks, and memory. Defaults are shown in brackets.

8.11.1 Display Lines: l, w

l (list) and **w** (window) are used to view source code and use similar formats.

```
l [line] [, number]
w [line] [, number]
```

l lists *number* of lines of the current source file starting at the specified *line* or current line and can be repeated by pressing <RETURN>. For example, the command

```
l 5, 15
```

lists 15 lines of code starting with line 5.

Similarly, **w** lists a window of source text surrounding the current or specified line. For example, the command

```
w 15, 20
```

lists the 20 lines surrounding line 15.

The **w** command sets the <RETURN> key to repeat the **l** command: typing **l** will continue listing where the last **l** or **w** command stopped.

The default for number of lines to be listed is 10, but can be changed through the **set lines** command. The default specification for *line* is the current line, which is marked with < character to the left. The possible forms of *line* are listed below.

number	move to the specified line
+number	move *number* lines forward
–number	move *number* lines backward
*	move to the home position

If * appears after the line number, the line is the home line; = after the line number indicates that a breakpoint is set there. When – appears after the line number, a breakpoint is set within the code generated for this line, but not on the first instruction. If both = and – apply to the same line, the last one set is displayed.

8.11.2 Display Breakpoints: lb

The command **lb** lists all the currently set breakpoints. A number in brackets is displayed to the left of each breakpoint and is used with **d** to delete individual breakpoints. Sample output for **lb** is shown below.

```
   20    task p2 is
   21       pragma priority (7);
   22    end p2;
   23    task body p2 is
   24=   i:integer;
   25    begin
   26       for i in 1..50 loop
   27          put ("Task p2 prints this");
   28          new_line;
   29       end loop;
   30    end p2;
   31
   32    begin
   33       put ("The main subprogram prints this");
   34=      new_line;
*-----------------------------------------------taskpr1.a--
:lb
[1]  "/vc/davidh/ada_ex/taskpr1.a":14 in p1
[2]  "/vc/davidh/ada_ex/taskpr1.a":24 in p1
[3]  "/vc/davidh/ada_ex/taskpr1.a":34 in taskpr1
```

8.11.3 Display Instructions: li, wi

li and **wi** display disassembled machine instructions and use the formats shown below where *number* is a set of lines of the current source file starting at the specified *line* or current line.

```
li [line|instruction] [, number]
wi [line|instruction] [, number]
```

li (list instructions) is used to list a specified number of lines of disassembled instructions (source code with corresponding assembly language code) and can be repeated by pressing <RETURN>. Similarly, **wi** (window instructions) prints a window of disassembled code surrounding a specified *line* or instruction address (hexadecimal number).

Sample output is shown below.

```
  21        pragma priority (7);
  22    end p2;
  23    task body p2 is
  24=   i:integer;
  25    begin
  26*       for i in 1..50 loop
  27            put ("Task pw prints this");
  28            new_line;
  29        end loop;
  30    end p2;
*-----------------------------------------------taskpr1.a--
:li
  26        for i in 1..50 loop
   3be4:  movl    sp,r0
   3be7:  movl    =1,-1c(fp)
   3beb:  movl    =32,r6
  27            put ("Task p2 prints this");
   3bee:  pushab  0x9e90
   3bf4:  pushab  0x9ea4
   3bfa:  calls   =2,_A_put.118S12.text_io
  28            new_line;
   3c01:  pushl   =1
```

The listing starts with the specified position (*line or instruction)* or with the home position in a new session. If the position is not specified, the listing will begin with the line or instruction following the position most recently listed by **l**, **li**. If no **l** or **li** command has been given since the last execution command or if the *line|instruction* parameter is *, listing will begin with the home position.

The display contains program machine instructions interspersed among the source lines. The first instruction is preceded by the source line that generated it except when no source is available or disassembly begins mid statement.

In addition to the **li** and **wi** commands for displaying instructions, the debugger can be put into *instruction* mode. In *instruction* mode, the source window will contain disassembled machine instructions, interspersed with source code, if available. Although the source window contains machine instructions, it can still be controlled as usual. In this mode, the **s** and **a** debugger commands will be interpreted as their machine instruction counterparts, the **si** and **ai** commands, respectively. The **b** command will be interpreted as **bi**, setting a breakpoint at the machine instruction under the cursor. All searching and window commands are available.

a.das may be used for disassembling object files outside the debugger.

See also *Analysis Tools. 7.2* **a.das**, page 7-1.

8.11.4 Display Tasks: lt

Several commands are used specifically in debugging tasks. The first is **lt** (list task), which lists the state of all tasks that have been created and have not been reclaimed after termination. The command

```
lt
```

produces output such as that shown in the abbreviated example below.

```
Q#      TASK             ADDR      STATUS

        <idle task>      07a908    ready
        rand_delay       077bd8    suspended at accept for
                                     entry rand
        T philosopher    072178    suspended calling
                                     fork[063f88].put_down
R1      T philosopher    06c718    ready
D1      T philosopher    0699e8    suspended at delay
                                     on delay queue, until
                                     367600 millisecs
*       fork             063f88    executing
                                     in rendezvous with T
                                     philosopher[072178] at
                                     entry put_down
        dining_room      058ac8    suspended at select on
                                     delay queue, until 401230
                                     millisecs open entries:
                                     allocate_seat
        T output         055d98    suspended at select
                                     (termination possible)
                                     open entries:  put_cursor

        <main program>   045068    ready
```

The simple **lt** command displays four columns of information for each task. The columns are labeled Q#, TASK, ADDR, and STATUS.

The Q# column (queue numbers) may have one of several values: R*n* indicates that the task is on the *run queue* in the *n*th position. R1 will run next. D*n* means that the task is on the *delay queue* in the *n*th position. The delay for D1 will expire next. * indicates the current task (the breakpointed task or the task most recently selected).

The TASK column contains either the name of the task or a T followed by the name of the task type that declared the task. Note that the parent task has no name and is listed as **<main program>**. Also, the runtime itself defines a task used when all other tasks are suspended and the scheduler is waiting for an interrupt event. This task is listed as **<idle task>** when present.

The ADDR column contains the address the run-time system assigns to the task when it is created. This address is used when setting breakpoints for a particular task with the **select** command. The address is also used with the **lt** command when the task is specified. The address value must be used when identifying instances of a task type, as the task type name is not a unique identifier.

The STATUS column shows the state of each task and additional information for some states. There are 11 possible states for each task.

STATUS	DESCRIPTION
`not yet active`	the task has been created, but not yet activated **See also** Ada RM 9.3
`awaiting activations`	the task is activating subtasks
`executing`	the task is executing now
`waiting`	The task may have been accepted for immediate rendezvous, may be in transition, or may be suspended on a semaphore queue
`ready`	the task is ready (normally, it will be on the run queue, but might be suspended on a semaphore, for example)
`suspended at accept`	the task has executed an **accept** on an entry that no task is currently calling, so it is now waiting until some task calls it
	`for entry` *entry_name* — produces the name of the accepting entry
`suspended at select`	the task has executed a **select** but no tasks are calling open entries; it is now waiting until some event allows it to proceed
	`(termination possible)` — means that the select statement has an open terminate alternative
	`OPEN ENTRIES:` *entry_name* — lists each open entry of the select
	`NO ENTRIES OPEN:` — means that no entries are currently open
`suspended calling`	the task has executed an entry call and will remain in this state until rendezvous is completed for the call
	task_name [*task_addr*]*.entry_name* — gives the target task and entry

`suspended at delay`	the task has executed a delay statement
`awaiting interrupts`	the task is a special task that is executing an idle loop until interrupted. Normally, this is IDLE_TASK
`completed`	the task has executed all its code body. Upon completion of all subtasks, it will terminate **See also** Ada RM 9.4(5)
`terminated`	the task is now terminated. **See also** Ada RM *9.4(6)*

Some additional information may be appended to the status entry.

`in rendezvous with` *task_name*`[`*task_addr*`] at entry` *entry_name*

means that the task has accepted the entry call from the named task at the named entry. This message may be given more than once. It is repeated for each rendezvous resulting from outer nested **accepts** in order from innermost to outermost **accept**.

`on delay queue, until` *number* `millisecs`

means that after the runtime's *current_time* reaches the given number of milliseconds, this task will become ready for execution. The task may be on the delay queue because of a simple delay statement, a timed entry call, or because of an open delay alternative of an active **select**.

`about to raise` *exception_string*

means that an exception occurred in this task and the named exception will be raised in the master.

When tasks are dynamically created (i.e., anonymous tasks), only their addresses are listed. The task number can be used with the **select** and **cs** (call stack) commands.

The **lt** *task* form of this command provides additional information about a single task.

The command

```
lt dining_room
```

produces the following output.

```
Q#      TASK            ADDR        STATUS
*       dining_room     058ac8      executing in rendezvous
                                     with T philosopher[06c718]
                                     at entry allocate_seat

        ENTRY           STATUS      TASKS WAITING
        allocate_seat   open        T philosopher[072178]
```

```
                                    T philosopher[06c718]
                                    T philosopher[0699e8]
enter           open          - no tasks waiting -
leave           open          - no tasks waiting -

static priority  = 3
current priority = 4
parent task:    <main program>[045068]
                at line 18 in /vc/sfz/t/ada.tests/phl.a
```

The first lines are the same as the brief display. The added information includes a table of entry queue status giving the entry name, whether the entry is open for a select, and the ordered lists of the tasks waiting at that entry.

The **static priority** line gives the task priority. Following that is **current priority**. A task executes at the higher of its own static priority and the current priority of any task with which it is in rendezvous.

The **parent task** line describes the master of this task. The master block for this task is at line 18 in the file **phl.a**, in the main program. The task that is executing this master is 045068. With this information it is possible to construct a tree of tasks, linked by their masters to their parent tasks. Note that the task executing the master is not necessarily the task that creates it.

See also Ada RM 9.4

8.11.5 Display Variables: p

The command

```
p expression
```

displays Ada variables. The debugger supports simple variable names (MARK, R_A), and, for Ada, selected components and expanded names (MARK.LINE) and indexed components (X(Y), Z(1,2)). These can also be used in combination (X(M.Z), B.X(1)). Array slices are not yet supported.

Ada subprograms and functions can also be called (entry calls are not yet supported), and the results of a function can be further used in an expression. Only parameters of mode **in** are supported: **in out** or **out** parameters are not. Default parameter values are not yet supported. The debugger will not call a function that returns an array or a record; however, functions returning access values are supported. A parameterless function or subprogram must be called using empty parentheses,

```
p display_board( )
```

8.11.6 Display Register Contents: reg

The simple command

```
reg
```

lists the contents of all registers as they were when the program stopped. Sample output is shown below.

```
   1  -- taskpr1.a
   2
   3  with TEXT_IO; use TEXT_IO;
   4
   5
*-----------------------------------------------taskpr1.a--
EAX:   402eb4   EBX:     d764   ECX:   4029a0   EDX:       1e
EBP: 7ffffdc0   ESI:       14   EDI:   402eb8   ESP: 7ffffdb8
CS:        17    SS:       1f   EIP:     d77c
EFLAGS:  297:  c:1 p:1 a:1 z:0 s:1 t:0 i:1 d:0 o:0 iopl: 0 n:0 r:0 vm:0
```

The command

```
reg f
```

lists the floating point coprocessor registers.

8.11.7 Display Call Stack of Task: cs

The **cs** command displays the call stack, starting with the current frame and has the following format.

```
cs [reg] [number] [of task_number]
```

number is used to specify the topmost number of frames to be displayed. The **of** *task* clause is used to display the call stack for a specific task with the *task_number* obtained using the **lt** command.

If the keyword **reg** is used, an extra three-line hexadecimal dump is provided per frame. The first line shows the values of the PC, FP, and AP registers. The second line displays a row of 32-bit words, starting with the word pointed to by FP and moving down to lower addressed words. The third line displays a row of 32-bit words, starting with the word pointed to by the AP and moving up to higher addressed words.

8.11.8 Display Task Number: task

The command

```
task
```

displays the address of the current task.

8.11.9 Value Assignment

The := operator is used in the following formats to assign a value to a variable, register, or memory location(s). No type checking is performed between the name and the expression.

name := *expression*
address [, *number*] := *expression*

name can be any Ada expression that identifies a scalar or a register name (preceded by a dollar sign). *expression* can be any scalar expression. *address* can be either a hexadecimal number (with a leading 0) or of the form *@number* where *number* (of *@number)* is hexadecimal or decimal. *number* (of the above value assignment example) is an optional parameter indicating how many bytes are to be modified. If *expression* evaluates to an integer, *number* must be 1, 2, 3, or 4. If expression evaluates to a floating point number, *number* must be either 4 or 8. If absent, *number* defaults to 4.

After the memory is modified, a line of the form

address: *new/old*

is printed. *address* is the altered memory location, *new* is the current value. *old* is its previous value. For Ada names, *address*: is not printed. **See also** Ada RM 4.1

8.11.10 Display Raw Memory

Command formats for displaying raw memory are shown below.

@number [:*display*] [*number*]
hexadecimal_address [:*display*] [*number*]
decimal_address :*display* [*number*]
name :*display* [*number*]

With a leading @, *number* can be either decimal or hexadecimal. Following @, hexadecimal numbers are distinguished from decimal numbers by use of a leading 0. A decimal number *may not* begin with 0.

display consists of a format and/or a length character from the following list. The default is shown in brackets.

The syntax for the *display* parameter is

[*length*] [*format*]

LENGTH

B 8-bit [default number base established by **set obase**]
D 64-bit floating point
F 32-bit floating point
L 32-bit [default number base established by **set obase**]
W 16-bit [default number base established by **set obase**]

FORMAT

- **a** show the address of the item
- **c** ASCII character
- **d** decimal [32 bits]
- **f** floating point [32 bits]
- **m** one line of STORAGE_UNITS, first in hexadecimal, then as ASCII characters
- **n** like **m** but bytes are interpreted in reverse order
- **o** octal [32 bits]
- **p** hexadecimal pointer [32 bits]
- **r** reverse-map the address to a procedure name
- **s** null-terminated (C-style) string
- **x** hexadecimal [32 bits]

number determines how many values are to be displayed. The address is advanced by a number corresponding to the letter being used. For example, to display 8 16-bit decimal values starting at address 01ac4, type the following command.

```
01ac4:Wd 8
```

The default length is 32 bits. The default format is either decimal, octal, or hexadecimal, depending on the setting of the **set obase** command. The default count is 1. Specifying an address of the form @ *number* or *hexadecimal_number* will cause 32 bits to be displayed in the current output base.

```
01ac4:m
```

See also *DEBUGGER REFERENCE,* **set**, page 20-70, (**set obase**).

The colon followed by a format specification is required to dump memory. Specifying *number* alone is the debugger command to change the current position.

8.12 EXECUTION COMMANDS

The debugger execution commands control the flow of execution of the program, including normal running, single stepping, advancing over called subprograms, and executing until a specified variable changes value.

8.12.1 Step One Line/Instruction: a, ai, s, si

The single stepping commands **a** (advance), **ai** (advance instruction), **s** (step), and **si** (step instruction) execute the program for a single line or instruction. **s** (step) and **si** (step instruction) step one line or instruction, stepping *into* called subprograms. **a** (advance) and **ai** (advance instruction) are used to step one source line or instruction, stepping *over* called subprograms. <RETURN> repeats all single stepping commands.

8.12.2 Continue Execution: g, gw, gx

The **g** (go), **gw** (go while), and **gx** (go signal) commands execute the program from where it last breakpointed until a specific condition is reached.

If the process is breakpointed because of a UNIX signal, **g** will continue the process, ignoring the signal. **gx** works in the same manner as **g**, but the program is continued with the signal that caused it to breakpoint.

gw causes the program to execute until the value of the specified variable changes, and uses the following format.

```
gw name|address [, number]
```

If an *address* (*@number*) is specified, *number* bytes at that address are watched for change (default = 4). *number* can be 1–16. (The value being watched is checked after the execution of each machine instruction, which is quite slow on some hardware. Running **a.db** from a script may be helpful in this case.)

8.12.3 Run Program: r

r (run) starts or restarts the program from the beginning and uses the following format.

```
r [shell_arguments]
```

If *shell_arguments* is specified, **r** runs the program as if it were being executed from the shell. The debugger supports a subset of shell command arguments. It supports the following arguments for I/O redirection: >, <, >>, and >& (for redirecting both standard and error output). The debugger supports substitution for shell environment (exported) variables. Additionally, when argument strings contain dollar signs ($), backquotes (‘), or globbing meta symbols (*, ?, []), they are passed to the shell for evaluation. The debugger uses **sh**, or the shell defined in the environment (exported) variable $SHELL. Strings enclosed in double quotation marks are passed as a single argument after removing the quotes. Other parameters are passed (**-o**, **-Pfoo**) just like the shell.

8.12.4 Read Debugger Commands From a File: read

The command

```
read file_name
```

causes the debugger to switch from executing keyboard instructions to executing instructions from the specified file. Additional **read** commands can occur in the file, but may be nested only to a maximum of four levels. After executing the file, commands are again read from the keyboard (unless the command file contained an **exit** or **quit** command).

8.12.5 Terminate the Debugger Session: exit, quit

Either **exit** or **quit** terminates the debugger session.

8.13 BREAKPOINT COMMANDS

The program being debugged can be stopped at a particular place in its execution by setting a breakpoint.

With **r** or **g**, the program will execute until it encounters the breakpoint. If **r** or **g** are used to run a program and there is the possibility of an infinite loop, the program can be stopped by pressing the interrupt key, <INTR>. The interrupt key can also be used to stop the debugger from completing a long display, for example if the display of a large array of large records is accidentally initiated.

The current frame is the position on the call stack. When a breakpoint is announced, the current frame is always set to the subprogram containing the breakpoint. This frame is also the topmost frame of the call stack and is known as the break frame. As one moves up and down the stack, the current frame is always the topmost frame of a **cs** listing. If **li** is used to produce a listing and disassembly, the next instruction in the current frame to be executed will have a * next to it (the home position). The current frame can only be changed using the call stack commands (**cu**, **cd**, **ct**, **cb**) when the program is stopped at a breakpoint.

8.13.1 Set Breakpoint: b

The basic forms of the command to set a breakpoint are shown below.

```
b [line|subprogram] [of task] [begin commands end]
```

This command sets a breakpoint at a specified line in the current file or at the beginning of a specified subprogram. *line* is typically given as a decimal number; however, all the options specified in the **line number** section of this guide are supported. To set a breakpoint at line 537 of the current source file, use the following command.

```
b 537
```

subprogram is the name of an Ada subprogram, task body, or package. Expanded names as well as simple identifiers can also be used (starting with STANDARD if desired). To set a breakpoint at the subprogram SORT_STAMPS, the following command is used.

```
b SORT_STAMPS
```

See also *Debugger Reference, visibility rules*, page 20–80, and for a discussion of expanded names, *Ada RM 4.1.3(13)*.

8.13.2 Command Blocks

Debugger commands can be entered on the same line as the breakpoint if they are separated by semicolons as illustrated below. The syntax is similar to that used in Ada.

```
b [line|subprogram] [of task] [begin commands end]
```

For example.

```
b 38 begin p i; p dump_node(root) end
```

Alternatively, as long as the beginning block mark **begin** appears on the breakpoint line, the commands can appear each on a new line. The debugger will prompt with a ?? prompt until the closing block mark **end** is entered, as illustrated below.

```
>b 38 begin
??p i
??p dump_node(root)
??end
```

8.13.3 Set Conditional Breakpoint: b

When a conditional breakpoint is reached, execution continues transparently if the condition is false. If the condition is true, the specified commands are performed.

Possible formats for the **b** command are shown below.

```
b [line|subprogram] [of task] when expression
b [line|subprogram] [of task] if expression then commands [else
   commands] end [if]
b [line|subprogram] [of task] if expression else commands end [if]
```

where the following terms are used.

line|subprogram	line number or subprogram name
task	task identifier **See also** Debugger Reference, **lt**, page 20–48.
expression	the condition at the breakpoint (a non–zero value is defined as TRUE)
commands	list of debugger commands to be executed

The **then** block will be executed when the breakpoint is reached and a specified condition is true. If there is no condition (no **if** clause), the **then** block will be executed every time the breakpoint is reached. The **else** block is only used in conjunction with an **if** clause.

Specify a conditional breakpoint by using **when** *expression* after the breakpoint command as shown in the example below.

```
b 38 when col <= row
```

This will cause the program to break at line 38 if col <= row. Alternatively, **if** *expression* **then** may be used.

```
b 38 if col < row then p col; p row; end if
```

This sets a breakpoint at line 38; however, the debugger only stops the process when it reaches the breakpoint *and* col < row. The expression is evaluated and (as with C expressions) 0 means FALSE, other values mean TRUE. All comparison operators (<, <=, =, etc.) return 0 for false and 1 for true.

The debugger verifies that the conditional expression is valid when the breakpoint is set. The expression is verified using the visibility rules, etc., that will exist when the breakpoint is hit and the expression is actually evaluated.

Debugger commands to be executed when the breakpoint is hit but the condition is false can also be set. These commands take the form of an **else** clause as shown below.

```
b 38 if i > 5 else p i; end if
```

In this example, the command **p i** between **else** and **end if** will be executed when the breakpoint at line 38 is reached, but i > 5 is not true. This provides a tracing facility.

8.13.4 Break Down: bd, bD

The **bd** command sets a breakpoint in the subprogram that called the current subprogram. The breakpoint is reached as soon as the current subprogram returns. The current subprogram is represented by the current frame. The following **bd** format (similar to those available for **b**) can be used.

Note that the case of the command is important.

```
bd [of task] [begin commands end]

bd [of task] when expression

bd [of task] if expression then commands [else commands] end [if]

bd [of task] if expression else commands end [if]
```

If the command is given in lower case (**bd**), the breakpoint will be removed automatically when it is hit. If it is given with the D in upper case (**bD**), the breakpoint will remain until explicitly removed.

bd breakpoints are shown with a hyphen next to the breakpoint number in a **lb** listing and at the source line when the breakpoint is reached.

8.13.5 Set Breakpoint at Instruction: bi

The **bi** command sets a breakpoint at an instruction address.

```
bi [instruction] [of task] [begin commands end]
bi [instruction] [of task] when expression
bi [instruction] [of task] if expression then commands
    [else commands] end [if]
bi [instruction] [of task] if expression else commands end [if]
```

The instruction address (which can be obtained using **li** or **wi**) should be given as a hexadecimal number with leading zero.

8.13.6 Set Breakpoint at Exception: bx

bx sets a breakpoint that is reached when the named *exception* occurs.

```
bx [exception] [of task] [begin commands end]
bx [instruction] [of task] when expression
bx [instruction] [of task] if expression then commands
    [else commands] end [if]
bx [instruction] [of task] if expression else commands end [if]
```

Like **b**, **bd**, **bi**, and **bx** breakpoints can be set for a particular task using the **of** *task* option and can be followed by a block of debugger commands to be executed when the breakpoint is reached.

8.13.7 Display Instruction: li, wi

li or **wi** can be used to produce a display of the instructions (disassembly) to determine where to set an instruction breakpoint.

See also *[VADS DEBUGGER], 8.11 Display Commands*, page 8–10.

8.13.8 Delete Breakpoint: d

The **d** command deletes the listed breakpoints with the following format.

```
d all|breakpoint_number [, breakpoint_number]...
```

Specify **all** to delete all breakpoints. Multiple breakpoint numbers comprising the *breakpoint_number* (obtained using **lb** and displayed in brackets) must be separated by commas in this command as illustrated below.

```
d 1, 2, 3
```

8.13.9 Implicit Breakpoints

The debugger sets some breakpoints automatically for its own use.

1. When the debugger is initially invoked and also after a 'set run' command, the *current position* and the *home position* are moved to point to the first executable statement of the *main* subprogram. If the user types either an **s** or **a** command, then the debugger will automatically set a breakpoint at the beginning of the *main* subprogram and run the program. When this breakpoint is reached, it is automatically removed and the program is then stepped according to the semantics of the **s** or **a** command.

2. When an **a** or **ai** command is typed, and there is a subprogram call to step over, the debugger automatically sets a breakpoint on the instruction following the call instruction and then lets the program call the procedure. This allows the program to proceed at normal speed until the breakpoint is reached. When the breakpoint is hit, after the subprogram returns, it is automatically removed.

3. A breakpoint is automatically set at a procedure in the Ada runtime system called 'slight_pause'. This null procedure is called just prior to the run–time system abandoning execution when an unhandled exception is discovered, as described in Ada RM 11.4.2.4.

8.13.10 Tasking Breakpoints

The debugger supports line number breakpoints set for a specific task using the **of** clause. When the program reaches a breakpoint set with this clause, the debugger checks to see if the task executing is the one specified by the breakpoint. If not, the task is continued. If the task executing is the one specified by the breakpoint, the breakpoint is announced to the user.

8.14 POSITIONING COMMANDS

a.db includes commands to move the current position for debugging purposes.

8.14.1 Specify New Position

Specifying a decimal *line_number* will move the current position to that line number in the current file. In screen mode, the line number must be followed with an upper case **G**. Specifying a plus or minus before a number allows relative positioning. Specifying * will move the current position to the current home position.

8.14.2 Call Stack Bottom, Top: cb, ct

cb (call bottom) and **ct** (call top) are similar commands that move the current position and current frame to the bottom or top of the call stack. To display the values of local variables and parameters of a subprogram currently active on the call stack, first move the current position to that frame on the stack. This makes the local variables visible.

8.14.3 Call Stack Up, Down: cd, cu

cd (call down) and **cu** (call up) move up and down the call stack and can take *name* or *number* as arguments.

```
cd [name|number]
cu [name|number]
```

If *name* is provided, the current frame moves down to the next frame on the call stack with that name. If *number* is specified, movement (in either direction) is to the frame having that number. The new frame becomes the the current frame, and the line being executed in that frame becomes the current home position, marked with a *.

8.14.4 Call Stack: cs

The **cs** command displays the contents of the stack, starting with the current frame and displays the frame numbers. The **cs** command has the following format.

```
cs [reg] [number] [of task]
```

If *number* is specified, only the topmost *number* frames are displayed. If the keyword **reg** is used, an extra 3 line hexadecimal dump is provided per frame. This is mainly used for debugging the debugger and compiler, but may be occasionally useful in other circumstances. If the **of** *task* clause is used, where *task* is a task name or address obtained through the **lt** *task* command, the stack for that task will be displayed. The default stack is that of the current (breakpointed) task.

8.14.5 Move Current Position: e, select

e (enter) and **select** provide ways to move the current position to a new file or subprogram or to change the current task using the following format.

```
e [subprogram|ada_source.a]
select new_task
```

The subprogram name need not be directly visible. For purposes of the **b**, **e**, **edit**, and **vi** commands, all subprogram names in the program are searched. Expanded names can also be given.

See also Ada RM 4.1.3(13).

If a subprogram name is overloaded, a diagnostic showing the alternatives is given. The appropriate suffix (**'1**, **'2**) must then be used to disambiguate the reference.

In screen mode, the **e** command can be invoked by positioning the cursor on the leftmost character of a subprogram name and then typing <CONTROL>]. This command is intended to be compatible with the **vi** editor's <CONTROL>] tag command.

select identifies a new task to become the current task. *task* is the task name or address identified by using the **lt** command. **select** makes the call stack of the selected task visible. The call stack commands (**cb**, **cd**, **cs**, **ct**, and **cu**) operate on the current task's call stack.

8.14.6 Enter Editor: edit

Use **edit** to enter **vi** or the editor specified in the environment (exported) variable $EDITOR or the specified file or subprogram. If no parameter is given, the file containing the current position is used. The default editor is **vi**.

Parameters are interpreted exactly the same as for the **e** command.

8.15 INVOKING THE SCREEN-MODE DEBUGGER

To use the screen interface, either invoke the debugger with the **-v** option,

```
a.db -v myprogram
```

or type the debugger command

```
vi
```

in response to a debugger prompt at any time during the line-mode debugging session.

To return to the line-oriented interface, type the screen command **Q**.

8.15.1 Screen-mode Windows

In screen mode, the debugger divides the screen into three windows. The top window, called the 'source window', extends from the top of the screen to a dashed line and is used for program source text. The window below the dashed line is called the 'command window' and is used to display debugger and program input and output. The last line on the screen is the 'error window' and is used for error messages from the screen interface, search patterns, etc.

The command and source windows each display a small portion of a (potentially) much larger area. For example, the source window can be moved to display a different part of the same source file or it can display part of a different source file.

The debugger keeps the last 750 lines of the text displayed in the command window in a history buffer. The command window is typically positioned to display the most recent interactions (i.e., the bottom of the history buffer). However, this window behaves just like the source window in that it can be moved to display a different part of the history buffer.

The command window is automatically paged. If output from either a single debugger command or the user's program would scroll off the top of the window, without requesting user input, this output is stopped and

```
--More--
```

is displayed on the last line. At this point the following responses are valid.

```
<SPACE>
```

to print the next window of text

```
q
```

to not display the remaining output, up to the next debugger request for input, but to enter it into the history buffer and to log it, if logging is turned on.

```
g
```

to enlarge the command window to keep the current text in the command window at the expense of the source window. The window sizes are restored at the next debugger command or by the window control command **C** without a specified number.

```
<RETURN>
```

to print one more line, scrolling one line off the top of the command window.

When in screen mode, the screen interface commands, known as 'window commands' may be used (they do not require the use of <RETURN> to complete them). In addition, any line-oriented debugger command can be used by typing a colon first.

8.15.2 Instruction and Source Modes

When in visual mode, the debugger supports two sub-modes, *instruction* or *source*. The **I** command will toggle the sub-mode, switching from one to the other. In source mode, the source window will display program source code and the **s** and **a** commands will single step at the source statement level. Source mode is the default.

In instruction mode, the source window will contain disassembled machine instructions, interspersed with source code, if available. Although the source window contains machine instructions, it can still be controlled as usual. In this mode, the **s** and **a** debugger commands will be interpreted as their machine instruction counterparts, the **si** and **ai** commands, respectively. The **b** command will be interpreted as **bi**, setting a breakpoint at the machine instruction under the cursor. All searching and window commands are available.

8.15.3 Screen-mode Window Commands

Screen-mode interface commands, known as window commands fall into two groups: those that are related to debugger operations and those that move the cursor around the screen (as with **vi**).

8.15.4 Window Command Syntax

Most window commands are a single character and are *not* followed by <RETURN>. The pattern-matching commands (/ and ?) are followed by the pattern to be found and are terminated by either <ESCAPE> or <RETURN>.

Some commands can be preceded by an optional number. When *number* is given for a <CONTROL>**d** or <CONTROL>**u** command (scroll up or down), that number will be used with all subsequent <CONTROL>**d** and <CONTROL>**u** commands until a new number is specified. The initial value of *number* is one-half the size of the window.

The window commands are applied to the window that contains the cursor. The comma command will move the cursor from one window to the other.

In addition to this set of debugger commands specially tailored for use in the screen mode, any debugger command can be used by typing a colon first. In response to the colon, the screen interface scrolls up the command window one line and positions the cursor on the bottom line of the command window. Then type in the debugger command and press <RETURN>. The command will execute and the cursor will be moved back to where it was when the colon was entered. After each debugger command, the cursor returns to the source window.

A special help facility is provided in screen mode and is used by typing

```
H
```

which will give one of several lines of help at the bottom of the screen. Typing **H** again will display the next line of help.

8.16 CROSS DEBUGGING [cross compilers only]

Cross debugging is very similar to host debugging. All the commands supported in the self-target debuggers are supported for the cross-target debuggers. However, there are some additional considerations for executing the cross debuggers described.

When a cross debugger is invoked, it automatically creates a subprocess (DB_IFACE) on the host that interfaces directly with the embedded system. The debugger communicates with this subprocess using UNIX sockets. DB_IFACE passes commands for debugger actions to the target controlling agent (an emulator or TDM) and receives I/O requests from the target.

See also *Debugger Reference, DB_IFACE*, page 20–27.

The cross debugger does not automatically load the executable program into the memory of the target when the debugger is invoked. The user must use the **load** command to do this. Since the runtime kernel for cross compilers is a separate executable, it must already be present in the target board. To debug the program *my_prog* when the RTS kernel needs to be downloaded as well, the following sequence would be used. The first two commands are typed at the UNIX shell level, the subsequent command at the debugger prompt.

```
a.run -l  kernel_executable --load RTS kernel
a.db my_prog     --invoke debugger
load          --load user program
```

A program already present on the target board may be debugged without reloading by using the **sync** command.This causes the debugger to 'reconnect' to the current memory image in the target system.

8.17 DB_IFACE [cross compilers only]

To enhance portability, the debugger employs a subprocess to communicate with the target processor board.

The name of the interface process is determined from the value of the DB_IFACE INFO directive. This value must be the full pathname of the desired interface process. If it is not, a further library search determines the values of the INFO directives VADS and TARGET_IFACE, and constructs the DB_IFACE process name in the form:

```
VADS_location/bin/db_iface_TARGET_IFACE
```

In the absence of a TARGET_IFACE directive, the value TDM is assumed.

For the following visible directives

```
VADS:info:/usr2/vads5_i386:
VERSION:info:5.7:
TARGET:info:I386:
TARGET_IFACE:info:tdm:
```

the interface process executable name would be

```
/usr2/vads5_i386/bin/db_iface_tdm
```

The DB_IFACE process is also responsible for downloading the memory image to the target.

8.17.1 Process Startup

The debugger prints out the name of the process and its invocation before executing it.

```
db_iface process: /usr2/vads5_i386/bin/db_iface_tdm
```

If **input** or **output** is set to **pty**, the interface process is invoked using the pseudo terminal drivers. After invocation, the interface process prints

```
This is db_iface!
```

to signal that the process has been correctly started. Information is then exchanged between the process and the target board. Correct communication and initialization is signaled by the message

```
initialized ok
```

and the debugger prompt appears.

8.18 DEBUGGING C PROGRAMS

In addition to debugging Ada programs, **a.db** can debug C programs in most VADS implementations. The module must be compiled with the **-g** option in System V UNIX. This can be useful when interfacing an Ada program to C modules or vice versa.

Most of the debugging facilities available for Ada are also available for C, but the coverage is not as thorough. For example, **p foo** displays a C variable named **foo**.

C functions can be called from the debugger. For example, if **foo** is a C function,

```
p foo(3, baz)
```

will call **foo** with parameters **3** and **baz**, where **baz** is a visible variable.

8.18.1 Names, Variables, and Visibility Rules

Looking up variables for C programs begins as for Ada. If the current home position is contained in a C function, the debugger will find the same definition of the name that the C compiler would have found if the name was part of the text of the program corresponding to the current home position. However, if the debugger does not find a definition using conventional C visibility rules, it will search program-wide to see if there are external variables having the same name. If so, the value of that variable will be displayed. As a result, when the current home position is contained in a C function, all external C variables are visible unless hidden by an inner declaration.

8.18.2 Type Checking

Like the C compiler, the debugger does no type checking of the actual parameters before making the function call.

Breakpoints can be set at a C function with

b *file_name*

a C function with the following.

e *file_name*

8.18.3 a.db Stack Commands

The stack commands (**cs**, **cd**, **cu**, **ct**, and **cb**) work with C functions on the stack as well as Ada subprograms.

8.18.4 a.db Screen Functions

All the screen functions of **a.db** work for debugging C programs as well as Ada programs. In particular, the screen commands **p**, **b**, **B**, **d**, and] are very convenient for debugging C code.

8.18.5 C Operations

The following operators are available for C programs:

. structure field selection
-> pointer dereference/structure field selection
[ ] subscript

It should be noted that variable assignment, conditional breakpoints, and relational expressions currently use Ada rather than C syntax even when debugging C programs.

8.18.6 Global C Variables

Global C variables that do not have symbolic information are visible. A message will be displayed and the address of the variable given as a 32-bit integer.

8.18.7 char * Variable Display

When displaying a variable declared as **char ***, both the address (i.e., the value of the pointer) and the string at that address will be displayed. This is useful when using **char *** as a generic pointer and not simply as a pointer to a string.

8.18.8 Warning

If the same structure field name is used in multiple structures in the same source file (and its **#includes**), the debugger cannot disambiguate between them. Note that an entire structure cannot be displayed — only a single field at a time.

8.18.9 Operators

The Ada syntax and operators for assignment (:=), equality (=), inequality (/=), and so forth, rather than the C equivalents, are used in debugger commands.

CHAPTER 9

SOURCE CODE FORMATTER

9.1 OVERVIEW

VADS includes an Ada source code formatter or 'pretty printer' utility, **a.pr**, which writes to the standard output and can easily be piped or redirected to a file.

A range of options are available that make **a.pr** easy to tailor for individual Ada coding standards.

Two tools, **a.error** and **a.list** provide program listings without formatting. **a.error** lists programs containing errors and intersperses the error messages produced by the compiler among the source lines. **a.list** produces a program listing with or without line numbers — but with no other additional formatting — for programs containing no errors.

See also *VADS COMMAND REFERENCE*, **a.error**, page 19–10; and **a.list**, page 19–18.

9.2 INVOKING a.pr

The 'pretty printer' is invoked by a command of the following form.

```
a.pr [options] [ ada_source.a]
```

a.pr reads as input the specified file or standard input. Options can be specified on the command line or in a runtime configuration file, **.prrc**, located in the user's current or home directory. An option specified on the command line takes precedence over options specified in the **.prrc** file. Command line options for the defaults are included to allow options specified in the **.prrc** file to be overridden.

Options for both are listed later in this chapter.

9.3 DEFAULT FORMAT SPECIFICATIONS

If no options are given, **a.pr** will print the source file as described below.

- Reserved words will be in lower case.
- Identifiers will be in upper case.
- Comments will remain as given.

- The outer-most level will begin in the first column.
- Each subsequent level will be indented 8 columns.
- Indentation will be performed with tabs rather than spaces.
- A formatted line of source, including comment and indentation, will not extend past the 132nd column without being continued on the next line.
- A maximum of 55 lines will be allowed on a page; a new page will be started if a program unit (package, subprogram, task, etc.) will not fit on the current page and it will be split among the necessary number of pages.
- Pagination will be performed with form feeds rather than blank lines.
- Record type declarations and type representations will have the word **record** on the same line as the word **type** or the word **for**.
- When aligned to the right, comments will be aligned 4 spaces to the right of the longest source line containing a comment.

9.4 Fixed Format Specifications

In addition to the format specifications listed above, which can be changed by command line or **.prrc** options, an Ada source file will be formatted in the following manner.

- Each Ada statement will be printed on its own line.
- Any blank lines in the input file will produce blank lines in the output except those falling at the beginning of a page other than the first.
- Subprogram or **task** body, **package** specification or body, or **accept** statement will be ended with its identifier name.
- **pragma** PAGE will be recognized and the code following started on a new page.

9.5 Error and Warning Messages

If errors are encountered in the .prrc file, an error message is printed and **a.pr** quits; the source file is not formatted.

All error and warning messages are printed to standard error. To prevent warnings from appearing in the formatted source, use the **-nw** command line option or the **set no_warning** command in the **.prrc** file.

To place the warning/error messages in a file separate from the formatted source, use the following from **sh(1)**.

```
a.pr [options] ada_source.a 1> output_file 2> error_file
```

9.6 .prrc FILE OPTIONS

The **.prrc** file can contain any of the printing options listed on the following page and must be located in either the user's current working directory or the home directory.

Each of the **set** commands to be placed in the **.prrc** file have the following syntax.

set *option* [*argument*]

Each option specification must be on its own line, with any number of blanks between the different parts and any number of blank lines between specifications.

In the options listed below, *number* indicates an integer argument to be provided by the user. If *number* is not specified or it is not within the valid range, **a.pr** exits.

case means the character case of the type of word encountered and must be one of the following.

upper	upper case
lower	lower case
same	same case as in the source file

where indicates the location of comments or **record**.

9.7 COMMAND-LINE OPTIONS

An option specified on the command line takes precedence over options specified in the **.prrc** file.

In the options listed below, *number* indicates an integer argument to be provided by the user. If *number* is not specified or it is not within the valid range, the default value is assumed and the appropriate error message is printed.

9.8 a.pr OUTPUT

Options for major formatting issues are discussed in the following sections.

9.8.1 Comments

Only comments within a single program unit (package, subprogram, or task) are aligned with each other. If, for example, a package body contains a subprogram and a task, the comments in the subprogram will be aligned, the comments in the task will be aligned, and the comments outside of these two will be aligned.

Comments are printed either to the right or to the left as described below. **set align_cmts** (or **-ac** and **-al**) can be used to specify whether a comment, when aligned at the right, is to be printed to the right of the longest line containing a comment (**-ac**, the default) or to the right of the longest line, regardless of whether or not it contains a comment (**-al**). If printed at the right, comments will be aligned four spaces to the right of either the longest line or the longest line containing a comment, depending on the option specified.

To be printed at the left, comments must either be preceded by a blank line or be no farther than one level of indentation to the right of the immediately previous statement. If the line immediately previous to this comment is also a comment on its own line, this comment is aligned with the previous one.

9.8.2 Pagination

Pagination can be accomplished in a variety of ways. **pragma** PAGE is recognized, which starts the code following the pragma on a new page. This cannot be turned off.

If no paging options are specified on the command line or in the **.prrc** file, pagination will be performed using form feeds rather than blank lines.

If the **-p** *number* option or the **.prrc set page** *number* command is given, pagination will be performed with blank lines. The number specified in the option gives the desired length of a page. For example, if

```
set page 65
```

is specified and a block of code ends on the 54th line, 11 blank lines will be printed. (Of course, the page size specified must be greater than or equal to the maximum number of lines desired on a page (set by **-l** *number* or **set lines** *number*).

If the command-line option **-pl** or the **.prrc** command **set page_lu** is used, each library unit (indicated by a **with** clause) will begin on a new page.

If a block of code will not fit on the current page, a new one is started. If the block of code will not fit on a single page, that block will be split between the necessary number of pages.

If the **-np** option or the **.prrc** option **set no_page** is used, pagination will only occur when a **pragma** PAGE is encountered.

9.8.3 Splitting Lines

A line will be split after one of the following.

abs	+
and	-
and then	&
is	/=
mod	=
not	<
or	<=
or else	>
renames	>=
rem	:
xor	:=
**	|
*	,
/	

No attempt is made to force a line to conform to the maximum desired length if it contains none of the above. If, after splitting a line as much as possible, its length is greater than desired, a warning message is issued.

When a line is split, the portion to be printed on the next line is indented one-half level from the original portion to show that it is a split line. For example, if each level is to be indented 4 spaces and the original line is 2 levels deep (indented 8 spaces), the split portion will be indented 10 (8+2) spaces. If each level is to be indented a single space, split portions will be aligned with the original portion.

Subprogram, **accept**, and **entry** headers or parameter lists are special cases. Parameters lists are split in so far as possible by aligning parameter declarations.

If warning messages are to be printed, the offending line's input line number is part of the message. This number is not always the line's true line number. For example, if the input file contains the following,

```
procedure TEST(ARGUMENT1 : integer;
               ARGUMENT2 : boolean;
               ARGUMENT3 : integer);
```

all three lines will be concatenated and then processed. If the combined line length is greater than desired, the line will be split, looking the same as above. If the length of the split lines is still greater than desired, a warning will be printed, but the line number given for ARGUMENT2... and ARGUMENT3... will be the input line number of **procedure** TEST.

9.8.4 Line Length Specification

The maximum line length includes the comment if there is one. If the comments are to be aligned to the right of the longest line (regardless of whether or not it contains a comment), a line of code can be the specified length minus 14. For example, if the specified line length is 80, the maximum length a line of code can be is 66. (80 minus 4 spaces for indentation of comment and 10 spaces for the comment itself regardless of how long the comment really is.)

The same situation occurs if comments are specified to be aligned to the right of the longest line containing comments and the line in question has a comment. If a line does not contain a comment, its maximum line length is that specified by the user.

9.8.5 Tabs

If the **-t** *number* option or the **.prrc** command **set tabs** *number* is used, tabs will be printed whenever the indentation needed for a given line is greater than or equal to the specified number. For example, if **set tabs 4** and **set indent 4** are both specified, each new level is indented with an additional tab. If **set indent 5** is given, each new level is indented with a tab and a blank.

By default, a tab will be printed for every 8 characters needed in indentation (as if **set tabs 8** were specified). The **-t 0** option or the **.prrc** command **set tabs 0** must be specified to indent with spaces rather than tabs.

9.9 EXAMPLES

The examples below show typical use of **a.pr** with options supplied in a **.prrc** file and from the command line.

9.9.1 Unformatted Source File

The following unformatted file is used for illustration.

```
--file: show_pr.a
Procedure show_pr Is
Type rec
    Is Record
i : integer;      -- "char" decl is the longest line
char : character:='x'; end Record;    r   :   rec   := (0, 'z');
    -- Longest line with comment
        int:integer:=5;-- Declarations and comments

   -- should each be aligned
Begin
  -- Because these comments are indented no more than one
 -- level of indentation from the previous statement, they
-- should be aligned at the left.    r.int :=
i;
        end;
```

9.9.2 The .prrc file

A .prrc file is created with options for indentation, margins, the placement of record block indicators and other formatting choices.

```
set indent 4
set margin 4
set tabs   0
set record next
set reserved upper
set ident lower
set align_cmts line
```

9.9.3 Output

Executing the command

```
a.pr show_pr.a
```

reformats the file according to the specifications of the .prrc file and places the output on stdout.

```
--file: show_pr.a
PROCEDURE show_pr IS
    TYPE rec IS
      RECORD
        i    : integer;            -- "char" decl is the longest line
        char : character := 'x';
    end record;

    r   : rec     := (0, 'z');     -- Longest line with comment
    int : integer := 5;            -- Declarations and comments
                                   -- should each be aligned
BEGIN
-- Because these comments are indented no more than one
-- level of indentation from the previous statement, they
-- should be aligned at the left.

    r.int := i;
END show_pr;
```

9.9.4 Command Line Options

The command line options override those given the .prrc file (if present). The command

```
a.pr -i 4 -m 0 -RS -rl -il -ac show_pr.a
```

in conjunction with the .prrc file above produces the following reformatted listing.

```
--file: show_pr.a
procedure show_pr is
    type rec is record
        i    : integer;           -- "char" decl is the longest line
        char : character := 'x';
    end record;

    r   : rec     := (0, 'z');    -- Longest line with comment
    int : integer := 5;           -- Declarations and comments
                                  -- should each be aligned
begin
-- Because these comments are indented no more than one
-- level of indentation from the previous statement, they
-- should be aligned at the left.

    r.int := i;
end show_pr;
```

CHAPTER 10

ADDITIONAL TOOLS

10.1 OVERVIEW

VADS provides two tools outside the normal range of compilation and debugging utilities.

a.help	interactive help utility for VADS
a.list	produce a program listing
a.report	report VADS deficiencies

10.2 a.help

Online help is available for each of the VADS tools using **a.help**. If the local system administrator has elected to install them, UNIX manual pages are available that contain the same information by using the system command **man**(1).

VADS_location/**bin/a.help** is a wrapper program that executes the correct executable based upon directives visible in the **ada.lib** file. This permits multiple VADS compilers to exist on the same host. The **-sh** option prints the name of the actual executable file.

The command is invoked as shown below.

```
a.help [options] [subject]
```

Without the **-p** (pager) option, **a.help** will use the paging program defined by the environment variable PAGER, requiring the full pathname including surrounding quotes for additional options. If PAGER is not defined, **more** is used.

10.3 a.list

a.list provides a convenient way of producing a listing for programs containing no errors that closely resembles the output of **a.error**. The listing is written to the standard output and may be piped or redirected to a file.

10.4 a.report

a.report is used to format reports about problems with VADS software or its documentation. Configuration options are provided that allow reports to be sent directly to Verdix.

a.report automatically captures the date, time, site name, user, and VADS configuration information. It then prompts for the name of source files illustrating the problem to be appended to the report and invokes **vi** or the user's default editor to allow editing of general text that describes the problem. The formatting lines supplied by **a.report** separate parts of the text for APSO's automated processing. Report generation can be aborted at any point using the <INTR> key (usually mapped to <CONTROL>**c**).

For a complete description of **a.report** and submitting problem reports, **see also** *VADS Reference*, **a.report**, page 19–29.

CHAPTER 11

GETTING STARTED WITH VADS

11.1 INTRODUCTION

To use VADS, add the path of the VADS tools to the UNIX $PATH variable. The complete range of VADS commands is then available. The path is usually set in the user's **.profile** so that commands will be available on subsequent logins.

11.2 GETTING STARTED

Begin writing and compiling Ada programs with VADS by creating a VADS library directory. This is simply a directory in which to compile programs. The library directory has several sub–directories of information created and maintained by the compiler. Below is a short example of creating the library directory, copying one of the sample programs supplied with VADS to it, and compiling and executing the program. The same basic process is used to compile any program. (Since VADS may be installed anywhere in the file system and several versions may exist on the same machine, VADS_location below refers to the root directory where the VADS version being used resides.)

```
a.mklib -i my_own_library

 1 version of VADS are available on this machine:

 1  SELF_TARGET  5.7       /urs2/vads5.7

 Selection (q to quit): 1

cd my_own_library

cp VADS_location/examples/hello.a .

ada -M hello.a

a.out
```

With the last command, **a.out**, the following result is displayed.

```
Hello, world.
```

If a collection of Ada source code files already exists in a directory, the files can be compiled in correct dependency order with a simple command sequence as illustrated below. Remember that VADS requires that source file names end in **.a**.

```
cd my_ada_directory
a.mklib -i

 1 version of VADS are available on this machine:

 1  SELF_TARGET  5.7       /urs2/vads5.7


 Selection (q to quit): 1

a.make -v -f *.a
```

After all units present in the library have been compiled, any desired executable can be created with **a.ld**.

Information on the details of using the VADS compiler, debugger, and other tools and their options are found throughout the rest of this manual.

CHAPTER 12

MACHINE CODE INSERTIONS

12.1 OVERVIEW

Verdix provides the **package** MACHINE_CODE described in Section 13.8 of the Ada Language Reference Manual (Ada RM 13.8). The Ada RM specifies the format of a machine code insertion procedure, but leaves the actual implementation to the compiler vendor.

Machine code insertions provide, from within the Ada language, low-level access to the processor that is normally available only from an assembly language.

Use of machine code insertions is necessary in programs that need to reference the hardware directly. They are also useful for optimizing time-critical sections beyond the capabilities of the compiler.

VADS machine code insertions provide features such as the X'REF attribute, a full range of addressing modes, and parameters that enable the programmer to integrate the machine code into surrounding code with minimal effort.

Machine code insertions are, however, a non-portable, processor-dependent feature of the language. The compiler cannot perform certain types of error checks that are performed for normal procedures (such as the enforcement of strong-typing). Machine code insertions should be used with discretion and only where absolutely necessary.

In VADS Documentation, we follow the notation conventions of the architecture reference manual: the destination register is on the left and the source is on the right.

12.2 MACHINE CODE PROCEDURES

A machine code procedure may be declared any place a procedure may be declared. The body of a machine code procedure must be within the context of a **with** of the **package** MACHINE_CODE (provided in the **standard** library). The language imposes the following restrictions.

- Machine code insertions are only allowed within a procedure body.
- The only declarations that may occur in the declarative section of a machine code procedure are **use** clauses and pragmas. Local variables and types cannot be declared.

- The only statements that may occur in the body of a machine code procedure are code-statements (described below).
- A machine code procedure cannot have an exception handler.

12.3 CODE-STATEMENTS

Conceptually, code-statements can be thought of as calls that take an OPCODE or SIZE as their first parameter, and *x* OPERANDs as their remaining parameters. Note that, because a code-statement is in fact a record aggregate, named notation must be used for CODE_0, e.g., CODE_0'(OP => NOP). (**See also** Ada RM 4.3(4).)

A code-statement is technically an aggregate data value of a data type declared in the **package** MACHINE_CODE. The VADS **package** MACHINE_CODE provides the variant record types CODE_0, CODE_1, CODE_2, CODE_3, CODE_4, DATA_1, DATA_N, CALL_0, CALL_1, and CALL_N. The CODE_*x* types have a variant of **type** OPCODE and *x* components of **type** OPERAND. The DATA_*x* types have a variant of **type** SIZE. DATA_1 has one OPERAND component, while DATA_N has a component that is a variable number of OPERANDs. The CALL_*x* types take a subprogram reference of the type OPERAND and *x* components of type OPERAND. The types OPCODE, OPERAND, and SIZE are also declared in the **package** MACHINE_CODE.

12.3.1 Opcodes

The **type** OPCODE is an enumeration type declared in the **package** MACHINE_CODE. The type provides all of the instructions for the 80386 and 80287/80387. Instructions are named by their standard mnemonics except for the instructions 'and', 'in', 'loop', 'not', 'or', 'out', and 'xor', which are appended by '_op', since each root is identical to an Ada reserved word.

12.3.2 Operands

The **type** OPERAND is a private type declared in the **package** MACHINE_CODE. The package also provides constants of **type** OPERAND and functions that return values of **type** OPERAND. The VADS-defined attribute X'REF can be applied to most Ada objects and denotes a value of **type** OPERAND.

All the 80386 and 80287/80387 registers are provided as OPERAND constants, including the segment registers, control registers, debug registers, and test registers.

All the 80386 addressing modes are made available by functions that denote values of the type OPERAND. For example, the following code-statement stores the value in EBX at the address pointed to by EAX.

```
CODE_2'(MOV, BASE(EAX), EBX);
```

The following table summarizes the available addressing modes and the functions used to implement them.

Form	Function
Displacement	DISP(disp)
Base	BASE(breg)
Base + Displacement	BASE(breg, disp)
(Index*Scale) + Displacement	INDEX(ireg, scale, disp)
Base + Index + Displacement	BASE_INDEX(breg, ireg, disp)
Base + (Index*Scale) + Displacement	BASE_INDEX(breg, ireg, scale, disp)
Segment	SEGMENT(sreg, mem)
Immediate	IMMED(value) +value -value
External Symbol	EXT(name, disp)

There are a set of type operators which specify the access size of a memory address. The following table summarizes the available operators and their access size.

Function	Access Size
BYTE(mem)	8-bits
WORD(mem)	16-bits
DWORD(mem)	32-bits
PWORD(mem)	48-bits
QWORD(mem)	64-bits
TBYTE(mem)	80-bits
UNTYPED(mem)	N/A

The **type operator** UNTYPED can only be used if the context already indicates the access size. This is useful for those instructions whose access size do not fall into one

of the sized categories. These instructions are: FLDENV, FRSTOR, FSAVE, and FNSAVE.

The following forms show the syntax of forming an addressing mode:

```
addr_func(...)
SEGMENT(sreg, addr_func(...))
size_func(addr_func(...))
size_func(SEGMENT(sreg, addr_func(...)))
```

Examples:

```
CODE_2'(MOV, EAX, ESP);
CODE_2'(MOV, EAX, BASE(ESP));
CODE_2'(LEA, EAX, INDEX(EAX, 4, 8));
CODE_2'(MOV, EBX, BASE_INDEX(EBX, EDX, 4, 0));
CODE_2'(MOV, EAX, SEGMENT(CS, BASE(ESP)));
CODE_1'(INC, BYTE(SEGMENT(CS, BASE(ESP))));
CODE_1'(INC, DWORD(INDEX(EAX, 4, 8)));
CODE_2'(LEA, EDX, EXT("FOO"));
CODE_1'(FSAVE, UNTYPED(BASE(EBP)));
```

The operands are used to determine which instruction is to be selected. For example,

```
CODE_2'(MOV, EAX, +1);
```

provides enough information to determine that the instruction is a *move immediate* dword *to a register*, because the register EAX indicates that the operation type is dword. The following example,

```
CODE_2'(MOV, BASE(EAX), +1);
```

is now ambiguous, because an anonymous access type is specified and the immediate value can be of any size. The address mode must specify the access size. The following example chooses the instruction *move immediate byte indirect through a register*.

```
CODE_2'(MOV, BYTE(BASE(EAX)), +1);
```

The following example,

```
CODE_2'(MOV, BASE(EAX), AX);
```

does not require that an access type be specified, since the register AX, determines that it is a 'word' (i.e., 16-bit) operation.

All arguments to code functions must be one of the following:

- static expression (see. Ada RM 4.9)
- type conversion (the expression operand must be a static expression)

- string literal
- representation attributes (see. Ada RM 13.7.2)
- X'REF attribute
- entity defined in the package MACHINE_CODE

The BASE, INDEX, and BASE_INDEX functions are overloaded to allow the displacement to be specified as a numeric (integer) or as a label reference (operand). Specifying a negative numeric reference, such as –4, can be interpreted as an integer or an operand, due to the overloading of "-". Use qualification or named notation within the function to resolve the overloading of BASE, INDEX, or BASE_INDEX. For example,

```
CODE_2'(MOV,  EAX, BASE(EAX, INTEGER'(-4)));
CODE_2'(MOV,  EAX, BASE(EAX, DISP_INT => -4));
```

12.3.3 Ada Entities as Operands

It may sometimes be necessary or desirable to reference Ada constants and variables from within a machine code insertion procedure. It would be very tedious and error-prone for a programmer to attempt to calculate these references by hand. VADS provides the attribute X'REF, which generates a reference to the entity X. (In some cases, using X'REF will cause more than a single instruction to be generated.)

Examples:

Given:

```
STRING_POINTER is access STRING(1 .. 3);
X : STRING_POINTER := new STRING'("ABC");
```

Then:

```
CODE_2'(MOV, AL, X.all(2)'REF)
```

Generates:

```
mov         eax, @X     --X to eax
test        eax, eax    --make sure X is not nul
jne         L
call        "raise_constraint_error"
L:  mov     al, 01[eax] --load character into al register
```

When X is static, the value used is an immediate literal.

Given:

```
X : constant integer := 5;
```

Then:

```
CODE_2'(MOV, EAX, X'REF);
```

Generates:

```
mov         eax, 05
```

The X'REF attribute can also be used to generate references to a procedure's parameters

Given:

```
procedure        ROTATE_LEFT(
                 ELEMENT: in INTEGER;
                 COUNT  : in TINY_INTEGER;
                 RESULT : out INTEGER) is
 begin
                 CODE_2'(MOV, EAX, ELEMENT'REF);
                 CODE_2'(MOV, CL, COUNT'REF);
                 CODE_2'(ROL, EAX, CL);
                 CODE_2'(MOV, RESULT'REF, EAX);
 end             ROTATE_LEFT;
```

Generates:

```
(preamble and postamble code generation omitted):
                 mov  eax, 0c[ebp]
                 mov  cl, 08[ebp]
                 rol  eax, cl
                 mov  010[ebp], eax
```

When the X'REF attribute is applied to labels, the address generated is an offset or a branch displacement depending on the instruction. For example, the following instructions generate an offset for LABEL'REF. Note that the first instruction is used to obtain the address of LABEL. The second instruction moves the double word at location LABEL.

```
<<LABEL>>
   ...

   CODE_2'(LEA, EAX, LABEL'REF);      -- address of LABEL
   CODE_2'(MOV, EAX, LABEL'REF);      -- value at LABEL
```

The following instruction uses a branch displacement.

```
CODE_1'(JMP, LABEL'REF);              -- jump to LABEL
```

When the X'REF attribute is applied to subprograms, the address generated is an offset.

12.4 PROGRAM CONTROL

OPERANDs are used to control the flow of execution with instructions such as CALL, Jcc, and LOOP/LOOPcond. Labels and subroutine names can be used in conjunction with the X'REF attribute to form destinations for these instructions.

The following example illustrates a typical startup routine for an Ada program. Its function is to call an initialization routine, elaborate the library units, call the main program, and call an exit routine.

```
with MACHINE_CODE;
procedure STARTUP is
```

```
   use MACHINE_CODE;
   pragma IMPLICIT_CODE(OFF);

begin
   -- User/Board/Default initializations.
   CODE_1'(CALL,   EXT("CONFIG_INIT"));

   -- Store the return pc.  Used by exception handling routines
   -- to prevent unwinding past this frame.
   CODE_2'(LEA, EAX, STOP_LAB'REF);
   CODE_2'(MOV, EXT("STOP"), EAX);

   -- The elaboration table is built by a.ld, it is built as an
   -- array containing the addresses of the library units to be
   -- elaborated, followed by the address of the main program,
   -- and ending with a null address.
   CODE_2'(XOR_OP, EDX, EDX);          -- EDX := 0
<<ELAB>>
   CODE_2'(LEA,    EBX, EXT("ELABORATION_TABLE"));
   CODE_2'(MOV,    EBX, BASE_INDEX(EBX, EDX, 4, 0));
   CODE_2'(TEST,   EBX, EBX);
   CODE_1'(JZ,  DONE'REF);
   CODE_1'(PUSH,   EDX);               -- save register
   CODE_1'(CALL,   EBX);
<<STOP_LAB>>
   CODE_1'(POP,    EDX);               -- restore register
   CODE_1'(INC,    EDX);
   CODE_1'(JMP,    ELAB'REF);

<<DONE>>
   CODE_1'(PUSH,   EAX);
   CODE_1'(CALL,   EXT("HALT"));

end STARTUP;
```

12.5 SUBPROGRAM CALL

A general form of a subroutine call can be made from within a machine code procedure by using a CALL statement. Three formats are provided, CALL_0, CALL_1, and CALL_N to handle the various argument lists needed by a subroutine. The first argument to the call statement is the name of the subroutine suffixed by 'REF to yield a value of type OPERAND. All subsequent arguments, if any, must also be of type OPERAND. The number of arguments and their type compatibility are checked. The three forms are:

```
CALL_0'(SUBP => FOO'REF);

CALL_1'(FOO'REF, ARG'REF);

CALL_N'(FOO'REF, (ARG1'REF, ..., ARGn'REF));
```

If *foo* is a procedure, then the following Ada procedure calls would be equivalent to the statements shown above.

```
FOO;

FOO(ARG);

FOO(ARG1, ..., ARGn);
```

If *foo* is function, the forms would be similar, but the result is not assigned.

If *foo* is overloaded, an error will be generated when using 'REF. The subroutine should be renamed to provide a distinct name.

The current limitation on the arguments, is that they must be of the form NAME'REF. No expression is supported, although the expression can be assigned to the name before the call.

The effect is to push the arguments onto the stack (if any), make the call, copy the out parameters (if any), and adjust the sp (if needed). The return value of a function will not be copied.

If **pragma** INLINE is indicated for *foo*, then *foo* would be expanded inline.

12.6 LOCAL DATA

As shown above, variables that are visible to a machine code procedure (either in packages or enclosing subprograms) can be referenced using the X'REF attribute. In some cases, however, it may be necessary to intermix data and generated code. The DATA_1 code-statement is used to place a single data item in the code, while the DATA_N code-statement is used for multiple data items.

An operand is restricted to the following:

- immediate
- external symbol
- label reference
- subprogram reference

All but the first (immediate) require the data size to be 32-bits (DWORD).

Examples:

```
DATA_1'(BYTE,  IMMED(ASCII.LF));
DATA_1'(DWORD, EXT("FOO"));
DATA_N'(DWORD, (LABEL1'REF, LABEL2'REF, LABEL3'REF));
```

12.6.1 Jump Table via Offsets

A jump table can be constructed by building a table of offsets. The table is built by using the data statement, where the operands consist of label references to the selected entry points. The following program fragment illustrates the technique.

```
procedure EXAMPLE (INDEX: INTEGER) is
begin

   CODE_2'(MOV, EAX, INDEX'REF);
   CODE_1'(JMP, INDEX(EAX, 4, TABLE'REF));

<<TABLE>>
   DATA_1'(DWORD, L0'REF);
   DATA_1'(DWORD, L1'REF);
   DATA_1'(DWORD, L2'REF);
   ...
   DATA_1'(DWORD, Ln'REF);

<<L0>>
   ...
<<L1>>
   ...
<<L2>>
   ...
<<Ln>>
   ...

end EXAMPLE;
```

12.7 PREFIX OVERRIDES

A prefix override can be inserted into the code stream by using a data statement with the appropriate value, preceding the code statement.
For example,

```
DATA_1'(BYTE, +16#67#);                -- Address-size override
DATA_1'(BYTE, +16#66#);                -- Operand-size override
CODE_2'(MOV,  EAX, BASE(EBX));
```

12.8 FLOATING POINT

The VADS compiler automatically generates code for the 80287/80387 coprocessor. The **package** MACHINE_CODE also provides the opcodes and registers for the 80287 subset of the 80387 so that it can be referenced directly from machine code insertions as well.

For example, the 80287 provides instructions that perform trigonometric functions. The Ada function below returns the partial tangent of a value given in degrees. It calls a machine code procedure that uses the 80287 to first convert the degrees to radians and then to take the partial tangent using the FPTAN instruction.

```
function PTAN (THETA: DEGREES) return FLOAT is

   DEGREES_180: constant FLOAT := 180.0;
   RESULT     : FLOAT;
```

```
procedure FPTAN(THETA: FLOAT; RESULT : out FLOAT) is
    use MACHINE_CODE;
begin
   CODE_1'(FLD, DEGREES_180'REF);
   CODE_0'(OP => FLDPI);
   CODE_0'(OP => FDIVR);              -- degrees to radians ratio
(180/pi)
   CODE_1'(FLD, THETA'REF);
   CODE_0'(OP => FDIV);               -- THETA in radians
{THETA/(180/pi)}
   CODE_0'(OP => FPTAN);              -- tan(THETA) = X/Y
   CODE_0'(OP => FDIVR);
   CODE_1'(FST, RESULT'REF);
end FPTAN;
  pragma inline(FPTAN);

begin
   FPTAN(X, RESULT);
   return RESULT;
end PTAN;
```

Note that the VADS Ada type FLOAT corresponds to a Long Real on the 80287/80387, while the type SHORT_FLOAT corresponds to a Short Real. No VADS Ada type corresponds to Temporary Real.

12.9 PRAGMAS

There are four pragmas that directly affect the generation of code for machine code procedures: INLINE, SUPPRESS, IMPLICIT_CODE, and OPTIMIZE_CODE.

12.9.1 pragma INLINE

This pragma works with machine code procedures having its normal effect of causing the routine to be expanded inline where called, rather than generating call/return instructions to a single body of code.

12.9.2 pragma SUPPRESS

This pragma has the effect of suppressing the generation of runtime checks. Runtime checks may be generated to perform an elaboration check at the start of the procedure and to perform constraint checks on objects accessed using X'REF. These checks only ensure that the reference is valid; they do not check whether the value assigned is valid. For example, the code-statement CODE_2'(MOV, X.all'REF, EAX) would perform an access check to ensure that X is not null, but would not check to ensure that the value moved to the referenced location was in the range of the type of X.all.

12.9.3 pragma IMPLICIT_CODE

This pragma controls the generation of implicit code. Implicit code is code generated for procedure entry and exit to support the calling conventions used by the compiler. (This does not include the RTS instruction, which is always generated unless **pragma** INLINE is used.) Implicit code also includes any additional code generated due to the use of the X'REF attribute (such as code to load a base register).

When pragma IMPLICIT_CODE(OFF) is specified, any stack allocation, and the STORAGE_CHECK that is generated for the stack allocation are not generated.

Implicit code is always generated for a X'REF attribute which requires it. A warning message is generated when IMPLICIT_CODE(OFF) is specified in such a case.

12.9.4 pragma OPTIMIZE_CODE

This pragma allows the programmer to specify whether the compiler should attempt to optimize through the machine code insertions. When pragma OPTIMIZE_CODE(OFF) is specified, the compiler will generate the code as specified.

The pragma should be placed in the declarative section of the machine code procedure.

12.9.5 Debugging Machine Code

The VADS debugger supports source-level debugging of machine code insertions. Breakpoints can be set at code-statements just like any other statements.

Register values can be determined using the **reg** command or by preceding the register name with a dollar sign and using either the **p** command or a longword dump raw memory command. For example, the register EAX can be examined as a longword decimal value using the line-mode command shown below.

```
$eax:Ld
```

The **li** and **wi** instructions can be used to disassemble the generated code.

12.10 package MACHINE_CODE

```
package MACHINE_CODE is

  -- Description for the Intel 80386 and 80287/80387.

type opcode is (aaa, aad, aam, aas, adc, add, and_op, arpl, bound, bsf,
bsr, bt, btc, btr, bts, call, cbw, cwde, clc, cld, cli, clts, cmc, cmp,
cmps, cmpsb, cmpsw, cmpsd, cwd, cdq, daa, das, dec, div, enter, hlt,
```

```
idiv, imul, in_op, inc, ins, insb, insw, insd, int, into, iret, iretd,
ja, jae, jb, jbe, jc, jcxz, jecxz, je, jg, jge, jl, jle, jna, jnae,
jnb, jnbe, jnc, jne, jng, jnge, jnl, jnle, jno, jnp, jns, jnz, jo, jp,
jpe, jpo, js, jz, jmp, lahf, lar, lea, leave, lgdt, lidt, lds, lss,
les, lfs, lgs, lldt, lmsw, lock, lods, lodsb, lodsw, lodsd, loop_op,
loope, loopne, loopnz, loopz, lsl, ltr, mov, movs, movsb, movsw, movsd,
movsx, movzx, mul, neg, nop, not_op, or_op, out_op, outs, outsb, outsw,
outsd, pop, popa, popad, popf, popfd, push, pusha, pushad, pushf,
pushfd, rcl, rcr, rol, ror, rep, repe, repne, ret, sahf, sal, sar, shl,
shr, sbb, scas, scasb, scasw, scasd, seta, setae, setb, setbe, setc,
sete, setg, setge, setl, setle, setna, setnae, setnb, setnbe, setnc,
setne, setng, setnge, setnl, setnle, setno, setnp, setns, setnz, seto,
setp, setpe, setpo, sets, setz, sgdt, sidt, shld, shrd, sldt, smsw,
stc, std, sti, stos, stosb, stosw, stosd, str, sub, test, verr, verw,
wait, xchg, xlat, xlatb, xor_op, fabs, fadd, faddp, fbld, fbstp, fchs,
fclex, fnclex, fcom, fcomp, fcompp, fcos, fdecstp, fdiv, fdivp, fdivr,
fdivrp, ffree, fiadd, ficom, ficomp, fidiv, fidivr, fild, fimul,
fincstp, finit, fninit, fist, fistp, fisub, fisubr, fld, fldcw, fldenv,
fldlg2, fldln2, fldl2e, fldl2t, fldpi, fldz, fld1, fmul, fmulp, fnop,
fpatan, fprem, fprem1, fptan, frndint, frstor, fsave, fnsave, fscale,
fsetpm, fsin, fsincos, fsqrt, fst, fstcw, fnstcw, fstenv, fnstenv,
fstp, fstsw, fnstsw, fsub, fsubp, fsubr, fsubrp, ftst, fucom, fucomp,
fucompp, fwait, fxam, fxch, fxtract, fyl2x, fyl2xp1, f2xm1);

  type size is (
    byte, word, dword);

  type operand is private;

  type operand_seq is array (positive range <>) of operand;
  n: positive;

  --
  --  Instruction formats.
  --

  type code_0 (op: opcode) is
    record
      null;
    end record;

  type code_1 (op: opcode) is
    record
      oprnd_1: operand;
    end record;

  type code_2 (op: opcode) is
    record
      oprnd_1: operand;
      oprnd_2: operand;
    end record;

  type code_3 (op: opcode) is
    record
```

```
    oprnd_1: operand;
    oprnd_2: operand;
    oprnd_3: operand;
  end record;

subtype prefix is code_0;

--
--  Data formats.
--

type data_1 (sz: size) is
  record
    oprnd_1: operand;
  end record;

type data_n (sz: size) is
  record
    oprnd_n: operand_seq (1..n);
  end record;

--
--  Call formats.
--

type call_0 is
  record
    subp: operand;
  end record;

type call_1 is
  record
    subp:    operand;
    oprnd_1: operand;
  end record;

type call_n is
  record
    subp:    operand;
    oprnd_n: operand_seq (1..n);
  end record;

--
--  Registers.
--

-- 32-bit registers.
eax: constant operand;
ecx: constant operand;
edx: constant operand;
ebx: constant operand;
esp: constant operand;
ebp: constant operand;
esi: constant operand;
edi: constant operand;
```

```
-- 16-bit registers.
ax: constant operand;
cx: constant operand;
dx: constant operand;
bx: constant operand;
sp: constant operand;
bp: constant operand;
si: constant operand;
di: constant operand;

-- 8-bit low registers.
al: constant operand;
cl: constant operand;
dl: constant operand;
bl: constant operand;

-- 8-bit high registers.
ah: constant operand;
ch: constant operand;
dh: constant operand;
bh: constant operand;

-- 287/387 stack elements.
st : constant operand;
st0: constant operand;
st1: constant operand;
st2: constant operand;
st3: constant operand;
st4: constant operand;
st5: constant operand;
st6: constant operand;
st7: constant operand;

-- Segment registers.
es: constant operand;
cs: constant operand;
ss: constant operand;
ds: constant operand;
fs: constant operand;
gs: constant operand;

-- Control registers.
cr0: constant operand;
cr2: constant operand;
cr3: constant operand;

-- Debug registers.
dr0: constant operand;
dr1: constant operand;
dr2: constant operand;
dr3: constant operand;
dr6: constant operand;
dr7: constant operand;
```

```
-- Test registers.
tr6: constant operand;
tr7: constant operand;

--
--  Addressing modes.
--

function disp (
    disp_int: integer)
  return operand;

  -- Description:
  --   The displacement indicates the offset of the operand.

function base (
    base_reg: operand)
  return operand;

  -- Description:
  --   The offset of the operand is specified indirectly in one of
  --   the general registers.

function base (
    base_reg: operand;
    disp_int: integer)
  return operand;

function base (
    base_reg: operand;
    disp_ref: operand)
  return operand;

  -- Description:
  --   The address of the operand is the sum of the contents of
  --   the register plus the displacement.

function index (
    index_reg: operand;
    scale:     integer;
    disp_int:  integer)
  return operand;

function index (
    index_reg: operand;
    scale:     integer;
    disp_ref:  operand)
  return operand;

  -- Description:
  --   The address of the operand is the sum of the displacement plus
  --   the contents of the index register multiplied by the scale
  --   factor.

function base_index (
    base_reg:  operand;
```

```
    index_reg: operand;
    disp_int:  integer)
  return operand;

function base_index (
    base_reg:  operand;
    index_reg: operand;
    disp_ref:  operand)
  return operand;

  -- Description:
  --   The address of the operand is the sum of the displacement plus
  --   the contents of the index register plus the contents of the
  --   base register.

function base_index (
    base_reg:  operand;
    index_reg: operand;
    scale:     integer;
    disp_int:  integer)
  return operand;

function base_index (
    base_reg:  operand;
    index_reg: operand;
    scale:     integer;
    disp_ref:  operand)
  return operand;

  -- Description:
  --   The address of the operand is the sum of the displacement plus
  --   the contents of the index register multiplied by the scale
  --   factor plus the contents of the base register.

function segment (
    seg: operand;
    mem: operand)
  return operand;

  -- Description:
  --   Specifies the segment register to be used with the addressing
  --   mode.

function byte (
    mem: operand)
  return operand;

  -- Description:
  --   Specifies the access size of the memory addressing mode.
  --   BYTE - one byte, 8 bits, an 80386 data type.

function word (
    mem: operand)
  return operand;
```

```
   -- Description:
   --    Specifies the access size of the memory addressing mode.
   --    WORD - one word, 16 bits, an 80386 data type.

function dword (
    mem: operand)
  return operand;

   -- Description:
   --    Specifies the access size of the memory addressing mode.
   --    DWORD - one dword ("doubleword"), 32 bits, a 80386 or
   --    80287/80387 data type.

function pword (
    mem: operand)
  return operand;

   -- Description:
   --    Specifies the access size of the memory addressing mode.
   --    PWORD - three words, 48 bits, an 80386 data type.

function qword (
    mem: operand)
  return operand;

   -- Description:
   --    Specifies the access size of the memory addressing mode.
   --    QWORD - one qword ("quadword"), packed decimal, an 80287/80387
   --    data type.

function tbyte (
    mem: operand)
  return operand;

   -- Description:
   --    Specifies the access size of the memory addressing mode.
   --    TBYTE - one ten-byte, 80 bits, an 80287/80387 data type.

function untyped (
    mem: operand)
  return operand;

   -- Description:
   --    Specifies that the access size be derived from its context.

function immed (
    val: integer)
  return operand;

function "+" (
    right: integer)
  return operand;

function "-" (
    right: integer)
  return operand;
```

```
      -- Description:
      --   An immediate operand, which may be 32-, 16-, or 8-bits long.

    function immed (
        val: character)
      return operand;

      -- Description:
      --   An immediate operand, which is 8-bits long.

    function ext (
        name: string;
        disp: integer := 0)
      return operand;

      -- Description:
      --   The name denotes an external symbol.

  private

    --
    --  Implementation specific.
    --

  end MACHINE_CODE;
```

CHAPTER 13

INTERFACE PROGRAMMING USING VADS

13.1 OVERVIEW

Translating programs from other languages into Ada is usually straightforward if the source language is one of the block-structured languages such as Pascal or C. Clearly-structured programs in other languages are not difficult to translate, but may require more work due to differences between the source and target languages. However, it is often desirable to make use of subprograms or libraries developed in some other language from inside Ada programs without having to translate everything into Ada.

This chapter presents (1) an approach to making existing libraries and programs written in C useful from Ada and (2) a discussion of a modular approach to program conversion into Ada. Although UNIX, C, and self-targeted compilers are referenced, this section has wider applicability because the types of declarations are similar to those used to solve interface problems on any operating system.

13.2 ADA INTERFACE TO curses

This section presents an Ada interface to the UNIX-derived **curses** library of compiled C functions for screen formatting. The Ada package for calling the library functions gives the programmer the same functional entities and objects as the original.

It is assumed that the reader has some familiarity with the UNIX **curses** library of cursor motion optimization routines for video terminals, which provides text input and output and functions for creating windows and altering terminal characteristics. It may be helpful to have a copy of the operating system manual section on the **curses** library available for reference. The complete source code for the **curses** interface package described in this example is in the **publiclib** library supplied with VADS.

The goal in this example is to provide a complete Ada interface to **curses** using the same subprogram and variable names provided in the original C version. Further, a programmer should be able to use the standard documentation for **curses** so that no additional effort is needed in order to make use of the library from Ada.

The following five steps necessary to accomplish this goal are discussed.

1. Create parallel data types.
2. Declare external subprograms.
3. Access global variables declared in the original language.
4. Link with existing libraries.
5. Reduce the call overhead of the interface.

13.2.1 Step 1 — Create Parallel Data Types

Whenever access to a subprogram or variable declared in an alternate language is required, it is the programmer's responsibility to ensure that any Ada variables used in conjunction with the subroutine or variable are of a compatible data representation in both languages. The programmer cannot, for example, assume that a data structure declared in Ada is identical to a data structure declared in C. Although the type names may be identical, the composition, length, or alignment of the object or component may not be. Type definitions in different languages, can, if taken at face value, cause erroneous results. For instance, VADS Ada **type** FLOAT is not the same as the C type **float** for 32-bit CPUs.

Two basic approaches are available for creating parallel data types: using *a priori* knowledge and using Ada representation specifications. There are some types that the programmer knows are parallel between two language implementations from reading the vendor's documentation. Remember, however, that neither Ada nor C compilers are required to use a particular size to represent any particular type, and an implementation is free to choose a representation based on hardware considerations.

Ada representation specifications, on the other hand, allow the Ada programmer to define an exact duplicate of the physical layout of any data type in another language once it is known. Ada allows type specifications that are largely independent of the implementation. Type, storage, record layout, and alignment can all be controlled.

When the underlying representation of a type has no analogue in one language (or for which the usage in one language is significantly different from Ada), the data type can be defined by the programmer using Ada representation specifications and UNCHECKED_CONVERSIONs.

Simple Types — Some samples of the more common C simple types and their corresponding Ada predefined types (on most VADS implementations with 32-bit integers) are given below. For example, the **type** SHORT_INTEGER is equivalent to the pcc-derived C compiler's type **short**, both representing a 16-bit integer.

C	Fortran	Ada
int	INTEGER*4	INTEGER
long	INTEGER*4	INTEGER
short	INTEGER*2	SHORT_INTEGER
char	CHARACTER	CHARACTER TINY_INTEGER
float	REAL	SHORT_FLOAT
double	DOUBLE PRECISION	FLOAT

For the declaration of a C unsigned integer

```
unsigned short u_var;
```

there is no predefined Ada equivalent, and the type would have to be created using representation specifications.

```
-- an Ada version of the C type: unsigned short
type C_UNSIGNED_SHORT is range 0 .. (2 ** 16) - 1;
for C_UNSIGNED_SHORT'LENGTH use 16;

u_var : C_UNSIGNED_SHORT;
```

The first statement creates a type whose range includes all the values the C type encompasses. The second assures that at most 16 bits of storage will be allocated to every object of this type.

The C type **char** is used both to represent a character (usually by its ASCII value) or a byte integer value. Further, some C implementations treat **char** as unsigned and some as signed quantities. There is no exact Ada analogue to this type: VADS Ada's TINY_INTEGER can be used for numeric representations and **type** CHARACTER to represent a character value as illustrated below. When C programs contain ambiguous assignments or uses of such types or of integer/address conversions, the **generic function** UNCHECKED_CONVERSION offers a method for controlled easing of type conversions.

C Source	Ada Source

`char number = 20;`	`with UNCHECKED_CONVERSION;`
	`function CONVERT_NO_TO_CHAR is new`
	`  UNCHECKED_CONVERSION(TINY_INTEGER,`
	`  CHARACTER);`
	`function CONVERT_CHAR_TO_NO is new`
	`  UNCHECKED_CONVERSION(CHARACTER,`
	`  TINY_INTEGER);`

`char ch;`	`NUMBER: TINY_INTEGER := 20;` `CH : CHARACTER;`
`ch = number;` `ch += 2;`	`CH := CONVERT_NO_TO_CHAR(NUMBER);` `CH := (CONVERT_CHAR_TO_NO(CH) +2);`

Record Types — The same two basic approaches can be taken to the representation of record types as with simple types. Again, *a priori* knowledge can be used. Both C

and VADS Ada associate the record label with a base address from which offsets to access individual components of the record are calculated. For the Ada programmer, as long as the record is composed of equivalent simple data types, the offsets will be calculated similarly and the record structures will be identical.

However, occasions arise where storage conventions are not so conveniently arranged. In such cases, Ada representation specification can be given for constructing records. For example, curses uses the following C structure.

```
struct sgttyb{
  char sg_ispeed;
  char sg_ospeed;
  char sg_erase;
  char sg_kill;
  short sg_flags;
};
```

A parallel structure could be assured using the following statements in conjunction with the type definitions for **short** and **char** of the type mentioned above.

```
type C_SHORT is range -(2 ** 15) .. (2 ** 15) - 1;
type C_TINY is range -(2 ** 7) .. (2 ** 7) - 1;
type SGTTYB is
record
  SG_ISPEED : C_TINY;
  SG_OSPEED : C_TINY;
  SG_ERASE  : C_TINY;
  SG_KILL   : C_TINY;
  SG_FLAGS  : C_SHORT;
end record;

for SGTTYB use
record
  SG_ISPEED at 0 range 0..7;
  SG_OSPEED at 1 range 0..7;
  SG_ERASE at 2 range 0..7;
  SG_KILL at 3 range 0..7;
  SG_FLAGS at 4 range 0..15;
end record;
```

In this example, the record is first defined in terms of its simple type components and then in terms of storage allocation characteristics. The value that appears directly after the **at** indicates the number of STORAGE_UNITS (in this case bytes) from the base address the element should be placed. The range specifies the bits the type should occupy.

Ada also supports an optional clause to specify object alignment. For instance, if it became necessary to have objects aligned on two byte boundaries, the statement **at mod 2** could be used as shown below:

```
for SGTTYB use
record at mod 2;
  SG_ISPEED at 0 range 0..7;
  SG_OSPEED at 0 range 8..15;
  SG_ERASE at 2 range 0..7;
  SG_KILL at 2 range 8..15;
  SG_FLAGS at 4 range 0..15;
end record;
```

Array Types — When defining Ada array types that are parallel to C array types, remember that the standard representation of an array in both languages is to associate the array label with the first component and use this location to calculate an offset. Once again this means that as long as the programmer assures that the individual components are compatible structures the basic array structure type should be the same. When in doubt, representation specifications should be used to assure that individual component representations are identical.

Dynamic Array Types — Several languages, including Ada and C, define dynamically-sized arrays. In C, the size of a dynamic array is calculated by the user based on data known only to the user. (An exception is the C string type, treated below.) In Ada, VADS implements dynamic arrays by keeping 'dope vectors' created with each dynamic subtype; the dope vector for a single dimensioned array contains

```
type dope vector is record
   element_size: implementation-defined;
   low_bound:   implementation_defined;
   high_bound:  implementation-defined;
   array_size implementation_defined;
end record;
```

Verdix recommends that the SYSTEM.ADDRESS of the first element of an Ada array be used to pass the array to C. Note, however, that C arrays always start at index 0 while Ada arrays start with any index. *element(1)* might name entirely different true elements.

Fortran multidimensional arrays are stored in column-major storage format. C and Ada arrays are stored in row-major format. Multidimensional indices must be reversed between Ada and Fortran.

Passing arrays from C to Ada is possible by creating an appropriate subtype for the value. If the array is referenced only within a closed scope in Ada, the **for... use at...;** representation control is convenient.

```
procedure deal_with(C_array: system.address; length:integer)is
   the_array: array(integer range)..length-1) of integer;
      for the_array use at C_array;
   -- the_array(0) names C_array[0] of the C caller
```

If a C array is passed and must be preserved over an open scope (e.g., library), use a fixed-length array in the Ada program, making the Ada array at least as large as any

possible C parameter. Index checks should be suppressed since they will be redundant. Note the C_STRING type mentioned below. While it is sometimes desirable to create a dynamically-typed reference to the C array, it is impossible without copying the array unless an appropriate dope vector is build by the user. The dope vector is placed in memory immediately preceding the array value, so that an access to a dynamic array can be used.

Pointers and Address Types — Although pointer and address types are implementation-specific, the Ada tactic of using host conventions usually allows the use of Ada pointer and address types parallel to their C counterparts. If for some reason host conventions are not followed, or if the Ada compiler supports several host implementations of a particular type and must choose a particular representation, representation specifications can be used to custom tailor the size and range of the data type.

String Types — String types in C present a special case of the C array. A character string in C is represented by a pointer to the first character in an array of bytes. By convention, strings in C are terminated by a null character (16#00#) and store no explicit length. In Ada however, a string is represented by a packed array of type CHARACTER with the maximum number of components specified as part of the type.

A parallel type using the declaration in Ada is shown below.

```
type C_STRING is access STRING (1..INTEGER'LAST);
```

Although this type declaration will allow access to C strings, caution must be exercised when transferring strings between C and Ada variables. For example, a C string is an array of characters, but an Ada string (or any aggregate) has additional array information represented in a dope vector. An example using strings is presented *Step 4: Link with Existing Libraries*.

13.2.2 Step 2 — Declare External Subprograms

Once parallel types have been established, the next step is to gain access to subprograms and macros provided in the interface target package. This is accomplished first by using functions or macros and then by using the predefined **pragma** INTERFACE_NAME to establish a link from the Ada procedure or function name to the corresponding C function or macro.

pragma INTERFACE allows Ada programs to call subroutines defined in other languages: C, PASCAL, FORTRAN, and UNCHECKED. The definition of **pragma** INTERFACE does not allow access to subprograms whose names are Ada reserved words or cannot be expressed as an Ada identifier.

For a discussion of **pragma** INTERFACE and calls to other languages, **see also** *IMPLEMENTATION REFERENCE, 18.14 INTERFACE PROGRAMMING*, page 18-25.

Verdix has added **pragma** INTERFACE_NAME which gives the exact name of the the external subprogram and uses the format shown below.

```
pragma INTERFACE_NAME (Ada_name, "link_name");
```

The first parameter is the name of the Ada subprogram, the second is the name of the target subprogram as known to the linker. In addition, the *link_name* argument may be formed from a string literal, a constant string object, or a catenation of these operands. This allows a system independent interface to common routines for different versions of an operating system. For example:

```
pragma INTERFACE_NAME (c_exit, C_PREFIX & "exit")
```

where

```
package C_conventions is
   --4.2BSD
   C_PREFIX:constant string :="_";
end;
```

makes the **pragma** INTERFACE_NAME shown above work for 4.2BSD UNIX and the following will make it work for System V UNIX:

```
package C_conventions is
   4.2 BSD
   C_PREFIX:constant string :=" "; -- no prefix
end;
```

If *Ada_name* denotes a subprogram, a **pragma** INTERFACE must have already been specified for the subprogram.

This form allows inter-language communication ability with a minimum of effort.

The Ada compiler handles parameter pushing and target language compiler naming conventions and checks to make sure the parameters are allowed in the target language. For example, Ada fixed point types do not have a C equivalent. C compilers generally push parameters from right-to-left and generate assembly instructions in which C subprogram names are usually preceded with an underscore.

There is no mechanism for accessing a global variable from another language module directly from an Ada program in many Ada implementations. VADS provides such a mechanism in **pragma** INTERFACE_NAME. It allows access to external variables in the same way as it does for subprograms, shown above. For example:

```
pragma INTERFACE_NAME (Ada_variable, "variable_link_name");
```

Access to external variables could be provided by extending the external library with procedural access to these variables, but this pragma allows faster-executing code

and does not require additional code to be written in the language of the external library.

Finally, because the interface between an Ada routine and an external routine might involve an intermediate Ada function call, **pragma** INLINE can be used to eliminate the overhead of the extra call.

Because Ada does not support macros and preprocessing, the C macros must be defined as Ada procedures. However, **pragma** INLINE causes procedures to be treated as if they were indeed macros with the added benefit of semantic checking. Default parameters are also available in Ada so that instances of C functions that use macros solely for the purpose of providing for default parameters can be even more easily written in Ada.

With **curses**, most cursor movement is done using window-specific functions. These window-specific routines are then used to define a set of macros that act on the default window **stdscr**, and these routine macros are in turn used to define more macros. For example, the procedure to add a character to a window, **waddch( )**, is shown below.

```
waddch(win,ch)
WINDOW win;
char ch;

#define addch(ch) VOID(waddch(stdscr,ch))
#define mvwaddstr(win,y,x,str)
VOID (wmove (win,y,x) == ERR ? ERR : waddstr(win, str))
#define mvaddstr (y, x, str) mvwaddstr (stdscr, y, x, str)
```

There are still some problems in using **pragma** INTERFACE for subprograms of this type that may not be immediately apparent. First, all parameters must be of mode **in** or 'call by value', and functions may only return results that are scalar, access, or SYSTEM.ADDRESS types. Second, this naming convention only allows the Ada programmer the ability to use the C label for an interface routine. This can become awkward when the language interfaced to supports different naming conventions than those allowed in Ada or when the creation of an intermediate routine is required to overcome the first restriction while preserving the original calling and naming conventions.

The most effective way to circumvent the first restriction is to pass other types of parameters than those listed by reference, using the predefined address attribute (Ada RM 13.7.2). The second restriction can be overcome using **pragma** INTERFACE_NAME to map between the Ada and C subprogram names.

The use of this pragma and the address attribute are both useful for **waddstr** (window add string). This subprogram involves conflicting data structures and parameter

passing techniques. The Ada string is passed as two units, the array itself and the length or dope vector. In contrast, its C counterpart is passed by transferring a pointer to the first character of the string.

The most obvious solution is to create an intermediate routine that will convert an Ada string input into a C-string format before before calling the C routine. **pragma** INTERFACE_NAME facilitates this operation while allowing the procedure to remain unchanged from the user's standpoint. An example using **waddstr** is shown below.

```
-- package spec
procedure WADDSTR( WIN: WINDOW; S: STRING );
procedure C_WADDSTR( win: WINDOW; STR: ADDRESS );

pragma INTERFACE( C, C_waddstr);
pragma INTERFACE_NAME(C_waddstr, "_waddstr" );

-- package body
procedure WADDSTR( WIN: WINDOW; S: STRING ) is
  T: STRING(1..(S'LAST + 1));
begin
  T(1..S'LAST) := S;
  T(S'LAST + 1) := ASCII.NUL;
  C_WADDSTR( win, T'ADDRESS );
end WADDSTR;
```

From the compiler's standpoint, the effect of **pragma** INTERFACE_NAME is to substitute the *link_name* given ("_waddstr") when generating the reference instead of the Ada label.

Once the interface correspondence between the C functions has been established, the programmer can begin defining macros in terms of these functions. In our example using the file **curses.h** as a template in conjunction with the interface declarations for WADDSTR declared in the last section, we can create the default window counterpart, the Ada version of the macro ADDSTR.

A C macro takes the form of a **#define** followed by the macro specifics. In this case **addstr** is defined as the following.

```
#define addstr(str)  VOID(waddstr(stdscr, str))
```

The variable **str** is already defined as a C string (*char) type and **stdscr** is the default window used by **curses**. Using this definition, our Ada procedure counterpart becomes the following.

```
-- package spec
procedure ADDSTR( S: STRING );

-- package body
procedure ADDSTR( S: STRING ) is
begin
  WADDSTR( STDSCR, S );
end ADDSTR;
```

Notice that the type declaration of S is parallel to the Ada procedure WADDSTR type declaration defined earlier and not the C interface type declaration for **str** from **C_waddstr**.

The majority of the **curses** screen routines can be created in this manner using intermediate routines and passing addresses or simulating macros when appropriate. There is a disadvantage in the form of additional overhead generated from the call intensive execution of these simulated macros, but this is addressed in *Step 5: Overhead Reduction*.

13.2.3 Step 3 — Access Global Variables

The third step when building a complete package from a C library is to gain access to global variables declared in C from Ada. Although not supported in the standard language definition, access to these items is essential in a package like **curses** where a number of global variables are provided for programmer use or use by internal routines. This connection is established by the VADS compiler with **pragma** INTERFACE_NAME.

An example of this pragma used to access the **curses** global variables LINES and COLS is shown below.

```
LINES, COLS : INTEGER;
pragma INTERFACE_NAME( LINES, "_LINES");
pragma INTERFACE_NAME( COLS , "_COLS" );
```

pragma INTERFACE_NAME can also be used to make structures that would normally be inaccessible available to the user. For example, the two variables **curscr** and **stdscr** are pointers to **curses** window structures and are the key to implementing changes on a screen in most of the **curses** routines. Using this pragma we can declare the following.

```
Stdscr : SYSTEM.ADDRESS;
Curscr : SYSTEM.ADDRESS;

Pragma  INTERFACE_NAME(STDSCR, "_stdscr");
Pragma  INTERFACE_NAME(CURSCR, "_curscr");
```

The first two statements declare the variables **curscr** and **stdscr** as predefined Ada variables; the latter two associate the same memory locations for the Ada variables

as those occupied by the C variables. As noted earlier, the C record access method allows the programmer access to these structures by base address as long as access to internal structure values is not necessary. Any subsequent reference to these labels from Ada will refer to the identical data structure currently being utilized by the C subprograms. As far as the compiler is concerned, both forms of this statement are equivalent to generating a reference in the object module for a variable with the linker name **_stdscr** or **_curscr**.

13.2.4 Step 4 — Link With Existing Libraries

A direct outgrowth of the EXTERNAL_NAME pragma is the key to conquering the fourth goal, the ability to map parallel data structures.

pragma EXTERNAL_NAME allows a specific link name to be given to an Ada subprogram so that it might, for example, be called from another language.

```
pragma EXTERNAL_NAME (Ada_proc, "Ada_proc");
```

In this example, the external link name is the string 'Ada_proc.' The link name is formed using the same rules as for **pragma** INTERFACE_NAME.

Examples of using this technique can be found in the **curses** library package, where the window structure, for instance, is declared in C as a record containing such things as the current cursor position, certain terminal attributes, and pointers to the actual screen structure.

```
struct _win_st{
  short _cury,  _curx;
  short _maxy,  _maxx;
  short _begy,  _begx;
  short _flags;
  bool  _clear;
  bool  _leave;
  bool  _scroll;
  char  **_y;
  short *_firstch;
  short *_lastch;
};
struct _win_st  *_nextp,
                *_orig;
```

To implement the macros **flushok (win, bf)**, **getyx (win, y, x)**, and **winch (win)** correctly, it is necessary to gain access to fields that reside within the window structure. The first step is to create a parallel Ada record that can be used as a template to place over the C structure.

```
Type WIN_STRUCTURE;
Type WIN_POINTER is access WIN_STRUCTURE;
Type WIN_STRUCTURE is
record
  CURY, CURX    :  SHORT;
  MAXY, MAXX    :  SHORT;
  BEGY, BEGX    :  SHORT;
  FLAGS         :  SHORT;
  CLEAR         :  BOOL;
  LEAVE         :  BOOL;
  SCROLL        :  BOOL;
  YBAR          :  STRING_POINTER;
  FIRST_CH      :  SHORT_POINTER;
  LAST_CH       :  SHORT_POINTER;
  NEXT_TP       :  WIN_POINTER;
  ORIG          :  WIN_POINTER;
end record;

type WINDOW is access WIN_STRUCTURE;
```

This declaration is then followed by type and INTERFACE_NAME declarations that essentially create labels for all values stored within the C structure, as illustrated in the examples below.

```
CURSCR : WINDOW
Pragma INTERFACE_NAME(CURSCR, "_curscr");

STDSCR : WINDOW
Pragma INTERFACE_NAME(STDSCR, "_stdscr");
```

This allows the programmer to declare any subsequent windows that are required as type WINDOW and gain similar structure access. Procedures like **getyx** can now be written as the following.

```
Procedure GETYX(WIN:WINDOW; Y, X: out INTEGER) is
begin
  Y:=INTEGER(WIN.CURY);
  X:=INTEGER(WIN.CURX);
end GETYX;
```

A procedure like **scrollok** simply becomes the following.

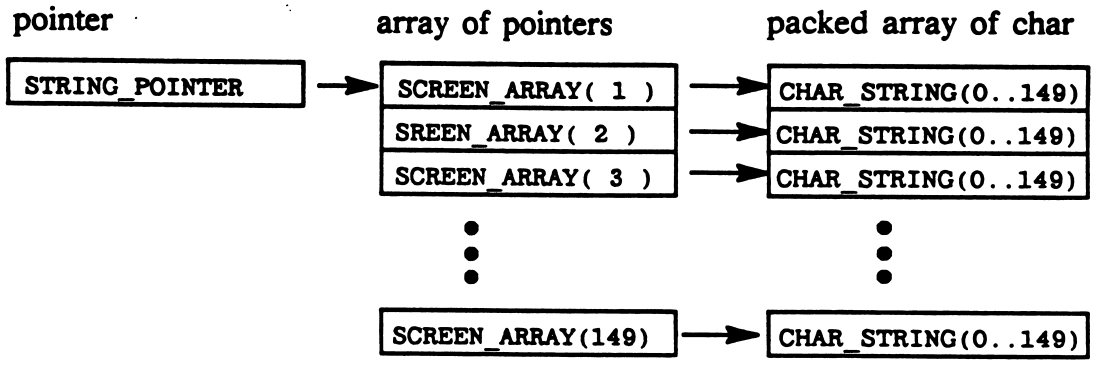
```
procedure SCROLLOK(WIN:WINDOW; BF : BOOLEAN) is
begin
  WIN.SCOLL:=BF;
end SCROLLOK;
```

However, this process does get more complex when implementation of a C macro like **winch** is required. This routine requires extracting a character from the screen at the current cursor location, making access to the screen character storage structure necessary. Using the C WINDOW structure for reference, it becomes clear that the

screen is represented by a two-dimensional C array of characters. In Ada this translates into a pointer to an array of C_Strings, which of course are represented by pointers to arrays of characters terminated by nulls. Although hard to visualize, this structure can be created using the following declaration.

```
type CHAR_STRING is array(0..149) of TINY_INTEGER;
type STRING_ACCESS is access CHAR_STRING;
type SCREEN_ARRAY is array(0..149) of STRING_ACCESS;
type STRING_POINTER is access SCREEN_ARRAY;
```

The final type declaration shown above (STRING_POINTER) is then used as the type declaration for the YBAR field in the Ada WINDOW structure. When the Ada structure is then overlaid on the C window structure, it creates a parallel character storage structure by capturing the pointer stored in the C variable _y in the Ada YBAR field. A diagram of this structure is shown below.



Ada requires limits on arrays at compilation time, hence the 150 component restriction on the arrays declared. However, this does not affect the usefulness of our mapping as long as SCREEN_ARRAY(0..ROWS-1) and CHAR_STRING(0..COLS-1) are overlaid correctly on the corresponding C structures in memory. The call to WINCH now becomes the following.

```
function WINCH( WIN: WINDOW ) return CHARACTER is
  T: TINY_INTEGER;
begin
  T := WIN.YBAR.ALL( INTEGER( WIN.CURY )).ALL( INTEGER(WIN.CURX ));
  T := ABS( T );
  return CHARACTER'VAL( T );
end WINCH;
```

Another interesting problem arises when implementing terminal characteristic macros. These operations are performed by masking bits contained in the short integer **sg_flags** of the C structure **sgttyb** given below. This field is then used as a packed array of boolean bit flags indicating the status of different terminal characteristics.

```
struct sgttyb {
  char  sg_ispeed;      /* input speed */
  char  sg_ospeed;      /* output speed */
  char  sg_erase;       /* erase character */
  char  sg_kill;        /* kill character */
  short sg_flags;       /* mode flags */
};
```

The operations on the **sg_flags** field are accomplished by means of bitwise ANDs and ORs setting or clearing flags as needed. The Ada language, in contrast, provides no provision for bitwise ANDs and ORs on integers. It does, however, provide for such operations on arrays of booleans. The key to resolution of this problem is to define an array of booleans mapped onto the integer storage defined by the C program. Once a connection is established using **pragma** INTERFACE_NAME, unchecked conversion can be used to convert the SHORT_INTEGER field to a packed array of boolean. In this form the fields can be altered by using the predefined Ada **and** and **or** operations in conjunction with the appropriate mask values. After flags have been altered the process of conversion can be reversed. Using this technique, a procedure like **echo** becomes the following.

```
procedure ECHO is
begin
  TTY.SG_FLAGS := PACK( UNPACK( TTY.SG_FLAGS ) OR UNPACK( M_ECHO ));
  ECHOIT := TRUE;
  TTY_POINTER := TTY'ADDRESS;
  STTY( TTY_CH, TTY_POINTER );
end ECHO;
```

The functions PACK and UNPACK are actually renamed instantiations of the **generic function** UNCHECKED_CONVERSION. For portability's sake, the **functions** PACK and UNPACK could be simulated by division on compiler systems not supporting bit level packing.

13.2.5 Step 5 — Overhead Reduction

The final step in the interface process is to reduce as much of the overhead resulting from frequent subprogram calls. In the C code, this was done with macros masquerading as functions, and the effect is to place the code inline. The Ada solution to this problem is provided by use of the predefined **pragma** INLINE (Ada RM 6.3.2). When used with secondary routines, or routines that are dependent on calls to others, this pragma compiles the C object code of the called routine into the Ada object code of the calling routine resulting in reduced stack manipulation and fewer subprogram calls during execution.

This pragma is best applied to Ada programs only after testing and debugging is done because narrowing down errors during debugging is more difficult with inline code.

(The pragmas can be placed appropriately in the code when constructing it, and temporarily commented out until the debugging phase is complete.) Depending upon the length of the files being inlined, this pragma also has the potential to greatly expand object file size so caution should be exercised when disk space is at a premium.

13.3 PROGRAM CONVERSION

Moving projects into Ada can be a lengthy process since the code being replaced may have been developed over many years. One modular approach is to write an Ada 'wrapper' program that surrounds the subprograms in other languages and allows them to be gradually replaced. **pragma** INTERFACE and **pragma** INTERFACE_NAME are useful for this gradual replacement, but because of the interrelation of subprograms that call many other subprograms, it is not normally possible to replace the upper level subprogram without rewriting all its dependents in Ada as well. VADS provides another pragma, EXTERNAL_NAME, that makes this replacement simpler. **pragma** EXTERNAL_NAME allows subprograms in other languages to call Ada subprograms: exactly the reverse of the INTERFACE and INTERFACE_NAME pragmas.

To illustrate, imagine that the program below was originally written in C.

```
/* C program example that calls an Ada subprogram with a
** parameter declared in C */
#include <stdio.h>

char *gets ();
int atoi ();

int service_number;
extern void ada_put ();        /* Ada: TEXT_IO.INTEGER_IO.PUT of an
                                  INTEGER */

main ()
{
  char buf[80];

  printf ("Enter an integer here: ");
  gets (buf);
  service_number = atoi (buf);
  ada_put (service_number);    /* originally used printf */
  putchar ('\n');
}
```

The **printf** call was replaced with ADA_PUT and an Ada package containing the procedure ADA_PUT and interface declarations for the C entities were written.

```
with TEXT_IO; use TEXT_IO;

package C_INTERFACE is
  SERVICE_NUMBER : INTEGER;
  pragma INTERFACE_NAME (SERVICE_NUMBER, "_service_number");

  procedure ADA_PUT (I : INTEGER);
  pragma EXTERNAL_NAME (ADA_PUT, "_ada_put");

  procedure MAIN;
  pragma INTERFACE (C, MAIN);              -- The name "main" is not
  pragma INTERFACE_NAME (MAIN, "_main");   -- special in in Ada: this
end C_INTERFACE;                           -- is used to call the
                                           -- C function, "main" from
                                           -- the Ada wrapper

package body C_INTERFACE is
  procedure ADA_PUT (I : INTEGER) is
    package I_IO is new INTEGER_IO (INTEGER);
    use I_IO;
  begin
    PUT (I);
  end ADA_PUT;
end C_INTERFACE;
```

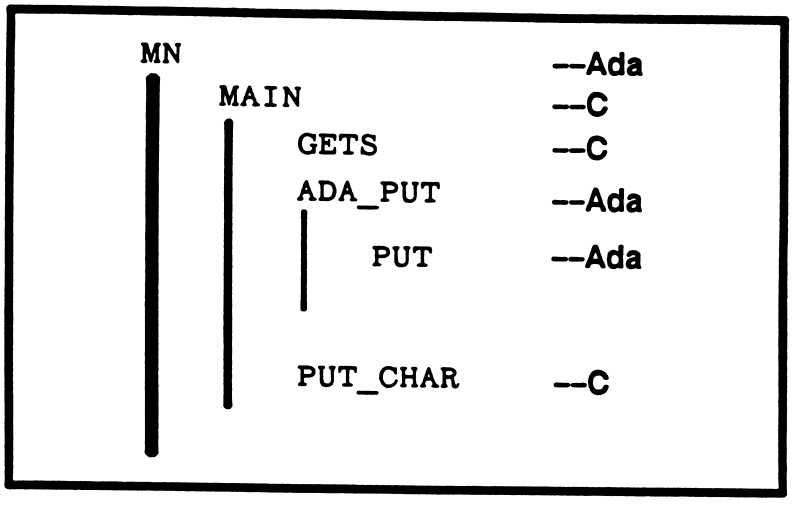
Now a simple Ada 'wrapper' to call the original C function **main** is written so that the linker **a.ld** can resolve all the references in the modules and perform its usual elaboration order checks.

```
with C_INTERFACE; use C_INTERFACE;
procedure MN is
begin
   MAIN;
end;
```

Compiling both the C and Ada portions can be done by

- compiling the C portion
- using the VADS linker to construct the 'main' program MN and including the C object in the link

The program now runs correctly. Its Ada calling sequence is graphically shown below.



The real benefit for the user is that new portions of large programs can be developed in Ada, but existing, tested, working code need not be replaced wholesale: individual modules or groups of modules can be replaced by newly developed Ada code without undue restrictions on the language of calling or called subprograms. An additional benefit is that once subprogram parameters are defined in Ada, the compiler will perform its usual type checking across subprograms.

13.4 CALLING ADA FROM OTHER LANGUAGES

Ada compilations generate objects like other compilers; these can be linked with objects generated by other compilers and can therefore be called from programs written in other languages. However, a number of complexities arise. This section discusses them, but we nonetheless recommend that main programs be Ada.

13.4.1 Finding the Right Object

Ada hides the objects in the objects directory. Several objects are often generated during the compilation of a single source file. Each object for a file is generated with the file name used as a prefix and two characters at the end, starting with **01**, **02**, etc. For this reason, we recommend compiling each unit in a separate source file. The VADS mapping tool can be used to determine which object contains the needed entry point.

Ada also generates unusual compound names for entities such as subprograms. These names are used to disambiguate overloading and assist in debugging. The names usually have the form:

_A_*subname*llXcc.*parent*

'subname' is the name of the sub-entity. 'll' is the line number of its definition. 'X' is S if defined in the spec and B for the body. 'cc' is the character number of its definition and *'parent'* is the name of the parent unit, less its prefix of '_A_'

Generated names contain line or character numbers and will change if the source that contains them changes. Therefore we recommend that you give explicit names to any entities that you wish to reference from other languages. See **pragma** EXTERNAL_NAME.

13.4.2 Avoiding Elaboration

This section applies to Archives of Ada object files.

Ada semantics dictate that units not be used before they are elaborated. PAEDS generates dynamic elaboration checks for many subprograms, to verify this property. If a user just links in and calls an Ada package the PROGRAM_ERROR exception will almost certainly be raised. Two approaches can prevent this.

One possibility is to do the elaboration. The simplest technique is to artificially create a main program that calls (directly or indirectly) all the entry points needed from the other language, and link it with the verbose option, saving the output in a file. An object having the file name with an 'o' appended is created, holding an ELABORATION_TABLE, among other things. To elaborate all units, the start up procedure simply calls each address in that table, in the given order – the table is an array of elaboration subprogram addresses.

For information concerning the elaboration of Library Units, **see also** Ada RM *10.5*.

Another possibility is to avoid elaboration. To do so, use **pragma** SUPPRESS (ELABORATION_CHECK) in every unit to be linked. Also avoid 'complex' initialization in units to be linked: complex initialization is any statically allocated object with an initial value that is not either an Ada static value, a value of system address, a record without a representation clause of these items, or an array of these items.

For a description of static expressions, **see also** Ada RM 4.9

Any complex initialization will generate code into an elaboration procedure. If you expect not to do elaboration, then these complex initializations would not get done. On most VADS systems, the disassembler can help detect and avoid complex initializations. Also linking only library subprograms assures that no elaboration is needed. Verdix supports **pragma** NOT_ELABORATED to assist in creating units that do no elaboration.

13.4.3 Linking a Non-Ada Main Program

To link, simply add the names of all appropriate VADS object files to the list of files given to the linker. If elaboration is intended, then link the file described above under

Avoiding Elaboration, using an artificial main program. You may wish to create a library or archive of these Ada objects for easier use.

13.4.4 Runtime Considerations

Each programming language has its own runtime system. Only one of these runtime systems can normally be used in a given execution. Normally that would be the runtime initialized by the program's startup entry point (**_ _start** for Ada). The VADS linker creates Ada programs that call the VADS **_ _start** entry point, initializing the VADS Runtime System. That runtime 'captures' the machine and, among other things, sets up signal (interrupt) handlers for all Ada-important signals. If the Ada runtime is *not* in control of the machine, then the user should not use tasking, exceptions, exception handling, or Ada I/O. ADA_IO requires elaboration.

13.4.5 Summary: Calling Ada From Other Languages.

If the main program is Ada, then calls can be made from Ada to non-Ada, and then from non-Ada back to Ada (once elaboration is complete) with no concern other than using **pragma** EXTERNAL_NAME . However, if the main program is not Ada, then many complexities arise: in this case Verdix recommends that only simple library-level subprograms be linked with non-Ada main programs.

CHAPTER 14

SAMPLE PROJECT

14.1 OVERVIEW

In the following pages, we will walk through a sample software development project. This material is presented informally as it might occur among the programmers involved. (*Note that the source code shown is not complete.*)

Two activities are shown going on in parallel to illustrate the interaction of VADS libraries. One project, called TIME, will provide some handy utilities relating to the input and output of calendar times. The second project, CAL, will use TIME to provide a desk calendar facility. These projects are presented as illustrations; their descriptions stop far short of providing usable calendar software.

14.2 THE PROJECT PLAN

Our project begins with Marti and Ashley deciding to build a desk calendar system to organize their daily activities. They agree that the package should allow its users to assign a priority to each activity and should provide a number of facilities, including listing a day's events in priority order.

14.3 USING a.mklib

Marti decides to handle the prioritizer, and Ashley will work on the input/output interface. They define a CAL project, and, since they do not need access to anything except the default libraries, put the project's library directory under Marti's home directory.

In the following examples, the programmers are using the Bourne shell, and the default Bourne shell prompt, **$**, is shown preceding user input.

Marti knows that, on her system, the Ada standard library has been installed in /usr2/vads5/standard. She uses the **a.mklib** command to create a library in which to do her development.

```
$ a.mklib /marti/work/CAL  /usr2/vads5/standard
```

Note: the recommended use of **a.mklib** is with **-i**. For more information, see also *LIBRARY MANAGEMENT TOOLS*, **a.mklib**, page 4-8.

Marti enters the project's library directory using the change directory command.

```
$ cd  /marti/work/CAL
```

Using the vi(1) editor, she enters a rough data structure into a file called **apts.a**.

```
$ vi  apts.a
. . .
```

14.4 USING ada AND a.error

Marti compiles her new file using the following command and receives an error message.

```
$ ada  apts.a
/marti/work/CAL/apts.a line 11, char 28:syntax error: "identifier
   range" inserted
```

At this point, Marti and Ashley could just edit **apts.a**, go to line 11, guess where character 28 is, and hope to insert the expected syntax. They instead consult their *VADS REFERENCE* where they learn about **a.error** and its invocation from **ada**.

They choose to use the **-v** option to **a.error** because it allows them to identify and correct errors in one step with their visual editor. Unfortunately, we cannot easily show the operation of the visual editor in this documentation, so we will show the programmers using the **-l** option of **a.error** to get listings. However, we highly recommend the **-v** option to **a.error** or, equivalently, the **-ev** option to **ada**. The **-l** option to **a.error** provides listings that are nearly identical to those produced with **-v**.

Marti types the following command, which displays the error listing shown below.

```
$ ada -el apts.a

 *************** /marti/work/CAL/apts.a ****************
   1:with calendar;
   2:package appointments is
   3:    type priorities is (if_bored, possible ,desired ,required);
   4:    type string_ptr is access string;
   5:    type appointment is record
   6:       date:         calendar.time;
   7:       priority:     priorities;
   8:       could_do:     string_ptr;
   9:    end  record;
  10:    max_to_do: constant := 5;
  11:       subtype to_do_range is 1..max_to_do;
A ------------------------------^
A:syntax error:  "identifier range" inserted
  12:    to_do:           array (to_do_range) of appointment;
  13:
  14:end;
```

The compiler is reminding us that in Ada, subtypes must explicitly state the base type. Marti corrects this line.

```
subtype to_do_range is integer range 1..max_to_do;
```

Marti recompiles:

```
$ ada -el apts.a
```

The package specification compiles correctly. Ashley now goes off to define the package body for this package while Marti starts work on the prioritizer by defining a test program in the file **cal.a**.

```
$ vi cal.a
. . .

$ ada -el cal.a

**************  /marti/work/CAL/cal.a  ****************
   1:-- The beginnings of a calendar program for keeping track
   2:-- of prioritized appointments. This "first cut" defines a
   3:-- set of dummy appointments just to test the selection of
   4:-- the highest priority appointment.
   5:
   6:with appointments, text_io;
   7:use appointments;
   8:procedure cal is
   9:    most_important: integer := 0;
  10:    NOTHING_TO_DO:  exception;
  11:    function "<"(left,right:appointment) return boolean
A ---------------^
A:error: RM 10.2(5):operator symbol not allowed as stub id.
  12:    is separate;
  13:begin
  14:    get_appointments;
A -------^
A:error: RM 8.3: identifier undefined
  15:    for apt in first_appointment .. last_appointment loop
  16:       if most_important = 0 then most_important := apt;
  17:       elsif to_do (most_important) < to_do(apt) then
  18:          most_important := apt;
  19:       end if;
  20:    end loop;
  21:    if most_important = 0 then raise NOTHING_TO_DO; end if;
  22:    declare
  23:       what_to_do: appointment renames to_do(most_important);
  24:    begin
  25:       text_io.put_line("AT" & readable_date(what_to_do.date)&
A ---------------------------------^
A:error: RM 8.3: identifier undefined
  26:          ":    " & what_to_do.could_do.all);
  27:    end;
  28:end;
```

This test program has illuminated several requirements. A relatively simple problem is the "<" as a subunit. Marti declares the function with an identifier and then renames it as the operator symbol.

Marti also notes that the appointments package should provide an initialization step, which she calls GET_APPOINTMENTS. She corrects this error by asking Ashley to provide this new routine in her package, APPOINTMENTS.

Finally, Marti has noted a deficiency in the Ada CALENDAR package: there is no provision for printing a date in human-readable form, and thinks through the possibility of a support package extending some of the features of the standard package CALENDAR. Another programmer, Steve, volunteers to establish such a package in a general library and builds the following package specification.

```
$ cat ~steve/his/work/calendar_support.a

with calendar;
use calendar;
package my_times is
   min:   constant day_duration := 60.0;
   subtype twelve_hours is integer range 1..12;
   function am(t: twelve_hours) return day_duration;
   function pm(t: twelve_hours) return day_duration;
   function readable_date(d: time) return string;
end;
```

14.5 USING a.path

Marti and Ashley wonder how to use Steve's new package, since it has been built in a completely separate library directory. Although Steve could move or copy it into the main library directory — since it will probably be there eventually — it is better for him to debug his program before he makes it generally available. Marti and Ashley agree to be one of his test cases, and they connect his library to theirs.

```
$ cd /marti/work/CAL
$ a.path -i ~steve/his/work
```

To be certain that Steve's directory is connected to Marti's CAL directory, they use **a.path** a second time to show the new library search path.

```
$ a.path

library search list:
       /usr2/steve/his/work
       /usr2/vads5/standard
```

Now, anything compiled in **/usr2/steve/his/work** will be available to Marti and Ashley.

Ashley amends **apts.a** to include a definition of GET_APPOINTMENTS and builds a file **aptsB.a** to start a dummy body for **apts.a**.

```
$ cat aptsB.a

with calendar;
package body appointments is
procedure get_appointments is
   begin
      to_do  :=
         ( ( time_of(1984,11,14,am(9) + 10*min),
             if_bored,
             new string'("fill out a status
           ( time_of(1984,11,14,pm(3)),
             desired,
             new string'("code up some Ada")),
           ( time_of(1984,11,14,pm(5)),
             required,
             new string'("have a beer with the gang"))
         );
      first_appointment := 1;
      last_appointment := 1;
   end get_appointments;
end;
```

Marti is also busy building a dummy body for the '<' separate function for the APPOINTMENTS package.

14.6 USING a.make

With all these changes and files, things are starting to get out of hand. Ashley and Marti have defined four files and have just begun the implementation. Soon they will have many files, especially if they follow good programming practice and (1) define many easily tested small units and (2) keep most units in individual files. Fortunately, VADS provides several tools to keep projects organized. The first of these tools is **a.make**, which will determine those files needing recompilation and accomplish this in the correct dependent order.

Marti types the required command.

```
$ a.make -v cal -f *.a

finding dependents of: apts.a
finding dependents of: aptsB.a
finding dependents of: cal.a
finding dependents of: calS.a
compiling apts.a
   spec of appointments
compiling cal.a
   body of cal
compiling calS.a
   body of cal.apts_less
compiling aptsB.a
   body of appointments
a.ld cal
Undefined:
_A_am.6S11.calendar_support
_A_pm.7S11.calendar_support
_A_readable_date.8S11.calendar_support
```

Marti's **a.make** has terminated in error. Three items were 'undefined' after the attempt to link by **a.ld**. These three names are in Ada internal name format and are generated by the UNIX linker. We can see, however, that they represent the three subprograms AM, PM, and READABLE_DATE that Steve was to provide in CALENDAR_SUPPORT.

14.7 USING a.ld

Steve has not finished a body for CALENDAR_SUPPORT. With help, he gets one written and compiled; and Marti can try **a.make** again. She could type either

```
a.ld cal
```

or

```
a.make cal
```

Marti runs the program by typing the following.

```
$ a.out

AT 1987/ 11/ 14  9: 10: 0:  fill out a status report
```

14.8 USING VADS LIBRARIES

Marti, Ashley, and Steve now each have improvements to make in their code. However, if they all try to make changes at once, there will be confusion about which version of the code to use. VADS libraries make the process easier as each can set up a separate library referencing Marti's library **/marti/work/CAL**. Each can then make changes, compile, test, and correct independently of each other until their changes are stable enough to let the others try using them.

Marti creates a library to change priority testing.

```
$ a.mklib /marti/work/fix_cal /marti/work/CAL
$ cd /marti/work/fix_cal
```

The new directory starts out empty, but in it VADS will have access to all units in libraries referenced by the parent library. For example, Marti can build CAL in this directory and get the same result as she got while inside **/marti/work/CAL**. That is because VADS will find non-local units from the first library on the path in which they exist.

```
$ a.ld cal
$ a.out

AT 1987/ 11/ 14  9: 10: 0: fill out a status report
```

14.9 IMPROVING THE PROGRAM

Marti copies her subprogram into this local library and works on it.

```
$ cp /marti/work/CAL/calS.a .
$ cat calS.a

separate (cal)
function apts_less(left, right: appointment) return boolean is
begin
   return TRUE;              -- just dummied up
end;

$ vi calS.a

. . .

$ cat calS.a

separate(cal)
function apts_less( left, right: appointment) return boolean is
begin
   return left.priority < right.priority
end;

$ a.make -v cal -f *.a

finding dependents of:  calS.a
compiling calS.a
   body of cal.apts_less
a.ld cal
$ a.out
AT 1987/ 11/ 14  9: 10: 0:  fill out a status report
```

Marti's change to improve priority testing has not solved her problem. It turns out that priority is really a function of the absolute priority and the time left to do it. Marti again changes **calS.a** in the sublibrary **fix_cal**.

```
$ vi calS.a
. . .
$ a.make -v cal -f *.a

compiling calS.a
        body of cal.apts_less
/marti/work/fix_cal/calS.a,line 10,char 8:syntax error:"if"inserted
a.ld cal
```

Marti is back in the position of having to try to find line 10. Although **a.make** does not directly provide a way to intersperse error messages, Marti can accomplish this in two ways. One way is to use the **-C** option of **a.make** which causes the compiler to call **a.error** as shown below.

```
$ a.make -C "ada -el" cal

compiling ./calS.a
********** ./calS.a  **********
   1:separate (cal)
   2:function apts_less(left, right:appointment) return boolean is
   3:begin
   4:    if left.priority < right.priority
   5:    or else (left.priority = right.priority
   6:       and then left.date > right.date) then
   7:       return TRUE;
   8:    else
   9:       return FALSE;
  10:    end;
A ---------^
A:syntax error:"if" inserted
  11:end;
```

Marti would, of course, use the **-ev** option to bring up the visual editor to fix any problems, but we use **-el** for illustration here. The **a.make** line is getting a little long to type, and most users will probably start using C shell aliases to reduce typing.

Marti's other approach is to save the output of **a.make** in a log file and then supply the whole log file as input to **a.error**. **a.error** will sort out the errors into their respective files and produce listings or editor files as requested.

```
$ a.make cal >&  log
$ a.error -v log
. . .
```

Marti repairs her program, recompiles, fixes a semantic error or two, and finally gets the result shown below.

```
$ a.make -C "ada -el" cal
$ a.out
AT 1984/ 11/ 14   9: 10: 0: fill out a status report
```

Unfortunately, this result is still wrong since 'have a beer with the gang' has the highest priority.

14.10 USING a.db

Marti decides to use the VADS debugger.

We show Marti using the line-oriented debugger interface; an example of the screen-oriented debugger interface appears in *Sample Screen-mode Debugging Session*. The following is a transcript of the debugging session in which Marti tracked down the bug, in this case due to Ashley's mistake in the dummy body of GET_APPOINTMENTS.

She invokes the debugger on the default file, **a.out** and uses the advance command (**a**) to execute the program up to the next source line in the current unit. In this case, since she is just starting, it means execute up to the first line in the main program.

```
$ a.db a.out
Debugging: /marti/work/fix_cal/a.out
>a

a.out
 stopped at "/marti/work/CAL/cal.a":9 in cal
 9   most_important: integer := 0;
```

Using the **a** command, Marti steps through a number of statements.

```
>a

stopped at "/marti/work/CAL/cal.a":11 in cal
 11    function apts_less( left, right: appointment) return boolean;
>a

 stopped at  "/marti/work/CAL/cal.a" :15 in cal
 15  get_appointments;
>a

stopped at "/marti/work/CAL/cal.a":16 in cal
 16  for apt in first_appointment .. last_appointment loop
>a

stopped at "/marti/work/CAL/cal.a":17 in cal
 17     if most_important = 0 then most_important := apt;
>a

stopped at "/marti/work/CAL/cal.a":20 in cal
 20     end if;
>a

stopped at "/marti/work/CAL/cal.a":22 in cal
 22  if most_important = 0 then raise NOTHING_TO_DO; end if;
```

Marti notices that the loop has ended after just one iteration, but it should have executed once for each of the three dummy entries in the TO_DO table.

She uses **p** to display the value of LAST_APPOINTMENT.

```
>p last_appointment

1
```

LAST_APPOINTMENT is wrong; it should be 3, not 1.

Marti knows that GET_APPOINTMENT must set LAST_APPOINTMENT, and she sets a breakpoint there and restarts the program.

```
>b 15
>r

a.out
[2] stopped at "/marti/work/CAL/cal.a":15 in cal
 15 get_appointments;
```

Marti steps into Ashley's code, listing it to see what is wrong.

```
>s

stopped at "/marti/work/CAL/aptsB.a":4 in get_appointments
  4 procedure get_appointments is
>l

4* procedure get_appointments is
5     begin
6        to_do :=
7           ( ( calendar.time_of(1984,11,14,am(9) + 10*min),
8               if_bored,
9               new string'("fill out a status report")),
10            ( calendar.time_of(1984,11,14,pm(3)),
11              desired,
12              new string'("code up some Ada")),
13            ( calendar.time_of(1984,11,14,pm(5)),
14<             required,
>
14              required,
15              new string'("have a beer with the gang"))
16          );
17        first_appointment := 1;
18        last_appointment := 1;
19     end get_appointments;
20< end;
```

There it is: LAST_ APPOINTMENT is mis-set. (Note that Marti could have used the editor-like search capability of the debugger to find this line without listing.)

She types **bd**, for breakpoint down, so that she will get control after GET_APPOINTMENT is done and continues.

```
>bd
>g

[3]- stopped at
  16   for apt in first_appointment .. last_appointment loop
```

She now has correct the value for LAST_ APPOINTMENT and continues.

```
>last_appointment := 3

3/1

>g

AT 1984/ 11/ 14 17: 0: 0: have a beer with the gang
```

GREAT! The expected result:

```
process has exited with status 0
reloading /marti/work/fix_cal/a.out

>quit
```

Marti can either fix Ashley's bug locally, or she can tell Ashley about it and let her fix it.

14.11 USING a.which

Ashley asks Marti to fix it, but forgets to tell her which library version of APPOINTMENTS she was using or what file it is in. Marti uses the VADS utility **a.which** to find out.

Marti wants to locate the current body for appointments.

```
$ a.which -b appointments

/marti/work/CAL/aptsB.a
```

She copies the file to her own local directory.

```
$ cp /marti/work/CAL/aptsB.a .
```

Marti changes LAST_APPOINTMENT and then uses the **-f** option to introduce new files into the **a.make** process.

```
> vi aptsB.a

. . .

$ a.make -v cal -f aptsB.a

finding dependents of: aptsB.a
compiling aptsB.a
   body of appointments
a.ld cal
$ a.out
AT 1987/ 11/ 14  17: 0: 0:   have a beer with the gang
```

14.12 USING a.rm

Marti's first step is finished, and it appears that she can now make her improved versions available to her fellow programmers.

```
$ cp aptsB.a calS.a /marti/work/CAL
$ a.rm aptsB.a calS.a

$ cd /marti/work/CAL
$ a.make -v cal

compiling ./calS.a
   units: body of cal.apts_less
compiling ./aptsB.a
   units: body of appointments
a.ld cal
```

Marti's **fix_cal** directory is now empty of Ada units and could be deleted, but she will keep it for continued development of her CAL package. She could have used **a.cleanlib** rather than **a.rm**, achieving the same result. In fact, if she does not need the subdirectory, **fix_cal**, she can use UNIX **rm**(1) **-fr** to get rid of the whole directory, *after* copying up her fixed files.

```
$ rm -fr /marti/work/fix_cal
```

14.13 USING a.du

Marti can use **a.du** to see what units are active in her libraries. **a.du** tells her the sizes (in bytes) of the various files that represent Ada units: the separate compilation information (**.nets**), the object file (**.objects**), and the line number debugging file (**.lines**).

```
$ cd /marti/work/CAL
$ a.du

    .nets    .objects    .lines
     4680         413        54  spec of appointments
                                    in ./apts.a
     5126        1385       138  body of appointments
                                    in ./aptsB.a
     5870        1727       125  body of cal
                                    in ./cal.a
     1670         447        97  body of cal.apts_less
                                    in ./calS.a
```

14.14 USING a.tags

We will follow Marti through one last problem: how can she find units when they are distributed among several — even hundreds — of files? VADS provides a tool called **a.tags** that works with the UNIX editor **vi**(1) and some other editors.

```
$ a.tags *.a
```

Marti can edit a unit by name instead of remembering the file name.

```
$ vi -t cal
```

The editor will bring up **cal.a**.

a.tags also allows access to entities declared within units, specifically to subprograms, tasks, packages, and optionally to types. Further, when Marti is within the editor, she can position the cursor at the first character of a name and then type <CONTROL>].

The editor will look in the **tags** file, find the declaration of that name, and switch over to that declaration. Be aware that while tags are handy, they are not kept up to date as your program changes and should be remade periodically.

CHAPTER 15

SAMPLE SCREEN-MODE DEBUGGING SESSION

15.1 OVERVIEW

The symbolic debugger, **a.db**, is a highly interactive tool, and this chapter shows a lengthy example of using it in its screen-oriented mode. In order to show the reasoning behind the debugging sessions, a more informal, conversational style is used.

15.2 THE PROCEDURE TO BE DEBUGGED

Imagine that you have written a procedure to translate dates of the form 'February 3, 1954' into the '2/3/54' format. Your program calls the procedure shown below with several different test dates.

```
-- stk1.a --
package date_mgt is
   type date_rep is
   record
      month: integer;
      day:   integer;
      year:  integer;
   end record;

   --
   -- Date must be of the form "November 7, 1985". If the month
   -- field returned is 0, the date is not syntactically correct.
   --
   function get_date_rep(date: string) return date_rep;
end;

package body date_mgt is
   type astring is access string;
   type month_list is array (1..12) of astring;
   month_tab: month_list := (
       1 => new string'("january"),
       2 => new string'("february"),
       3 => new string'("march"),
       4 => new string'("april"),
       5 => new string'("may"),
       6 => new string'("june"),
       7 => new string'("july"),
       8 => new string'("august"),
```

```
        9 => new string'("september"),
       10 => new string'("october"),
       11 => new string'("november"),
       12 => new string'("december"));

procedure get_number(src: string; val, pos: out integer)
   is separate;


function month_no(month: string) return integer is
begin
   for i in month_tab' range loop
      if month = month_tab(i) .all then
         return i;
      end if;
   end loop;
   return 0;
end;

function get_date_rep(date: string) return date_rep is
   type adate is access date_rep;
   d: adate := new date_rep'(0,0,0);
   the_month: integer;

   next: integer := date'first;
begin

   while (date(next) /= ' ') and (next <= date'last) loop
      next := next + 1;
   end loop;

   if next > date'last then
      return d.all; -- error exit;
   end if;

   -- We postpone assigning the month into d.month since 0
   -- indicates an error and it is 0 now.

   the_month := month_no(date(date'first..next-1));
   if the_month = 0 then
      return d.all; -- error exit
   end if;

   -- get the day
   get_number(date(next..date'last), d.day, next);
   if next = 0 then
      return d.all; -- error exit;
   end if;

   -- find the comma
   next := next + 1;
   while (date(next) /= ',') and (next <= date'last) loop
```

```
      next := next + 1;
   end loop;

   -- make sure there is more string left to hold the year
   next := next + 1;
   if next > date'last then
      return d.all; -- error exit;
   end if;

   -- Get the year
   get_number(date(next..date'last), d.year, next);
   if next = 0 then
      return d.all;
   end if;
      d.month := the_month;
      return d.all;
   end;
end;

with text_io;
package iio is new text_io.integer_io(integer);

with date_mgt; use date_mgt;
with text_io; use text_io;
with iio; use iio;
procedure stk is
   procedure test_date(date: string) is
      day: date_rep;
   begin
      day := get_date_rep(date);
      put('"');
      put(date);
      put(""" -- ");

      if day.month = 0 then
         put_line(" ERROR");
         return;
      end if;

      put(day.month);
      put('/');
      put(day.day);
      put('/');
      put(day.year - 1900);
      new_line;
   end;
begin
   -- first test some good dates.

   test_date("February 21, 1985");
   test_date("December 31, 1900");
   test_date("January 1, 1999");
```

```
   -- now test it with errors

   test_date("Aug. 1, 1981"); -- bad month
   test_date("March 7 1983"); -- no comma
   test_date("July 3");       -- no year
   test_date("June 3,");      -- comma, no year
end;

with io_exceptions;
with iio; use iio;
separate (date_mgt)
procedure get_number(src: string; val, pos: out integer) is
begin
   get(src, val, pos);           -- RM 14.3.7(13)

   exception
      when io_exceptions.DATA_ERROR =>
         pos := 0;
end;
```

15.3 SCREEN-ORIENTED DEBUGGING SESSION

Like any optimistic programmer, you run the program to see if it will work the first time.

It doesn't work — but it doesn't crash either — as the following output shows.

```
"February 21, 1985" -- ERROR
"December 31, 1900" -- ERROR
"January 1, 1999" -- ERROR
"Aug. 1, 1981" -- ERROR
"March 7 1983" -- ERROR
"July 3" -- ERROR
"June 3," -- ERROR
```

You decide to use the debugger to find out why perfectly good dates show up as errors. You invoke the debugger with

```
a.db -v stk
```

which brings up the debugger in screen mode, the better mode for 'exploring' a program. The first thing you do is change the tab stop setting from the UNIX default of 8 to 4 (:set tabs 4), which you prefer and have used in your source code. This causes the screen to be reprinted with the new tab stop setting.

There are several things to notice about the screen. The * next to line 104 indicates the current line. In this case, it shows that the next thing to be executed will be the procedure stk. The * in the dashed line (under line 117) tells you that the debugger is waiting for input. It will not appear while the debugger is actually executing a

program. At the right-hand end of the dashed line is the name of the source file being displayed.

```
103   with iio; use iio;
104*  procedure stk is
105      procedure test_date (date:string) is
106         day: date_rep;
107      begin
108         day: = get_date_rep (date);
109         put('"');
110         put(date);
111         put("""   --");
112
113         if day.month = 0 then
114            put_line("ERROR");
115            return;
116      end if;
117
-*-------------------------------------------------stk1.a--
Debugging: vc/sbq/tst3/stk
VADS_library: bc/code
library search list:
   bc/code
   bc/vads/standard
:vi
:set tabs 4
```

Now to run the program. You can start a program in one of four ways. The **r** (run) command is explicitly for this purpose. Used as **:r** in screen mode, it will process invocation parameters for the program and also handle I/O redirection. The parameter syntax supported by the **r** command is intended to be identical to the **csh**(1), since that is how most users invoke their programs. Once a set of invocation parameters are given, they are remembered by the debugger and do not need to be re-specified.

The three other ways to get it running are the **a** (advance), **s** (step), and **g** (go) commands. These commands cannot supply invocation parameters, but they will use the current parameters.

The program you are debugging does not use invocation parameters, and you start it using the **a** command because you want to step through it and see what it does. Commands typed in the screen mode are not followed by RETURN and are not echoed to the screen.

Execution jumps over the elaboration of TEST_DATE and is now ready to call TEST_DATE on line 128. Remember, the * inside the source window tells you where execution is located. Next, you will step into TEST_DATE by typing **s**.

Notice that the * on the dashed line is now in column 1. This * bounces back and forth between column 1 and column 2 with every command. This movement is useful

because it tells you something is happening even though there is no other indication. For example, if a small loop is contained entirely in the source window and you have a breakpoint set in the loop, g will cause execution to proceed around the loop one time, stopping at the same breakpoint. The screen contents would be identical, except that the * on the dashed line changes columns.

```
121         put ('/');
122         put(day.year - 1900);
123         new_line;
124      end;
125   begin
126      -- first test some good dates.
127
128*     test_date("February 21,1985");
129      test_date("December 31,1900");
130      test_date("January 1,1999");
131
132      -- now test it with errors
133
134      test_date("Aug. 1, 1981"); -- bad month
135      test_date(March 7 1983"); -- no comma
*-----------------------------------------------------------stk1.a--
VADS_library: /bc/code
library search list:
   /bc/code
   /bc/vads/standard
:vi
:set tabs 4
stk
```

You type s and see that the * is now at line 105 (marked here for illustration purposes with a 1 to the left of the line number).

Now you type s again to move the * from line 105 to line 108 (marked with 2). It will also cause the star on the dashed line to change columns.

You type s again to step into GET_DATE_REP.

```
   98  with text_io;
   99  package iio is new text_io.integer_io(integer);
  100
  101  with date_mgt; use date_mgt;
  102  with text_io; use text_io;
  103  with iio; use iio;
  104  procedure stk is
1 105*    procedure test_date(date: string) is
  106        day: date_rep;
  107     begin)
2 108*       day := get_date_rep(date);
  109        put('"');
  110        put(date);
```

```
  111         put("""" --");
  112
-*-------------------------------------------------stk1.a--
VADS_library: /bc/code
library search list:
   /bc/code
   /bc/vads/standard
:vi
:set tabs 4
stk
```

You are now inside the procedure being tested. You step a few more times to see what happens. You see the * move down the screen, stopping at the declarations that caused code to be generated by the Ada compiler (at the numbers 1, 2, and 3 on the left).

```
  38          if month = month_tab(i).all then
  39             return i;
  40          end if;
  41       end loop;
  42       return 0;
  43    end;
  44
1 45*  function get_date_rep(date: string) return date_rep is
  46       type adate is access date_rep;
2 47*     d: adate := new date_rep'(0,0,0);
  48       the_month: integer;
  49
3 50*     next: integer := date'first;
  51    begin
  52
-*-------------------------------------------------stk1.a--
VADS-library: /bc/code
library search list:
   /bc/code
   /bc/vads/standard
:vi
:set tabs 4
stk
```

That last step (s) put the * on line 53. You see that this is a loop and decide not to single step through it. Instead, you use the **j** key (or down-arrow) to move down further in the source.

As the cursor moves down, the source window scrolls up. When you get to the next statement after the loop (line 57), you set a breakpoint there by pressing **b**. In screen mode, the debugger shows lines with breakpoints by putting an = next to the line number.

To skip over the loop, use the **g** command.

```
  43   end;
  44
  45   function get_date_rep(date: string) return date_rep is
  46      type adate is access date_rep;
  47      d: adate := new date_rep'(0,0,0);
  48      the_month: integer;
  49
  50      next: integer :=date'first;
  51   begin
  52
  53*     while (date(next) /= ' ') and (next <= date'last) loop
  54         next := next + 1;
  55      end loop;
  56
  57=     if next > date'last then
-*-------------------------------------------------------stk1.a--
VADS-library: /bc/code
library search list
   /bc/code
   /bc/vads/standard
:vi
:set tabs 4
stk
```

Notice now that both the * and the = are on the same line, indicating that you have stopped execution at a breakpoint.

You now use the **j** (or down arrow) key again to scroll more source text into the window. When you do, you see that the IF statement is to detect an error. You step over the IF using **s**. If an error is detected, you'll step to statement 58; if not, you'll step to statement 64, the next executable statement after the IF.

```
  43   end;
  44
  45   function get_date_rep(date: string) return date_rep is
  46      type adate is access date_rep;
  47      d: adate := new date_rep'(0,0,0);
  48      the_month: integer;
  59
  50      next: integer := date'first;
  51   begin
  52
  53      while (date(next) /= ' ') and (next <= date'last) loop
  54         next := next + 1;
  55      end loop;
  56
  57*=    if next > date'last then
*---------------------------------------------------------stk1.a--
VADS-library: /bc/code
library search list;
   /bc/code
   /bc/vads/standard
```

```
:vi
:set tabs 4
stk
```

Made it to 64!

You decide not to step into the procedure MONTH_NO, so you use **a**, which single steps one source line, but steps over procedure calls. Now you are at line 65. If THE_MONTH is 0, you have an error.

You move the cursor on top of the **t** of THE_MONTH and use **p**. This causes the name of the variable and its value to be displayed in the bottom window. THE_MONTH is zero, so something went wrong in MONTH_NO.

```
  57=     if next > date'last then
  58        return d.all;                    --error exit
  59      end if;
  60
  61      -- We postpone assigning the month into d.month
  62      -- since 0 indicates an error and it is 0 now.
  63
1 64*     the_month := month_no(date(date'first..next-1));
2 65*     if the_month = 0 then
  66        return d.all;                    --error exit
  67      end if;
  68
  69      -- get the day
  70      get_number(date(next..date'last), d.day, next);
  71      if next = - then
-*-------------------------------------------------------stk1.a---
   /bc/code
   /bc/vads/standard
:vi
:set tabs 4
stk
the_month
0
```

You type in **cs** to display the current call stack. This will also display the procedure parameters, so you can make sure that the input to the procedure is what you expected it to be.

The parameter looks fine. You decide that the easiest thing to do is set a breakpoint at line 64 and rerun the program. You don't need to step through the beginning any more because you know the error is in the call to MONTH_NO.

```
  57=     if next > date'last then
  58        return d.all;                    -- error exit
  59      end if;
  60
  61      -- We postpone assigning the month into d.month since 0
```

```
  62      -- indicates an error and it is 0 now.
  63
  64      the_month := month_no(date(date'first..next-1));
  65*     if the_month = then
  66         return d.all;                    -- error exit
  67      end if;
  68
  69      -- get the day
  70      get_number(date(next..date'last), d.day, next);
  71      if next = 0 then
*-------------------------------------------------------------stk1.a--
stk
the_month
0
 # line  procedure              params
 1   65  get_date_rep (date="February 21, 1985")
 2  108  test_date (date="February 21, 1985")
 3  128  stk ()
```

You move the cursor up to line 64 and use **b** to set a breakpoint.

Now you use **r** to rerun the program. This is the only one of the execution commands that will work under these circumstances.

Notice that 'stk' is printed in the command window. Every time a program is run from the beginning the debugger displays the invocation of the program, which in this case is just its name, **stk**.

```
  57=     if next > date'last then
  58         return d.all;                    -- error exit
  59      end if;
  60
  61      -- We postpone assigning the month into d.month
  62      -- since 0 indicates an error and it is 0 now.
  63
  64=     the_month := month_no(date(date'first..next-1));
  65*     if the_month = 0 then
  66         return d.all;                    -- error exit
  67      end if;
  68
  69      -- get the day
  70      get_number(date(next..date'last), d.day,next);
  71      if next = 0 then
-*-------------------------------------------------------------stk1.a--
    /bc/code
    /bc/vads/standard
:vi
:set tabs 4
stk
the_month
0
```

As you can see, you reached the breakpoint set at line 57. Since you are here, you might as well delete this one since it will only get in your way.

```
   57*=    if next > date'last then
   58         return d.all;                -- error exit
   59      end if;
   60
   61      -- We postpone assigning the month into d.month
   62      -- since 0 indicates an error and it is 0 now.
   63
   64=     the_month := month_no(date(date'first..next-1));
   65      if the_month = 0 then
   66         return d.all;                -- error exit
   67      end if;
   68
   69      -- get the day
   70      get_number(date(next..date'last), d.day,next);
   71      if next = 0 then
*-------------------------------------------------------------stk1.a--
   /bc/vads/standard
:vi
:set tabs 4
stk
the_month
0
stk
```

You use the **d** key to delete the breakpoint on the line where the cursor is positioned.

After deleting the breakpoint, use **g** to go to the breakpoint you are interested in. You single step into the MONTH_NO function by typing **s**.

```
1 57*     if next > date'last then
  58         return d.all;                -- error exit
  59      end if;
  60
  61      -- We postpone assigning the month into d.month,
  62      -- since 0 indicates an error and it is 0 now.
  63
  64*=    the_month := month_no(date(date'first..next-1));
2 65      if the_month = 0 then
  66         return d.all;                -- error exit
  67      end if;
  68
  69      -- get the day
  70      get_number(date(next..date'last), d.day,next);
  71      if next = 0 then
*-------------------------------------------------------------stk1.a--
   /bc/vads/standard
:vi
:set tabs 4
stk
```

```
the_month
0
stk
```

You decide to take a quick look at the parameter by typing **1cs**. You could also move the cursor over on top of the variable MONTH and press **p**. (Either method will work to display the value of the parameter.)

```
  28          9 => new string'("september"),
  29         10 => new string'("october"),
  30         11 => new string'("november"),
  31         12 => new string'("december"));
  32
  33      procedure get_number(src: string; val, pos: out integer)
  34         is separate;
  35
  36*     function month_no(month: string) return integer is
  37      begin
  38         for i in month_tab'range loop
  39            if month = month_tab(i).all then
  40               return i;
  41            end if;
  42         end loop;
  43         return 0;
*-------------------------------------------------------------stkl.a--
the_month
0
stk
 # line  procedure        params
 1   35  month_no (month="February")
month
"February"
```

The value looks OK.

You step through the loop that compares the parameter to the table of months.

To look at the first table entry, you use the **p** command, because it is a complex expression. You type **:** and the debugger moves the cursor down into the command window and prompts you with a colon. You then type the **p** command. Notice that if you just say

```
p month_tab(i)
```

i would just get the value of the access type, not the value of the object. To get the value of the object you must use **.all**.

'january' looks ok. Of course it doesn't match 'February'. So you step through the loop again, this time expecting a match.

```
  28          9 => new string'("september"),
  29         10 => new string'("october"),
```

```
  30          11 => new string'("november"),
  31          12 => new string'("december"));
  32
  33       procedure get_number(src: string; val, pos: out integer)
  34          is separate;
  35
  36       function month_no(month: string) return integer is
  37       begin
1 38*         for i in month_tab'range loop
2 39*            if month = month_tab(i).all then
  40               return i;
3 41*            end if;
  42          end loop;
  43          return 0;
*-------------------------------------------------------------stk1.a--
stk
 # line  procedure        params
 1   35  month_no (month="February")
month
"February"
:p month_tab(i).all
"january"
```

But you don't get a match.

You use the p command again to print the value in the table.

AHA! You used a 'F' on input, but the table has a 'f'. You can see that you need a procedure that will 'lower case' the input string so that this will not be a bug. You write that procedure and try again.

```
  28           9 => new string'("september"),
  29          10 => new string'("october"),
  30          11 => new string'("november"),
  31          12 => new string'("december"));
  32
  33       procedure get_number(src: string; val, pos: out integer)
  34          is separate;
  35
  36       function month_no(month: string) return integer is
  37       begin
1 38*         for i in month_tab'range loop
2 39*            if month = month_tab(i).all then
  40               return i;
3 41*            end if;
  42          end loop;
  43          return 0;
-*-------------------------------------------------------------stk1.a--
1   35  month_no (month="February")
month
"February"
:p month_tab(i).all
```

```
"january"
:p month_tab(i).all
"february"
```

You put in a LOWER_CASE procedure and run the program. It still fails. You follow basically the same procedure as above, stepping down into MONTH_NO. Before doing the comparison with 'February', you check to see what your new LOWER_CASE procedure returned.

It looks like you did something wrong — the procedure is converting to upper case.

```
29          10 => new string'("october"),
30          11 => new string'("november"),
31          12 => new string'("december"));
32
33      procedure get_number (src: string; val, pos: out integer)
34          is separate;
35      procedure lower_case (str_in: string; str_out: out string)
36          is separate;
37
38      function month_no(month: string) return integer is
39          lc_month: string(month'range);
40      begin
41          lower_case(month, lc_month)
42*         for i in month_tab'range loop
43              if lc_month = month_tab(i).all then
44                  return i;
-*-----------------------------------------------------------stk1.a--
:set tabs 4
stk
the_month
0
stk
lc_month
"FEBRUARY"
```

You go back, fix it up and run the program again. This time, you are confident the program will work, so you run it outside of the debugger. The output is shown below.

```
"February 21, 1985"          --  2/ 21/ 85
"December 31, 1900"          -- 12/ 31/  0
"January 1, 1999"            --  1/  1/ 99
"Aug. 1, 1981" -- ERROR
**MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error" RAISED
```

You're making progress, but there is still a problem.

Unhandled exceptions are easy to figure out with the debugger. All you have to do is get into the debugger and run the program. It will stop at the point of the unhandled exception and show where the problem is.

The program runs until the exception occurs. The output of the program is displayed in the command window as the program runs, the last line is still visible below the dashed line.

The diagnostic that shows what went wrong is below. It appears that NEXT is out of range.

```
  73  get_number(date(next..date'last), d.day, next);
  74  if next = 0 then
  75     return d.all; --error exit;
  76  end if;
  77
  78  -- find the comma
  79  next := next + 1;
 80* while (date(next) /= ',') and (next <= date'last) loop
  81     next := next + 1;
  82  end loop;
  83
  84  -- make sure there is more string left to hold the year
  85  next := next + 1;
  86  if next > date'last then
  87     return d.all; -- error exit;
-*-------------------------------------------------------------stk.a--
"Aug. 1, 1981"    -- ERROR
**MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error" RAISED
 (program is now stopped on line where exception was raised)
 stopped 12 instructions after "./stk.a":80 in get_date_rep
 80    while (date(next) /= ',') and (next <= date'last) loop
    ---------^
=> RM 4.1.1(4): index value greater than upper bound of index
   subtype
```

Display some variables: NEXT is 13, and DATE is also shown.

```
  73  get_number(date(next..date'last), d.day, next);
  74  if next = 0 then
  75     return d.all; --error exit;
  76  end if;
  77
  78  -- find the comma
  79  next := next + 1;
 80* while (date(next) /= ',') and (next <= date'last) loop
  81     next := next + 1;
  82  end loop;
  83
  84  -- make sure there is more string left to hold the year
  85  next := next + 1;
  86  if next > date'last then
  87     return d.all; -- error exit;
-*-------------------------------------------------------------stk.a--
  80    while (date(next) /= ',') and (next <= date'last) loop
     ---------^
```

```
=> RM 4.1.1(4): index value greater than uper bound of index subtype
next
13
date
"March 7 1983"
```

AHA, there's no comma in DATE, and the purpose of the loop is to find a comma. You figure that you should switch the two clauses of the WHILE loop so that the second clause is tested first.

You change line 80 to read

```
while (next <= date'last) and (date(next) /= ',') loop
```

to test the out-of-bounds condition first and not do the subscripting if NEXT is too big.

You do this and rerun the program. You get an identical result: unhandled exception. This has you quite mystified (and a little disappointed) so you run it with the debugger again.

It seems that the program evaluates both sides of the AND even if the left side is false. You recall that Ada supplies a specific short circuit AND operator called AND THEN. You replace AND with AND THEN.

```
while (next <= date'last) and then (date(next) /= ',') loop
```

You recompile and run the program before going back in the debugger. This time it works!

```
"February 21, 1985" -- 2/ 21/ 85
"December 31, 1900" -- 12/ 31/ 0
"January 1, 1999" -- 1/ 1/ 99
"Aug. 1, 1981"    --  ERROR
"March 7 1983" -- ERROR
"July 3" -- ERROR
"June 3," -- ERROR
```

CHAPTER 16

APPLICATION NOTES

16.1 BENCHMARKING

Many customers wish to benchmark VADS along various metrics. This section discusses some of the common misconceptions about benchmarking.

16.2 BENCHMARKING COMPILATION SPEED

The most common mistake in benchmarking VADS compilation speed is to start by doing a **a.cleanlib**, cleaning out the directory. VADS makes a number of optimizations based on the assumption that real use of VADS will involve more than one compilation of many source files in an Ada library. Compile the source files a second time and measure the time on the second compilation. This can be done quite simply.

```
a.make -v -f *.a
rm .objects/*
a.make -v -f *a
```

Using the commands above, there is extra overhead in the program generator tool **a.make** that may effect compilation timings. This overhead can be eliminated if **a.make** is used to establish the recompilation order separately from the compilations themselves:

```
a.make -v -f *.a > log.list
```

Edit the file **log.list** to produce a shell script that compiles each file in order with the command

```
ada -T -v ada_source.a > timing_file_name
```

Several constructs are especially slow for VADS as currently implemented:

- generics whose bodies do not share object code
- inline expansions of subprograms
- full optimization (-O9)
- real number literals
- statically initialized large arrays
- many derived types that cause extensive overloading
- very large source files

In addition, system load can increase both wall clock time and measured CPU time.

16.3 BENCHMARKING RUNTIME PERFORMANCE

The most common mistake affecting runtime benchmark performance is the use of 'standard' Ada types. The Ada reference manual does not constrain implementations in their choice of representation for these types, therefore programs that use them are *not portable.*

VADS implements type STANDARD.INTEGER as the 'natural' integer supported on the machine: i.e., the type supported by its registers. On a SUN machine registers are 32 bits wide. Some other Ada compiler vendors have chosen 16 bits for type INTEGER, achieving somewhat better performance. To write portable benchmark programs, use size clauses for types.

```
type integer_as_32bits is -(2**31) .. (2**31)-1;
   for integer_as_32bits'size use 32;
```

Similarly, VADS supports IEEE standard floating point, and so SHORT_FLOAT is 32 bits, FLOAT is 64 bits, and LONG_FLOAT is any longer floating point type supported by the architecture. Many other Ada vendors have chosen to equate type FLOAT with the 32 bit type. This discrepancy has major performance impacts on floating point benchmarks.

When benchmarking performance of arrays between C and Ada, be aware that in C the lower bound of dynamic arrays is always 0. For a fair comparison, define Ada array types with a fixed lower bound also, as many optimizations are then possible.

The current implementation of CALENDAR.TIME is portable but not configurable. As a result, benchmarking using package CALENDAR is slow. To prevent its influence on benchmarking time, it should be called outside the program being timed, once before executing and again afterward.

When time slicing is enabled, tasks are preemptable. When a task is preempted, the RTS must save the states of the floating point coprocessor registers, a slow operation on some processors. This can affect tasking benchmarks adversely.

CHAPTER 17

INSTALLATION GUIDE

17.1 OVERVIEW

This document includes information for system administrators on installing and maintaining a single or multiple versions of VADS.

A new section is included on the use of VADS' automated customer reporting tool, **a.report**.

17.2 INSTALLATION

VADS can be installed anywhere within the file system and multiple compilers (bearing release number Version 5.41 or greater) may co-exist and be in simultaneous use on the same machine. A step-by-step installation procedure is described below, but the alternatives are so numerous that it is not possible to enumerate each possible variation. Instead, a list of requirements is provided that should allow the experienced system administrator in collaboration with the VADS administrator, if any, to make the best choices for a particular environment.

17.2.1 Installation Requirements

1. VADS requires a single reference file, **/etc/VADS** to be present on the system. This file contains information about each version of VADS present. The user installing any of the VADS versions must be able to modify the file **/etc/VADS**.

2. Various versions of VADS need not be placed in any particular relation to one another, although maintaining several versions of compilers on a system is generally simplified by placing them in a single directory tree.

3. If it is likely that the system administrator and the person maintaining VADS may not be the same individual, a separate user login should be added to the system for installing and administering VADS. He should have write access to the file **/etc/VADS**.

4. All versions of VADS on a single system should be owned by the same user.

5. It is necessary for a user of VADS to place only a single VADS **bin** directory (for releases numbered 5.41 and greater) on the path; reference files in the libraries in which Ada programs are compiled will cause execution of the compiler and tools corresponding with the desired VADS version.

6. On systems with multiple versions of VADS (Versions 5.41 or greater), the disk storage required may be somewhat reduced by eliminating tools that are identical across multiple versions. Care must be exercised in so doing because only some of the tools are common among VADS implementations. Suggestions for reclaiming space are contained at the end of the installation section.

7. The swap space needed by VADS ranges from a minimum of 5MB to a maximum of 17MB.

17.2.2 Typical VADS Installation

The following steps are for a typical VADS installation. Some explanation is given where choices need to be made. If greater changes in the installation procedure are desired, they should be carefully planned using the rules previously given. *VADS_location* is used below for the name of the directory created for the current version of VADS being installed.

1. Preliminary set up.

   If this is the first installation of a VADS release numbered 5.41 or greater, create or determine the user account from which to perform the installation, and create the empty file /etc/VADS. Make certain that the user who will install VADS has write permission on the file.

2. Create a directory for a particular version of VADS.

   Each separate VADS compiler on a system must reside in a separate directory. Do *not* place more than one VADS version in the same directory. It is usually convenient to create a separate subdirectory for each one under the same parent location.

   ```
   mkdir VADS_location
   chmod 755 VADS_location
   ```

3. Become the installation user.

   Either login as the designated user, or **su** to it.

4. Change directory to the directory just created.

   ```
   cd VADS_location
   ```

5. Extract the files from the distribution tape.

   Depending on the type of media (cartridge or floppy diskettes) supplied, execute *one* of the commands — either 5a or 5b–below.

   5a. Cartridge tape:

   ```
   cpio -icBdm < tape_device_name
   ```

   5b. Floppy diskettes:

   1) Insert diskette #1 into the drive and execute the command:

   ```
   cpio -icBdm < floppy_device_name
   ```

   2) Execute the command:

   ```
   ./sup/a.cpio
   ```

   This initiates a script that prompts the installer to insert the rest of the set of diskettes into the drive and extracts the files from those diskettes using CPIO(1).

   Note that files from any single diskette can be extracted with the command:

```
cpio -icBdm < tape_device_name
```

6. Run the installation script.

   The script sets up the correct pathnames for the supplied libraries and other files and fills and edits the file **/etc/VADS**. The only choice required of the user is to edit (using **vi**) the file **/etc/VADS** to rearrange the lines representing each VADS compiler on the system. To allow the debugging of C programs with the VADS debugger, the most recently installed version of VADS must appear first. Installers unfamiliar with **vi** can simply exit the program with **:wq** and edit the file **/etc/VADS** with another editor after the installation script has completed successfully.

   ```
   sup/a.install
   ```

   The installation script will test the compiler by compiling a test program. If the installation is unsuccessful, please record the messages produced and call APSO for assistance.

7. Configure the **a.report** files:

   ```
   cd VADS_location/sup
   ```

   Now use an editor to modify the file **a.report.info**. The editor **vi** is shown in the command below, but any editor may be used.

   ```
   vi a.report.info
   ```

The file now looks as shown below.

```
# writeseq is a file writeable by everyone
# (mode 666) to store the report sequence number.
# mailer is the system mail command and subject option.
# mailto specifies the people to whom the Deficiency
# Report is mailed (locally and at Verdix if using
# electronic mail).
#
writeseq="VADS_location/sup/a.report.seq"
mailer="Mail -s"
mailto=""
```

The file should be edited as follows. The quotes after **mailto** should contain the login name of the person responsible for mailing deficiency reports to Verdix Ada Product Support Office (APSO). This will generally be either the system administrator or some other person who is the point of contact with Verdix. APSO can be reached via e-mail at *uunet!vrdxhq!apso*.

The quotes after **writeseq** will normally contain the *VADS_location* directory with **/sup/a.report.seq** appended. If for some reason the file is not writeable by everyone in this location, the file **a.report.seq** itself can be moved to some other location and this line adjusted accordingly.

8. Notify users of the correct path to use.

   The location of the new **bin** directory should be conveyed to VADS users, so that they can change their local **.profile** file. VADS users should always place *only* the *newest* release's **bin** directory on their UNIX **$PATH** variable, no matter which VADS version they are executing.

9. Conserve disk space.

   If multiple versions of VADS 5.41 or greater are installed on the same system, only one of the **bin** directories needs to be placed on the user's **$PATH** to use all these versions. Several of the tools supplied are duplicated in each release so the **bin** directories of those VADS versions not placed directly on a user's path may be carefully removed.

10. Change to a new release of VADS.

    This VADS release uses net files incompatible with previous versions. To recompile groups of program libraries, perform the following steps.

    Execute **a.cleanlib** in each VADS library and use a script calling **a.path** or **sed**(1) to modify the library search list in each library's **ada.lib** file to the new locations of **standard**, **verdixlib**, and **publiclib**.

    Execute **a.make** in each of the user libraries with the **-f** option as shown below.

```
a.make -f *.a
```

17.3 CUSTOMER REPORTS

VADS customers can contract with Verdix for two levels of product support, which includes tracking of reported problems with VADS by Verdix's Ada Product Support Office (APSO). The tool **a.report** is provided for customers to use to submit these Customer Reports to APSO (**see** *VADS COMMAND REFERENCE,* **a.report**, page 19-29).

a.report automatically captures the date, time, site name, user, and VADS configuration information. It then prompts for the name of source files illustrating the problem to be appended to the report and invokes **vi** or the user's default editor to allow editing of general text that describes the problem. Report generation can be aborted at any point using the INTERRUPT key.

a.report assigns a number to each report to aid the customer in tracking reports originating from a given host. This number is not used by Verdix, and the numbers may be preset by the system administrator to meet local needs.

When a report has been completed, the entire file is electronically mailed to persons determined by the system administrator when VADS was installed. This usually includes the site Ada administrator. The user can add additional mail destinations and/or save a copy of the report by supplying a file name when prompted. Mail directly to Verdix's Ada Product Support Office (APSO) at *uunet!vrdxhq!apso*.

A warning is displayed if the report length is greater than that acceptable by most network mailers. If the warning appears, the user should be certain that there are no intervening systems between Verdix and the report site. If there are, the report should be saved in a file and transmitted to APSO via magnetic tape.

17.3.1 Submitting a Customer Report to APSO

Two methods can be used to submit the reports generated by **a.report** to APSO: electronic mail and magnetic tape. If absolutely necessary, a user can submit a hard-copy Customer Report to APSO; however, manual processing of these reports will generally be slower.

Electronic Mail — Forwarding the file generated by **a.report** to APSO by electronic mail is the preferred method. APSO's electronic mail is received and transmitted by means of a dial-up network between UNIX and other systems using UUCP protocols. Verdix can establish direct dial-up connections to all customers with maintenance or support contracts. The system administrator for sites using UUCP should contact APSO for more information. For Level II support customers, Verdix can provide immediate dial back and nightly polling services to speed communication and minimize customer telephone costs.

Magnetic Tape — Incompatible operating systems or extremely large source code files can make electronic mail impossible, and in these cases a magnetic tape containing the file generated by **a.report** and other files related to the deficiency can be mailed to APSO. The tape should be labeled clearly, indicating its originator, content, and format. A cover letter outlining the nature of the deficiency and giving a point of contact familiar with the problem should be included. If the tape is to be returned, also indicate the name and mail stop of the recipient.

Verdix can accept a variety of media and formats on tape. For UNIX systems, the preferred format is UNIX **tar**(1); the media may be 1/2-inch 9-track tape reels or 1/4-inch data cartridges. All 1/2-inch tapes can be read at 1600bpi or 6250bpi.

Verdix can also accept, if necessary, ANSI standard 1/2-inch 9-track tapes generated by UNIX **ansitar**, or UNIX **tar**(1) tapes on TK50 cartridges. In each of these cases, however, short file names and no file hierarchy should be used because the recovery mechanisms available to APSO will truncate file names and lose any hierarchy information.

17.4 CONFIGURING THE VADS RUNTIME SYSTEM

The only configuration that a user can perform is to change the rate of time slicing and the range of priorities that are time sliced. Time slicing can be enabled for all task priorities below a given priority or can be turned off completely.

To change the default values for these parameters, the file **config.a** in *VADS_location*/**user_config** should be copied to a VADS library, edited, and compiled with new values. It must be explicitly WITH'd in the user's program source or linked by means of a WITH*n* directive in the **ada.lib**.

The program source for **package** CONFIG is shown below.

```
package CONFIG is

  STACK_SIZE              : constant INTEGER := 16#10_000#;
  pragma EXTERNAL_NAME(STACK_SIZE, "STACK_SIZE");

  KRN_STACK_SIZE          : constant INTEGER := 4000;
  pragma EXTERNAL_NAME(KRN_STACK_SIZE, "KRN_STACK_SIZE");

  INTR_STACK_SIZE         : constant INTEGER := 4000;
  pragma EXTERNAL_NAME(INTR_STACK_SIZE, "INTR_STACK_SIZE");

  TIME_SLICING_ENABLED    : BOOLEAN          := TRUE;
  pragma EXTERNAL_NAME(TIME_SLICING_ENABLED,
   "TIME_SLICING_ENABLED");

  TIME_SLICE_INTERVAL     : INTEGER          :=-- milliseconds
   1000;
  pragma EXTERNAL_NAME(TIME_SLICE_INTERVAL,
   "TIME_SLICE_INTERVAL");

  -- If this is true, the Ada program will use the UNIX SIGFPE
  -- signal to catch numeric errors.  If it is false, numeric
  -- errors will not be caught as per the Ada RM and the Ada
  -- program will not affect SIGFPE.  This is useful when calling
  -- foreign packages that use or ignore SIGFPE.
  NUMERIC_SIGNAL_ENABLED : BOOLEAN           := TRUE;
  pragma EXTERNAL_NAME(NUMERIC_SIGNAL_ENABLED,
   "NUMERIC_SIGNAL_ENABLED");

  -- Like 'numeric_..' above, except for catching storage
  -- violations with SIGSEGV and generating storage_error.
  STORAGE_SIGNAL_ENABLED : BOOLEAN           := TRUE;
  pragma EXTERNAL_NAME(STORAGE_SIGNAL_ENABLED,
   "STORAGE_SIGNAL_ENABLED");

  -- enable hardware floating point exceptions (like mc68881
  -- coprocessor)
  HW_FLT_EXCEPT_ENABLED  : BOOLEAN           := TRUE;
  pragma EXTERNAL_NAME(HW_FLT_EXCEPT_ENABLED,
   "HW_FLT_EXCEPT_ENABLED");
end CONFIG;
```

CHAPTER 18

IMPLEMENTATION REFERENCE (INCLUDING THE REQUIRED MATERIAL FROM THE ADA REFERENCE MANUAL, APPENDIX F)

18.1 INTRODUCTION

This document summarizes the tools and implementation of VADS. Information of a more general nature applying to all VADS implementations is presented first, followed by information specific to this implementation. The material required by Appendix F of the *Ada Reference Manual* (RM) is covered.

VADS Ada provides the full Ada language as specified in the *Ada Reference Manual* (RM). Within the RM, a number of sections contain the annotation *implementation dependent*, meaning that the interpretation is left to the compiler implementor. Verdix has attempted to make those choices that provide the programmer with an essentially unlimited capability to program in Ada. Consequently, an applications programmer can usually program in Ada according to the RM and good engineering practices without consideration of any VADS specifics.

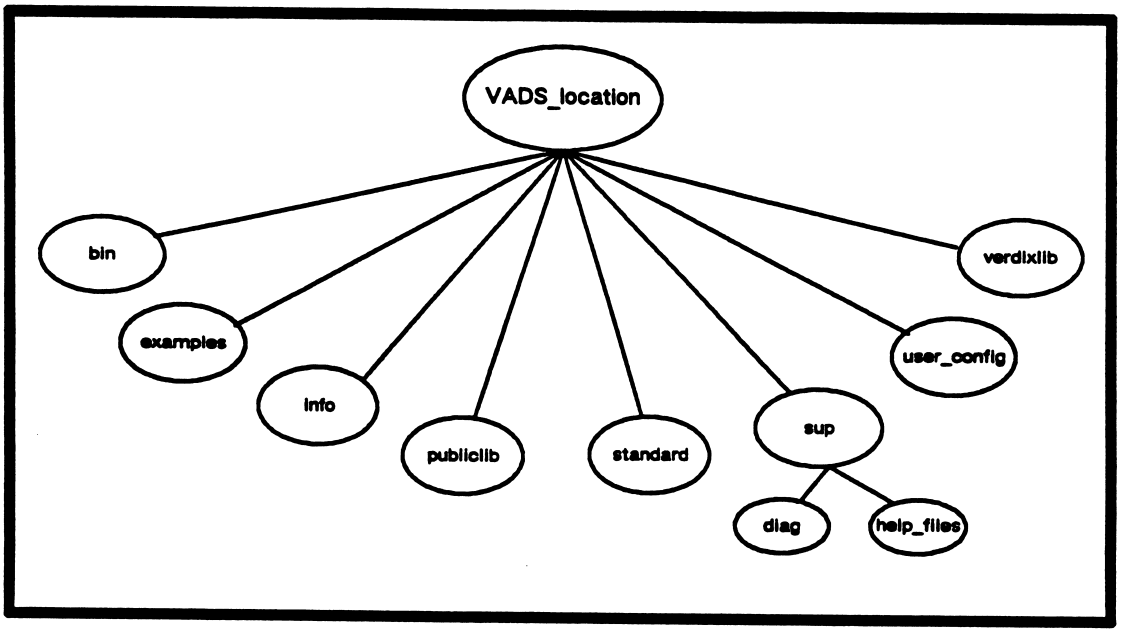
Each successive release of the VADS compiler has provided more features or extended those already present, including the following.

- shared **generic** bodies
- all-Ada runtime system
- representation clauses to the bit level and **pragma** PACK (RM 13.1)
- length clauses and unsigned types (8- and 16-bit) (RM 13.2)
- enumeration representation clauses (RM 13.3)
- record representation clauses (RM 13.4)
- interrupt entries (RM 13.5.1)
- representation attributes (RM 13.7.2)
- machine code insertions and **pragma** IMPLICIT_CODE (RM 13.8)
- interface programming features, including **pragma** INTERFACE, **pragma** EXTERNAL_NAME, **pragma** INTERFACE_OBJECT, WITH directives, **a.info**, and external dependencies capabilities (RM 13.9)
- unchecked deallocations (RM 13.10.1)
- unchecked conversions (RM 13.10.2)
- pool-based memory allocation option

In addition, the tools provided with the compiler have grown in number and functionality with each release.

18.2 RELEASE DIRECTORY STRUCTURE

The directory structure of a VADS release is outlined below.



bin contains all VADS executables.

examples contains sample programs illustrating Ada language use and demonstrating the capabilities of the language. [The **examples** directory is not warranted or supported by Verdix.]

info contains an info file that provides the user necessary information about this release

publiclib contains public domain packages written in Ada. [The **publiclib** directory is not warranted nor supported by Verdix.]

standard contains the VADS implementation of **package** STANDARD and all other predefined packages and library units. Packages used to support the predefined environment are also included.

sup contains additional executable programs called by other VADS tools, help files used by **a.help** and **a.db**, and installation tools.

user_config contains configuration files for the runtime system.

verdixlib contains mathematical functions, operating system calls, command line arguments, and other programs.

18.2.1 The Standard VADS Library

The **standard** library contains specifications and bodies for predefined Ada packages.

Packages other than those defined in the RM in the directory **standard** support the predefined packages. The following packages are generally supported.

A_STRINGS — implements a set of variable-length string operations based on the type definition:

```
type STRING_REC(len: natural) is record
   s: string(1..len);
end record;
type A_STRING is access STRING_REC
```

This representation is chosen to provide the best opportunity for optimizations in the future, while having convenient reference. Verdix recommends its use for variable-length strings.

CALENDAR — implements the predefined package CALENDAR.

C_STRINGS — implements a set of variable-length string operations based on the type definition:

```
type C_STRING is access string (1..integer'last);
--WARNING: this package mimics the behavior of
--ASCII.nul-terminated strings commonly used in the C
--programming language. It should be used to represent
--strings that are to be passed to and from UNIX utilities,
--for example.
```

CLOSE_ALL — closes all open files. It is generally called by the RTS on program exit. Users may change it to add their own activity to the exit process.

FILE_NAMES — defines a set of utilities for dealing with file names on UNIX. It is portable and its use is recommended for creating portable system utilities. For example, the following code calls a subprogram DO_IT for all files in a directory with names matching the pattern '*str*.a*'.

```
   THIS_DIR: FILE_NAMES.FIND_FILE_INFO :=
      FILE_NAMES.INIT_FIND_FILE (to_A("*str*.a*"));
   FILE: A_STRINGS.A_STRING;
begin
   loop
      file := FILE_NAMES.FIND_FILE(THIS_DIR);
      doit(file);
   end loop;
exception
when FILE_NAMES.NO_MORE_FILES => ...
```

FILE_SUPPORT — supports file activities for Ada standard I/O, This package can be used for lower-level file activities than those defined by Ada's TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO.

HEX — supports INTEGER'IMAGE and INTEGER'VALUE (as functions rather than attributes) but using hexadecimal strings rather than decimal.

IFACE_ALLOC — interfaces to the VADS memory allocation services.

IFACE_BITS — supplies BIT_AND, BIT_OR, BIT_XOR, and BIT_NEG on integers.

IFACE_RAISE — interfaces to the VADS exception support services.

IFACE_SEMA — interfaces to VADS semaphore and critical region services.

IFACE_TASKOP — supplies some control functions over VADS tasks.

IFACE_TASKS — interfaces to the VADS task operation services.

IFACE_TYPES — supplies the types for entities used in the VADS Runtime System. These types are useful because code that uses them will be portable across VADS compilers. They should be used for changes and additions to the runtime.

For further information on the IFACE packages, see also *[IMPLEMENTATION REFERENCE], 18.20 Runtime System*, page 18-37.

LANGUAGE — supplies a language interface for UNIX

MACH_CONFIG — machine configuration definitions and data structures used by the RTS kernel (for example, kernel service interrupt number).

MACHINE_TYPES — supplies definitions of byte (8 bit) and word (16 bit) unsigned types.

MACHINE_CODE — defines machine code statements described in RM 13.8.

See also *[IMPLEMENTATION REFERENCE], 18.18 MACHINE CODE INSERTIONS,* page 18–31.

MEMORY — supplies PEEK, POKE and COPY on untyped memory. This package should be used with extreme caution.

OS_FILES — supplies a machine-independent interface to low-level I/O operations. Users may write programs using OS_FILES if the files contain untyped binary information.

RAW_DUMP — prints regions of memory in hexadecimal representation.

SHARED_IO — triggers generation of shared object code for generic packages in TEXT_IO for the common Ada types. This package generally improves disk usage by ensuring that only one instantiation of these I/O packages exists for each VADS.

STRLEN, STRNCPY — duplicate the corresponding C functions.

SYSTEM — is the predefined Ada package SYSTEM. It may be altered and recompiled as long as the Ada-specific entities are changed only in limited ways.

ERRNO — defines the UNIX errno global variable. When calling UNIX utilities, values defined here will be returned to indicate the status of the requested activity.

IOCTL and IOCTL_FMT — supply an interface to UNIX I/O control functions.

LIBC — supplies an interface to UNIX directory functions.

STATUS — supplies a definition of the UNIX STATUS_BUFFER data type. This type is returned by calls to stat(2) and fstat(2).

TTY and TTY_SIZES — supplies a definition of the UNIX tty(4) data structures.

U_ENV — supplies a definition of the command line argument interface.

UNIX — supplies a direct interface to the most common UNIX system calls.

UNIX_DIRS — supplies a specialized interface for the UNIX directory manipulation utilities.

UNIX_PRCS — supplies a specialized interface to the UNIX process-control utilities.

18.2.2 The verdixlib VADS library

The VADS library **verdixlib** contains Ada packages that provide mathematical functions, operating system calls, and other capabilities to the user. These packages are proprietary to Verdix. Both **verdixlib** and **standard** are automatically on the path of a VADS library created by **a.mklib** (unless a parent library is given), and their packages are thus available to a user's program. Additional information is supplied in the header of each of the package specification files. Files in this directory should not be recompiled in this library by the user.

The following files are contained in **verdixlib**.

File	Package	Purpose
cmd_line_s.a **cmd_line_b.a**	COMMAND_LINE	provides easy access to the command line arguments used to invoke a program. These closely follow the C language conventions and permit the program to have access to environment variables as well
complex_spec.a **complex_body.a**	COMPLEX	provides functions for complex value arithmetic using generic floating types and functions for composition and decomposition of the complex data type
dates.a **dates_b.a**	DATES	provides functions for commonly used manipulations of dates
math_spec.a **math_body.a**	MATH	provides mathematical constants, exponential, logarithmic, circular trigonometric, inverse circular trigonometric, hyperbolic trigonometric, polar conversion, and Bessel functions
ordering_s.a **ordering_b.a**	ORDERING	provides various generic sorting and permutation routines and can be instantiated with a variety of data types

report_spec.a **report_body.a**	ORDERING	provides functions for reporting the pass/fail/not-applicable results of tests.
unixcallspec.a **unixcallbody.a**	UNIX_CALLS	provides Ada language routines for performing many of the most common and useful UNIX system calls

18.2.3 The publiclib VADS library

publiclib contains public domain and other Ada packages not supported by Verdix, but believed to be of programming utility. Complete source code is provided. These currently include packages for character class comparison, manipulating character strings and a curses interface. Additional information is supplied in the headers of the package specification files.

File	Package	Purpose
bit_flg_fix.a	BIT_FLG_FIX	integer to bit field conversions, used with CURSES.
c_printf.a **c_printf_b.a**	C_PRINTF	implementation of C print for converting C programs to Ada
c_to_a_type.a	C_TO_A_TYPE	Ada equivalents for C types.
char_type.a	CHARACTER_TYPE	character class comparisons, like C's **ctype** macros
curses_spec.a **curses_body.a**	CURSES	interface to curses library.
u_rand.a	U_RAND	package implementing a universal random number generator
vstring_spec.a **vstring_body.a**	VSTRING	package implementing variable length strings.

18.2.4 The examples Directory

examples is a directory containing Ada programming examples using the libraries listed above. Each file contains directions on compiling, and linking its program. The files should be copied into a user's VADS library to be compiled.

File	Purpose
arguments.a	tests the COMMAND_LINE interface in **verdixlib**
date.a	show date/time using CALENDAR package from **standard** library
hanoi.a	tower of hanoi with screen oriented display
hello.a	prints Hello, world.
mortgage.a	uses the MATH package from **verdixlib**
permute_list.a	uses the ORDERING package from **verdixlib**
phl.a	a screen oriented implementation of dining philosophers
queens.a	solve the eight queens problems
random.a	use CALENDAR from standard to generate random numbers
slideshow.a	show interface to CURSES in **publiclib**
sort_file.a	a tree-based sort program for text files
sort_ints.a	sort files of integers using TEXT_IO and ORDERING
hanoi.a	tower of hanoi with screen oriented display
termbody.a	
termspec.a	terminal interface, used by HANOI
uctran.a	a calendar reminder program

18.3 ESTIMATING DISK SPACE REQUIREMENTS FOR USING VADS

Disk space is needed for the Ada source code and for several files produced by the VADS compiler. A formula for estimating disk space required for software projects using VADS is

$$N * (V * 500 + P * 50) = \text{bytes}$$

where

N	= lines of Ada source code
V	= number of versions of the source on a given machine
P	= number of programmers

For a single programmer working on a single project with only one complete copy of the source code, VADS is expected to use about 250 bytes of disk storage per line of source.

The largest files are the DIANA nets generated by the VADS compiler. About 100 bytes of net file is produced for an average source line. Nets are relatively larger for specification files, sometimes reaching 300 bytes per line. Nets will also be larger for bodies of generics and inlines, and will produce additional net files or nets for their instantiations. Normal non-generic bodies often run less than 50 bytes per line of source.

The compiler, without optimization on, produces an average of about 20–25 bytes of object file for each line of Ada source. This figure includes generic and inline expansions, and includes the overhead bytes of the object file organization. This figure is typically reduced by about 20% when full optimization is enabled.

The Ada compiler produces small files mapping source lines to object code addresses. These files are typically about 4 bytes per line.

The VADS tools cache DIANA nets from remote libraries to improve the performance of the compiler and debugger. The amount of space used varies with the degree to which the remote libraries change and with the sequence in which the withs requiring remote entities are encountered. Stepping into a remote body with the debugger will also cause an import and possible caching. Such cache files are estimated at about 5–10 bytes per line of local source, although local lines and imported caching are *not* directly related.

The executable size depends on what remotely defined units and libraries are needed. runtime support for tasking and TEXT_IO is not present unless specifically needed.

A 10,000-line Ada program should require about 2.5 megabytes of disk for the source and VADS-generated files.

When several programmers are working on the same program, they will use more disk space to keep local versions of changed entities. Typically, programmers will make changes to 1-10% of a program before updating their changes into the main stream version. The executable for their program will be entirely their own. Roughly speaking, each programmer will need about 20% of the project disk space to maintain these local versions, or roughly 40 bytes per line in the resulting Ada program.

Multiple programmers, machines, and versions lead to estimates like the following. A 100,000-line Ada program, developed on three independent machines with four programmers per machine, keeping one old version and maintaining separate versions for two related products:

$$\text{Disk Needed} = N * (V * 500 + P * 50) = 360 \text{ Megabytes}$$

In this model:

N	=.1 Megabyte
V	= 6 (1 version for each of 3 machines plus 1 old version and 2 related versions)
P	= 12 (4 programmers for each of three machines)

Of the 360 meagabytes, there are 70 megabytes for each of the 3 machines for development plus an additional 50 megabytes for each of the 3 additional versions.

Verdix strongly recommends supplying at least double these 'minimal' disk usage figures because the above estimates assume that disk areas are kept clean of old files, old executables, etc. In addition, the estimate above is only for the Ada program. Space for test suites, documentation, management files, PDL specifications, and so forth is not included.

18.4 PRAGMAS AND THEIR EFFECTS

Each of this implementation's pragmas is briefly described here; additional information on some of them is found under discussions of particular language constructs.

pragma BUILT_IN may be used in some parts of the code for TEXT_IO, MACHINE_CODE, UNCHECKED_CONVERSION, UNCHECKED_DEALLOCATION, and lower level support packages in STANDARD. It is reserved for use by Verdix and is not directly accessible to the user.

pragma CONTROLLED is recognized by the implementation, but has no effect in the current release.

pragma ELABORATE is implemented as described in Appendix B of the RM.

pragma EXTERNAL_NAME allows the user to specify a *link_name* for an Ada variable or subprogram so that the object can be referenced from other languages.

See also *[IMPLEMENTATION REFERENCE], 18.14 INTERFACE PROGRAMMING,* page 18–25.

pragma IMPLICIT_CODE specifies that implicit code generated by the compiler is allowed (ON) or disallowed (OFF) and is used only within the declarative part of a machine code procedure.

See also *[IMPLEMENTATION REFERENCE], 18.18 MACHINE CODE INSERTIONS,* page 18–31 and *MACHINE CODE INSERTIONS, 12.9.3 pragma IMPLICIT_CODE,* page 12–11 for details.

pragma INLINE is implemented as described in Appendix B of the RM with the addition that recursive calls can be expanded with the pragma up to the maximum depth of 8. Warnings are produced for too–deep nestings or for bodies that are not available for inline expansion.

pragma INLINE_ONLY when used in the same way as **pragma** INLINE, indicates to the compiler that the subprogram must *always* be inlined (very important for some code procedures.). This pragma also suppresses the generation of a callable version of the routine which saves code space.

pragma INTERFACE with the parameters (*language, subprogram*), supports calls to C, PASCAL, FORTRAN, and UNCHECKED language functions. The Ada specifications can be either functions or procedures. **pragma** INTERFACE can also be used to call code written in unspecified languages using UNCHECKED for the language name (**see also** *[IMPLEMENTATION REFERENCE], 18.14 INTERFACE PROGRAMMING,* page 18–25).

pragma INTERFACE_NAME with the parameters (*Ada_subprogram_name, "link_name"*) allows variables or subprograms defined in another language to be referenced directly in Ada, replacing all occurrences of *Ada_subprogram_name* with an external reference to *link_name* in the object file (**See also** *[IMPLEMENTATION REFERENCE], 18.14 INTERFACE PROGRAMMING,* page 18–25).

pragma INTERFACE_OBJECT allows variables defined in another language to be referenced directly in Ada, replacing all occurrences of *variable_name* with an external reference to *link_name* in the object file.

See also
[IMPLEMENTATION REFERENCE], 18.14 INTERFACE PROGRAMMING, page 18–25.

pragma LIST is implemented as described in Appendix B of the RM.

pragma MEMORY_SIZE is recognized by the implementation, but has no effect in the current release.

pragma NO_IMAGE suppresses the generation of the image array used for the IMAGE attribute of enumeration types. This eliminates the overhead required to store the array in the executable image.

pragma OPTIMIZE is recognized by the implementation, but has no effect in the current release.

For code optimization options, **see also** *VADS COMMAND REFERENCE,* **ada, -O,** page 19–2.

pragma OPTIMIZE_CODE specifies whether the code should be optimized (ON) by the compiler or not (OFF). It is allowed only within a machine code procedure. When OFF is specified, the compiler will generate the code as specified. The default is OFF.

pragma PACK will cause the compiler to minimize gaps between components in the representation of composite types. For arrays, components will only be packed to bit sizes corresponding to powers of 2 (if the field is smaller than STORAGE_UNIT bits). Objects larger than a single STORAGE_UNIT are packed to the nearest STORAGE_UNIT.

pragma PAGE is implemented as described in Appendix B of the RM. It is also recognized by the source code formatting tool **a.pr.**

pragma PRIORITY is implemented as described in Appendix B of the RM.

pragma SHARE_CODE provides for the sharing of object code between multiple instantiations of the same generic procedure or package body. A 'parent' instantiation is created and subsequent instantiations of the same types can share the parent's object code code, reducing program size and compilation times. The name **pragma** SHARE_BODY may be used instead of SHARE_CODE with the same effect.

pragma SHARED is recognized by the implementation, but has no effect in the current release.

pragma STORAGE_UNIT is recognized by the implementation, but has no effect in the current release. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling **package** SYSTEM with altered values.

pragma SUPPRESS is supported in the single parameter form. The pragma applies from the point of occurrence to the end of the innermost enclosing block. DIVISION_CHECK cannot be suppressed. The double parameter form of the pragma with a name of an object, type, or subtype is recognized, but has no effect in the current release.

pragma SYSTEM_NAME is recognized by the implementation, but has no effect in the current release. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the file **system.a** from the STANDARD library can be copied to a local VADS library and recompiled there with new values.

18.5 IMPLEMENTATION-DEFINED ATTRIBUTE: 'REF

VADS provides one implementation-defined attribute, 'REF. There are two forms of use for this attribute, X'REF and SYSTEM.ADDRESS'REF(N). X'REF is used only in machine code procedures while SYSTEM.ADDRESS'REF(N) can be used anywhere to convert an integer expression to an address.

18.5.1 X'REF

The attribute generates a reference to the entity to which it is applied.

In X'REF, X must be either a constant, variable, procedure, function, or label. The attribute returns a value of the type MACHINE_CODE.OPERAND and may only be used to designate an operand within a code-statement.

The instruction generated by the code-statement in which the attribute occurs may be preceded by additional instructions needed to facilitate the reference (i.e., loading a base register). If the declarative section of the procedure contains **pragma** IMPLICIT_CODE (OFF), a warning will be generated if additional code is required.

References may also cause runtime checks to be generated. **pragma** SUPPRESS may be used to eliminate these checks.

```
CODE_1'(JMP, LABEL'REF);
CODE_2'(MOV, X.ALL(2)'REF, EAX);
```

See also *MACHINE CODE INSERTIONS 12.9 Pragmas,* page 12-10.

18.5.2 SYSTEM.ADDRESS'REF(N)

The effect of this attribute is similar to the effect of an unchecked conversion from integer to address. However, SYSTEM.ADDRESS'REF(N) should be used instead in the following listed circumstances, and in these circumstances, N must be static.

In SYSTEM.ADDRESS'REF(N), SYSTEM.ADDRESS must be the type SYSTEM.ADDRESS. N must be an expression of type UNIVERSAL_INTEGER. The attribute returns a value of type SYSTEM.ADDRESS, which represents the address designated by N.

- within any of the runtime configuration packages
  Use of unchecked conversion within an address clause would require the generation of elaboration code, but the configuration packages are not elaborated.

- in any instance where N is greater than INTEGER'LAST
  Such values are required in address clauses which reference the upper portion of memory. To use unchecked conversion in these instances would require that the expression be given as a negative integer.

- to place an object at an address, use the 'REF attribute
  The *integer_value*, in the example below, is converted to an address for use in the address clause representation specification. The form avoids UNCHECKED_CONVERSION and is also useful for 32-bit unsigned addresses.

```
--place an object at an address
for object use at ADDRESS'REF (integer_value)

--to use unsigned addresses
for VECTOR use at SYSTEM.ADDRESS'REF(16#808000d0#);
TOP_OF_MEMORY : SYSTEM.ADDRESS := SYSTEM.ADDRESS'REF(16#FFFFFFFF#);
```

18.6 RESTRICTIONS ON 'MAIN' PROGRAMS

VADS requires that a 'main' program be a non-generic subprogram that is either a procedure or a function returning an Ada STANDARD.INTEGER (the predefined type). A 'main' program may be neither a generic subprogram nor an instantiation of a generic subprogram.

18.7 GENERIC DECLARATIONS

VADS does not require that a generic declaration and the corresponding body be part of the same compilation, and they are not required to exist in the same VADS library. An error is generated if a single compilation contains two versions of the same unit.

18.8 SHARED OBJECT-CODE FOR GENERIC SUBPROGRAMS

The VADS compiler generates code for a generic instantiation that can be shared by other instantiations of the same generic thus reducing the size of the generated code and increasing compilation speed. There is an overhead associated with the use of shared code instantiations because the generic actual parameters must be accessed indirectly and in the case of a generic package instantiation, declarations in the package are also accessed indirectly. Also, greater optimization is possible for unshared instantiations because exact actual parameters are known. It is the responsibility of the programmer to decide whether space or time is most critical in a specific application.

To give the programmer control of when an instantiation generates unique code or shares code with other similar instantiations, Verdix provides **pragma** SHARE_CODE. This pragma can be applied to a generic declaration or to individual instantiations.

It is not practical to share the code for instantiations of all generics. If the generic has a formal private type parameter the generated code to accommodate an instantiation with an arbitrary actual type would be extremely inefficient.

The VADS compiler will share code by default if the generic formal type parameters are restricted to integer, enumeration, or floating point. To override the default, the **pragma** SHARE_CODE(name, FALSE) must be used. If there are formal subprogram parameters instantiations will not be shared unless an explicit **pragma** SHARE_CODE(name, TRUE) is used.

Generics are shared by default, if a parent is visible, except in the following cases.

- when generic formal types other than integer, enumeration, SYSTEM.ADDRESS or floating point are used
- when pragma INLINE is applied to a generic subprogram or instantiation or to a subprogram visible at the library level within a generic package or instantiation
- when the representations of the actual type parameters are not the same for each of the instantiations
- when the generic has a formal in out parameter and the subtype of the corresponding actual is not the same as the subtype of the formal parameter

The **pragma** SHARE_CODE is used to indicate desire to share or not share an instantiation. The pragma can reference either the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on or off for all instantiations of that generic unless overridden by specific SHARE_CODE pragmas for individual instantiations. When it references an instantiated unit, sharing is on or off only for

that unit. The default is to share all generics that can be shared unless the unit uses **pragma** INLINE.

The **pragma** SHARE_CODE is only allowed in the following places: immediately within a declarative part, immediately within a package specification, or after a library unit in a compilation, but before any subsequent compilation unit. The form of this pragma is

```
pragma SHARE_CODE (generic_name, boolean_literal)
```

Note that a parent instantiation (the instantiation that creates the shareable body) is independent of any individual instantiation, therefore reinstantiation of a generic with different parameters has no effect on other compilations that reference it. The unit that caused compilation of a parent instantiation need not be referenced in any way by subsequent units that share the parent instantiation.

Sharing generics causes a slight execution time penalty because all type attributes must be indirectly referenced (as if an extra calling argument were added). However, it substantially reduces compilation time in most circumstances and reduces program size.

Verdix has compiled a unit, SHARED_IO, in the **standard** library that instantiates all Ada generic I/O packages for the most commonly used base types. Thus, any instantiation of an Ada I/O generic package will share one of the parent instantiation generic bodies unless the following pragma is given:

```
pragma SHARE_CODE ( generic_name, FALSE );
```

18.9 REPRESENTATION SPECIFICATIONS

Representation Clauses — VADS supports bit-level representation clauses.

pragma PACK — Objects and components are packed to the nearest power of two bits. VADS does not define any additional representation pragmas.

Length Clauses — VADS supports all representation clauses.

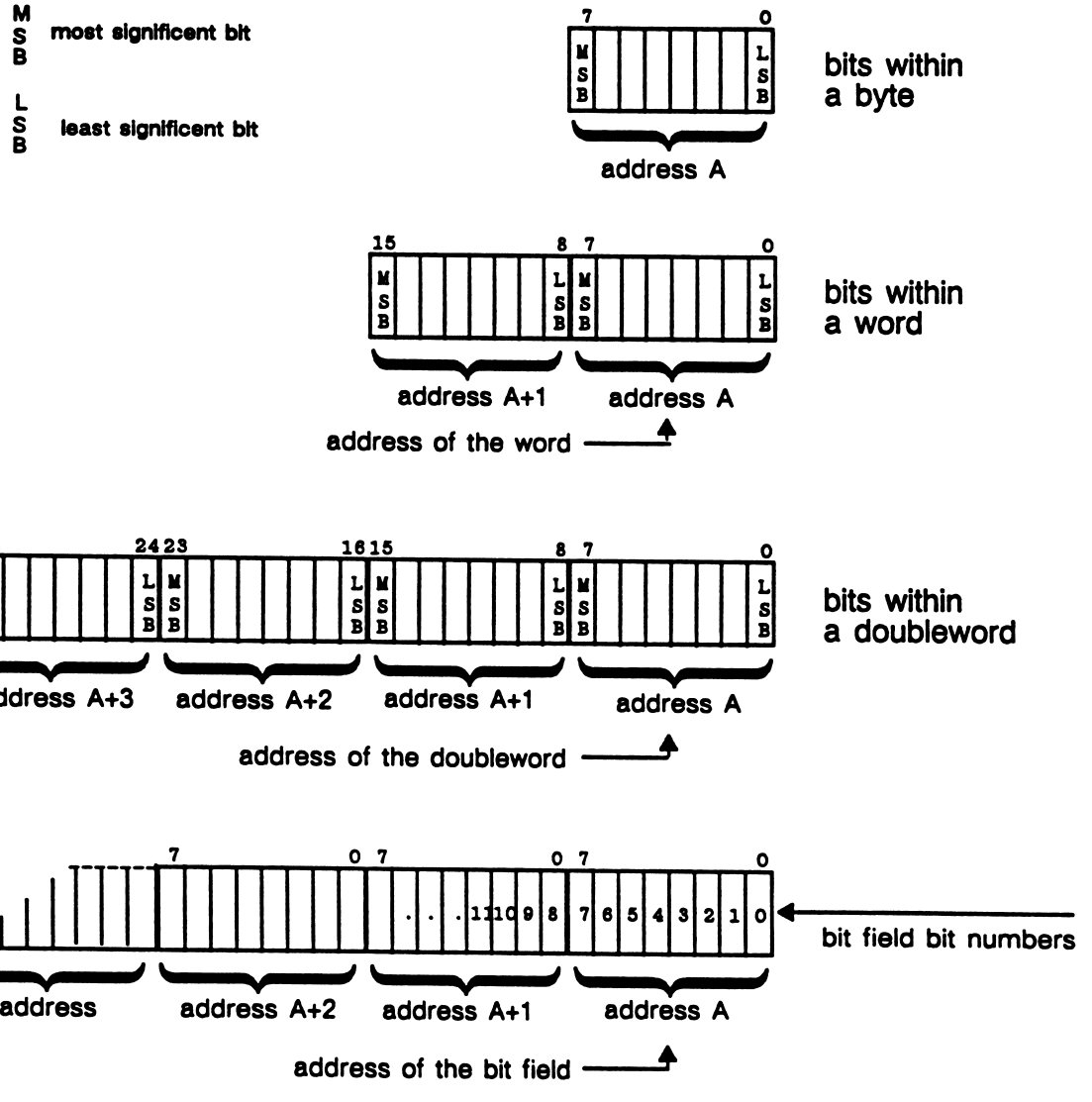
Enumeration Representation Clauses — Enumeration representation clauses are supported.

Record Representation Clauses — Representation specifications are based on the target machine's word, byte, and bit order numbering so that VADS is consistent with machine architecture manuals for both 'big-endian' and 'little-endian' machines. Bits within a STORAGE_UNIT are also numbered according to the target machine manuals. It is not necessary for a user to understand the default layout for records and other aggregates since fine control over the layout is obtained by the use of

record representation specifications. It is then possible to align record fields correctly with structures and other aggregates from other languages by specifying the location of each element explicitly. The 'FIRST_BIT and 'LAST_BIT attributes can be used to construct bit manipulation code that is applicable to differently bit-numbered systems.

Refer to the 80386 addressing and bit numbering illustration on the next page.

80386 ADDRESSING AND BIT NUMBERING*



* *See also 80386 Programmer's Reference Manual,* Section 2.2 Data Types, Figure 2-2, Intel Corporation 1986

The only restrictions on record representation specifications are the following: if a component does not start and end on a storage unit boundary, it must be possible to get it into a register with one move instruction. On the 80386, this means such a field must fit into 4 bytes. For example, the following specification is legal on a 80386 target.

```
for REC use record at mod 2;
  FIELD at 1 range 2 .. 25;
  -- On a 386, extraction can start at byte 1. Byte 1, bit 25 is
  -- 26 bits beyond the start of byte 1 (26 bits <= 4 bytes, so
  -- can extract).
end record;
```

Also, a component that is itself a record must occupy a power of 2 bits. Components that are of a discrete type or packed array can occupy an arbitrary number of bits, subject to the above restrictions.

Note that in the example above a size specification could be given,

```
for REC'size use 39;
```

but due to alignment, such a record would always take 5 bytes (i.e., 40 bits).

Address Clauses — Address clauses are supported for the following entities.

objects
entries

For information on converting INTEGER values to ADDRESS values, **see also** *[IMPLEMENTATION REFERENCE], Implementation-Defined Attributes: 18.5 'REF,* page 18-13, and *18.5.2 SYSTEM.ADDRESS'REF(N)* page 18-14.

Interrupt Entries on UNIX — VADS allows task entries to be associated with UNIX signals. UNIX handles all interrupts and faults initially and then returns control to the user program as a signal.

The available UNIX signals are described in UNIX documentation, **signal**(3C). Some of the signals can not be caught due to restrictions in UNIX. The attempt to assign an entry to these signals results in a kernel error 'replace vector' (**See also** *[IMPLEMENTATION REFERENCE], 18-50 Runtime System Panic Numbers and Messages,* page 18-50*([40] replace vector.)*):

```
#define    SIGKILL     9    /* kill */
```

The Ada Runtime discourages attempts to catch the timer-related signals. The attempt to assign an entry to these signals results in a kernel error 'invalid interrupt to attach' (**See also** *[IMPLEMENTATION REFERENCE], 18.21 Runtime System Panic Numbers and Messages,* page 18-50, *([45] invalid interrupt to attach.)*):

```
#define     SIGALRM     14     /* alarm clock */
```

The following example program shows how a user can attach to the <CONTROL>c or interrupt-from-keyboard signal:

```
with system;     use system;
with text_io;

task interrupt is
    entry SIGINT;
      for SIGINT use at address'ref(2);  -- interrupt
end;
task body interrupt is
begin
    loop
        accept SIGINT do
            text_io.put_line("SIGINT");
        end;
    end loop;
end;
```

Signal handlers are set up for the following signals by the Ada Runtime System:

```
#define     SIGFPE      8     /* floating point exception */
#define     SIGSEGV    11     /* segmentation violation */
```

If a task entry is attached to SIGFPE, numeric_error exceptions will not be raised correctly. If a task entry is attached to SIGSEGV, then storage_error exceptions may not be raised correctly.

On some UNIX machines, the SIGEMT is handled by the Ada Runtime.

```
#define     SIGEMT      7     /* EMT instruction */
```

If a task entry is attached to SIGEMT, the numeric_error exceptions may not be raised correctly.

Use of signal handlers is complicated when non-Ada routines are involved; for further information, see also *Chapter 13, INTERFACE PROGRAMMING*

Change of Representation — Change of representation is supported.

package SYSTEM — For the specification of **package** SYSTEM, **see also** *[IMPLEMENTATION REFERENCE], 18.15.1 Specification of* **package** SYSTEM, page 18.15.1. This specification is also available on line in the file **system.a** in the release **standard** library. The pragmas SYSTEM_NAME, STORAGE_UNIT, and MEMORY_SIZE are recognized by the implementation, but have no effect. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling the SYSTEM package with altered values. Note that such a compilation will cause other units in the STANDARD library to become out of date. Consequently, such recompilations should be made in a library other than **standard**.

System-Dependent Named Numbers — For the specification of **package** SYSTEM, **see also** *[IMPLEMENTATION REFERENCE], Predefined Packages and Generics, 18.15.1 Specification of* **package** SYSTEM, page 18.15.1. This specification is also available on line in the file **system.a** in the release **standard** library.

Representation Attributes — The 'ADDRESS attribute is supported for the following entities.

- variables
- constants
- procedures
- functions

All other representation attributes are supported.

Representation Attributes of Real Types — These attributes are supported. **See** *[IMPLEMENTATION REFERENCE], 18.15 Predefined Packages and Generics,* page 18.15 for their values.

Machine Code Insertions — Machine code insertions are supported. **See** *[IMPLEMENTATION REFERENCE], 18.18 MACHINE CODE INSERTIONS,* page 18-31 and *Chapter 12, MACHINE CODE INSERTIONS* for details.

Interface to Other Languages — **See** *[IMPLEMENTATION REFERENCE], 18.4 PRAGMAS AND THEIR EFFECTS,* page 18-10 and *Chapter 13, INTERFACE PROGRAMMING* for details.

Unchecked Programming — Both UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION are provided.

Unchecked Storage Deallocations — Any object that was allocated may be deallocated. No checks are currently performed on released objects.

Unchecked Type Conversions — The predefined generic function UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

18.10 SOURCE FILE STRUCTURE/RESTRICTIONS

Character Set — VADS provides the full *graphic_character* textual representation for programs. The character set for source files and internal character representations is ASCII.

Lexical Elements, Separators, and Delimiters — VADS Ada uses normal UNIX text files as input.

Compiler/Tool Limits —

- 499 characters in identifiers and literals
- 4,000,000 STORAGE UNITS in a statically sized record type
- 10,240 STORAGE_SIZE default for a task
- 100,000 STORAGE_UNITS default collection size for access type
- no limit number of declared objects (except virtual space)
- 800 characters in a rooted name (full path of an object)
- 8 number of recursive inlines
- 8 number of nested inlines
- 400 number of nested constructs
- 2048 characters in ADAPATH (library search list)
- 2048 characters in a WITH or INFO directive
- 16M memory use per compilation (other UNIX limits may apply)
- 50 lexical errors before the front end exits
- 100 syntax errors before the front end exits
- 10 attempts to lock **GVAS_table**
- 10 attempts to lock **ada.lib**
- 20 attempts to lock **gnrx.lib**
- 64 debugger breakpoints
- 32 debugger array dimensions in a **p** command
- 9 debugger 'call parameters'
- 256 debugger 'run parameters'

Source File Limits —

- 499 characters per source line
- 1296 Ada units per source file
- 32767 lines per source file

18.11 PARAMETER PASSING

Parameters are passed by pushing values (or addresses) on the stack. Extra information is passed for records ('CONSTRAINED) and for arrays (dope vector address).

Small results are returned in registers; large results with known targets are passed by reference. Large results of anonymous target and known size are passed by reference to a temporary created in the caller. Large results of anonymous target and unknown size are returned by copying the value down from a temporary in the callee so that the space used by the temporary can be reclaimed.

The compiler assumes the following calling conventions.

1. caller pushes arguments on stack in reverse order from their declaration
2. caller calls callee
3. callee builds display and allocates space for local variables
4. callee executes
5. callee leaves result in EAX or top of floating point stack (ST(0)) if callee is a function
6. callee deallocates local variables (LEAVE)
7. callee returns to caller
8. caller copies back any out parameters or function value
9. caller deallocates space used for arguments on the stack

Note that the compiler does not depend on any general purpose or floating point registers being the same after a call, so they do not have to be saved. As stated above, additional steps may be taken when structures are passed or returned.

Caution: Compilers for other languages may follow calling conventions other than those expected by VADS, for example, pushing arguments in the opposite direction. The VADS debugger should be used to verify that the call interface is as expected.

Machine code insertions can be used to explicitly build a call interface when compiler conventions are not compatible, or when interfacing to assembly language.

For example, suppose an interface to a C function **pass_int** is desired, where the C compiler generated code such that the callee deallocates space for the parameters.

```
int pass_int(x);
  int x;
{...
}
```

The following Ada code would provide a wrapper to call this function, while allowing the C function to handle the deallocation.

```
with MACHINE_CODE;
function PASS_INT(X : INTEGER) return INTEGER is

 RETURN_VAL : INTEGER;

 procedure WRAPPER is
   use MACHINE_CODE;
 begin
   CODE_1'(PUSH, X'REF);                -- push x onto the stack
   CODE_1'(CALL, EXT("_pass_int"));  -- call pass_int via its link name
                                        -- don't need to increment stack
                                        -- pointer, pass_int has taken
                                        -- care of it
   CODE_2'(MOV, EAX, RETURN_VAL'REF); -- save result
 end WRAPPER;

begin
  WRAPPER;
  return RETURN_VAL;
end PASS_INT;
```

NOTE:

This example and some that follow assume the existence of a C compiler that generates object files compatible with the VOX format files produced by VADS.

See also *Chapter 12, Machine Code Insertions.*

18.12 CONVERSION AND DEALLOCATION

The predefined **generic function** UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

There are no restrictions on the types with which **generic function** UNCHECKED_DEALLOCATION can be instantiated. No checks are performed on released objects.

18.13 PROCESS STACK SIZE

The compiler and other large dynamic compiled programs can occasionally give problems due to the shell's stack limit. Altering the stack size and recompiling or re-executing is sometimes necessary. A process inherits its stack limit from the invoking process, usually the shell.

18.14 INTERFACE PROGRAMMING

pragma INTERFACE — This pragma supports calls to the language names C, PASCAL, FORTRAN, and UNCHECKED. The Ada specifications can be either functions or procedures. All parameters must have mode **in**.

For C, the types of parameters and the result type for functions must be scalar, access, or the predefined **type** ADDRESS in SYSTEM.ADDRESS. Record and array objects can be passed by reference using the 'ADDRESS attribute.

For PASCAL, the types of parameters and the result type for functions must be scalar, access, or the predefined **type** ADDRESS in SYSTEM.ADDRESS. Record and array objects can be passed by reference using the ADDRESS attribute.

For FORTRAN, all parameters are passed by reference; the parameter types must have **type** SYSTEM.ADDRESS. The result type for a FORTRAN function *must* be a scalar type.

UNCHECKED may be used to interface to an unspecified language, such as assembler. The compiler will generate the call as if it were to an Ada procedure, but will not expect a matching Ada procedure body. **See also** *[IMPLEMENTATION REFERENCE], 18.11 PARAMETER PASSING,* page 18-23

pragma INTERFACE_NAME allows variables and subprograms defined in another language to be referenced directly in Ada. **pragma** INTERFACE_NAME replaces all occurrences of *Ada_subprogram_name* with an external reference to *link_name* in the object file using the format shown below.

```
pragma INTERFACE_NAME (Ada_subprogram_name, "link_name");
```

If *Ada_subprogram_name* denotes an object, the pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object *cannot* be any of the following.

- loop variable
- constant
- initialized variable
- array
- record

If *Ada_subprogram_name* denotes a subprogram, a **pragma** INTERFACE must have already been specified for the subprogram.

The *link_name* must be constructed as expected by the linker. For example, some C compilers and linkers preface the C variable name with an underscore. The example below makes the C global variable **errno** available within an Ada program.

```
package PACKAGE_NAME is
   ...
   ERRNO:INTEGER;
   pragma INTERFACE_NAME (ERRNO,"_errno");
   ...
end PACKAGE_NAME;
```

pragma EXTERNAL_NAME allows the user to specify a *link_name* for an Ada variable or subprogram so that the Ada object can be referenced from other languages using the syntax shown below.

```
pragma EXTERNAL_NAME (object_or_subprogram_name, "link_name");
```

Objects must be variables defined in a package specification; subprograms can be either library level or within a package specification.

See also, *Chapter 13, INTERFACE PROGRAMMING USING VADS.*

18.15 PREDEFINED PACKAGES AND GENERICS

The following predefined Ada packages given by RM Appendix C(22) are provided in the STANDARD library.

- **package** STANDARD
- **package** CALENDAR
- **package** SYSTEM
- **generic procedure** UNCHECKED_DEALLOCATION
- **generic function** UNCHECKED_CONVERSION
- **generic package** SEQUENTIAL_IO
- **generic package** DIRECT_IO
- **package** TEXT_IO
- **package** IO_EXCEPTIONS
- **package** LOW_LEVEL_IO
- **package** MACHINE_CODE

18.15.1 Specification Of package SYSTEM

```
package SYSTEM
is
   type NAME is ( i386_System_V );

   SYSTEM_NAME     : constant NAME := i386_System_V;

   STORAGE_UNIT : constant := 8;
   MEMORY_SIZE     : constant := 16_777_216;

   -- System-Dependent Named Numbers

   MIN_INT         : constant := -2_147_483_648;
   MAX_INT         : constant := 2_147_483_647;
   MAX_DIGITS      : constant := 15;
   MAX_MANTISSA : constant := 31;
   FINE_DELTA      : constant := 2.0**(-31);
   TICK         : constant := 0.01;

   -- Other System-dependent Declarations

   subtype PRIORITY is INTEGER range 0 .. 99;

   MAX_REC_SIZE : integer := 64*1024;

   type ADDRESS is private;

   NO_ADDR : constant ADDRESS;

   function PHYSICAL_ADDRESS(I: INTEGER) return ADDRESS;
   function ADDR_GT(A, B: ADDRESS) return BOOLEAN;
   function ADDR_LT(A, B: ADDRESS) return BOOLEAN;
   function ADDR_GE(A, B: ADDRESS) return BOOLEAN;
   function ADDR_LE(A, B: ADDRESS) return BOOLEAN;
   function ADDR_DIFF(A, B: ADDRESS) return INTEGER;
   function INCR_ADDR(A: ADDRESS; INCR: INTEGER) return ADDRESS;
   function DECR_ADDR(A: ADDRESS; DECR: INTEGER) return ADDRESS;

   function ">"(A, B: ADDRESS) return BOOLEAN renames ADDR_GT;
   function "<"(A, B: ADDRESS) return BOOLEAN renames ADDR_LT;
   function ">="(A, B: ADDRESS) return BOOLEAN renames ADDR_GE;
   function "<="(A, B: ADDRESS) return BOOLEAN renames ADDR_LE;
   function "-"(A, B: ADDRESS) return INTEGER renames ADDR_DIFF;
   function "+"(A: ADDRESS;
                INCR: INTEGER) return ADDRESS renames INCR_ADDR;
   function "-"(A: ADDRESS;
                DECR: INTEGER) return ADDRESS renames DECR_ADDR;

   pragma inline(ADDR_GT);
   pragma inline(ADDR_LT);
   pragma inline(ADDR_GE);
   pragma inline(ADDR_LE);
```

```
      pragma inline(ADDR_DIFF);
      pragma inline(INCR_ADDR);
      pragma inline(DECR_ADDR);
      pragma inline(PHYSICAL_ADDRESS);

   private

      type ADDRESS is new integer;

      NO_ADDR : constant ADDRESS := 0;

   end SYSTEM;
```

18.15.2 package CALENDAR

CALENDAR's clock function (in **package** CALENDAR.LOCAL_TIME located in the file **calendar_s.a**) uses the UNIX service routine LOCALTIME for getting the current time.

18.15.3 package SEQUENTIAL_IO

Sequential I/O is currently implemented for variant records, but with the restriction that the maximum size possible for the record will always be written. This is also true of direct I/O. For unconstrained records and arrays, the constant, SYSTEM.MAX_REC_SIZE, can be set prior to the elaboration of the generic instantiation of SEQUENTIAL_IO or DIRECT_IO. For example, if unconstrained strings are written, SYSTEM.MAX_REC_SIZE effectively restricts the maximum size of string that can be written. If the user knows the maximum size of such strings, the SYSTEM.MAX_REC_SIZE may be set prior to instantiating SEQUENTIAL_IO for the string type. This variable can be reset after the instantiation with no effect.

18.16 TYPES, RANGES, AND ATTRIBUTES

Numeric Literals — VADS Ada uses unlimited precision arithmetic for computations with numeric literals.

Enumeration Types — VADS Ada allows an unlimited number of literals within an enumeration type.

Attributes of Discrete Types — VADS Ada defines the image of a character that is not a graphic character as the corresponding 2- or 3-character identifier from **package** ASCII of RM Annex C-4. The identifier is in uppercase without enclosing apostrophes. For example, the image for a carriage return is the 2-character sequence CR (ASCII.CR).

The type STRING — Except for memory size, VADS Ada places no specific limit on the length of the predefined type STRING. Any type derived from the type STRING is similarly unlimited.

Name of Attribute	Attribute Value of INTEGER	Attribute Value of SHORT_INTEGER	Attribute Value of TINY_INTEGER
FIRST	-2_147_483_648	-32_768	-128
LAST	2_147_483_647	32_767	127

Operation of Floating Point Types — VADS Ada floating point types have the attributes listed below.

Name of Attribute	Attribute Value of FLOAT	Attribute Value of SHORT_FLOAT
SIZE	64	32
FIRST	-1.79769313486231E+308	-3.40282E+38
LAST	1.79769313486231E+308	3.40282E+38
DIGITS	15	6
MANTISSA	51	21
EPSILON	8.88178419700125E-16	9.53674316406250E-07
EMAX	204	84
SMALL	1.94469227433160E-62	2.58493941422821E-26
LARGE	2.57110087081438E+61	1.93428038904620E+25
SAFE_EMAX	1021	121
SAFE_SMALL	2.22507385850720E-308	1.17549435082228E-38
SAFE_LARGE	2.24711641857789E+307	4.25352755827077E+37
MACHINE_RADIX	2	2
MACHINE_MANTISSA	53	24
MACHINE_EMAX	1024	128
MACHINE_EMIN	-1021	-125
MACHINE_ROUNDS	TRUE	TRUE
MACHINE_OVERFLOWS	TRUE	TRUE

Fixed Point Types — VADS Ada provides fixed point types mapped to the supported integer sizes.

Operations of Fixed Point Types — VADS Ada fixed point type DURATION has the attributes shown below.

Name of Attribute	Attribute Value for DURATION
SIZE	32
FIRST	-2147483.648
LAST	2147483.647
DELTA	1.00000000000000E-03
MANTISSA	32
SMALL	9.76562500000000E-04
LARGE	4.19430399902343E_06
FORE	8
AFT	3
SAFE_SMALL	9.76562500000000E-04
SAFE_LARGE	4.19430399902343E+06
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE

18.17 INPUT/OUTPUT

The Ada I/O system is implemented using UNIX I/O. Both formatted and binary I/O are available. There are no restrictions on the types with which DIRECT_IO and SEQUENTIAL_IO can be instantiated, except that the element size must be less than a maximum given by the variable SYSTEM.MAX_REC_SIZE. This variable can be set to any value prior to the generic instantiation; thus, the user can use any element size. DIRECT_IO can be instantiated with unconstrained types, but each element will be padded out to the maximum possible for that type or to SYSTEM.MAX_REC_SIZE, whichever is smaller. No checking — other than normal static Ada type checking — is done to ensure that values from files are read into correctly sized and typed objects.

VADS file and terminal input–output are identical in most respects and differ only in the frequency of buffer flushing. Output is buffered (buffer size is 1024 bytes). The buffer is always flushed after each write request if the destination is a terminal. The procedure FILE_SUPPORT.ALWAYS_FLUSH (*file_ptr*) will cause flushing of the buffer associated with *file_ptr* after all subsequent output requests. Refer to the source code for **file_spprt_b.a** in the **standard** library.

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as a string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed before instantiating DIRECT_IO to provide an upper limit on the record size. The maximum size supported is 1024 * 1024 * STORAGE_UNIT bits. DIRECT_IO will raise USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as STRING where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_REC_SIZE is defined in SYSTEM and can be changed by a program before instantiating INTEGER_IO to provide an upper limit on the record size. SEQUENTIAL_IO imposes no limit on MAX_REC_SIZE.

18.18 MACHINE CODE INSERTIONS

The general definition of **package** MACHINE_CODE provides an assembly language interface for the target machine including the necessary record types needed in the code statement, an enumeration type containing all the opcode mnemonics, a set of register definitions, and a set of addressing mode functions. Also supplied (for use only in units that WITH MACHINE_CODE) are **pragma** IMPLICIT_CODE and the attribute 'REF.

Machine code statements take operands of **type** OPERAND, a private type that forms the basis of all machine code address formats for the target.

The general syntax of a machine code statement is

```
CODE_n'(opcode, operand [, operand]));
```

where *n* indicates the number of operands in the aggregate.

When there is a variable number of operands, they are listed within a subaggregate using the syntax shown below.

```
CODE_n'(opcode, (operand [, operand]);
```

In the example shown below, **code_2** is a record 'format' whose first argument is an enumeration value of **type** OPCODE followed by two operands of **type** OPERAND.

```
CODE_2'(add, a'ref, b'ref);
```

For those opcodes requiring no operands, named notation must be used (**see also** Ada RM 4.3(4)).

```
CODE_0'(op => opcode);
```

The *opcode* must be an enumeration literal (i.e., it cannot be an object, attribute, or a rename). An *operand* can only be an entity defined in MACHINE_CODE or the 'REF attribute.

Arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals, or the functions defined in MACHINE_CODE.

'REF — The 'REF attribute denotes the effective address of the first of the storage units allocated to the object. 'REF is not supported for a package, task unit, or entry. For details, **see also** *[IMPLEMENTATION REFERENCE], 18.5 IMPLEMENTATION-DEFINED ATTRIBUTE,* page 18.5.

pragma IMPLICIT_CODE — The IMPLICIT_CODE pragma specifies that implicit code generated by the compiler is allowed (ON) or disallowed (OFF) and is used only within the declarative part of a machine code procedure. Implicit code includes preamble and postamble code (e.g., code used to move parameters from and to the stack). Use of **pragma** IMPLICIT_CODE does not eliminate code generated for runtime checks, nor does it eliminate call/return instructions (these can be eliminated by **pragma** SUPPRESS and **pragma** INLINE, respectively). A warning is issued if OFF is used and any implicit code needs to be generated. This pragma should be used with caution.

See also *Chapter 12, Machine Code Insertions.*

18.19 VADS FILE FORMATS

File formats of several types represent the interface between various portions of the VADS tools. For example, the IL (Intermediate Language) file generated by the front end is used as input to the code generator. This section contains detailed file formats that are of particular interest to the VADS user.

The details of these files are generally of interest only for specialized applications involving the creation or modification of object or other files by tools other than those supplied with VADS. Use care when modifying reference files such as **ada.lib** with a text editor. Special tools are provided for reading and modifying this file for general use.

See also *LIBRARY MANAGEMENT TOOLS, 4.7.3* **a.info**, page 4-6 and *4.7.4* **a.ls**, page 4-8.

18.19.1 Object, Executable and Archive Files

This implementation of VADS generates object and executable files as described in the host operating system documentation under **a.out**(4) and in **/usr/include/a.out.h**. The system utility **ar**(1) can be used to generate libraries that can be linked with objects generated by the VADS Ada compiler.

For a discussion of external dependencies (WITH*n* directives), **see also** *PROGRAM GENERATION TOOLS, 6.3* **a.ld**, page 6–1.

18.19.2 VADS Library Reference Files: ada.lib and gnrx.lib

Although the **ada.lib** and **gnrx.lib** files are standard text files and could be edited with a text editor (e.g., to insert or change INFO directives or to change the library search list), it is recommended that only the tools designed for this purpose, **a.path** and **a.info**, be used. In addition, some lines of the file may extend to lengths not correctly processed by many standard text editors. Nevertheless, a description of the **ada.lib** reference file can be of some aid in understanding operation of the compiler. A sample **ada.lib** file is shown below.

```
!ada library
ADAPATH=
LIBRARY:LINK:/rc/vads5.7/VADS/standard/.objects/library.a:
TASKING:LINK:/rc/vads5.7/VADS/standard/.objects/tasking.a:
TARGET:INFO:SELF_TARGET:
FLOATING_POINT_SUPPORT:INFO:SOFTWARE:
VERSION:INFO:5.7:
VADS:INFO:/rc/vads5.7/VADS:
STANDARD:BNLPS:STANDARD00:
hex:NLPS:hex01:
hex:NLPB:hex_b01:
low_level_io:NLPB:lowlevel_io02:
low_level_io:NLPS:lowlevel_io01:
```

The first line is an internal identification of the file and may not be changed. The second line contains the library search list, an ordered list of predecessor libraries. The maximum length of this line (excluding the characters preceding the first library directory name) is 2048 characters. Subsequent lines are of three types, each having three fields that are separated by colons, and ending with a colon.

LINK directives have the word LINK as the second field. A name is in the first field (having some specific meaning to the linker) or the word WITH*n* directing the linker to link the value in the third field (a file name or library) each time a program is linked using this library. INFO directives have the word INFO in the second field, a name of some significance to the compiler and other VADS tools in the first field, and a value in the third field. The use of LINK and INFO directives is described in the chapter *LIBRARY MANAGEMENT TOOLS* in the *USER'S GUIDE*. In this release, the following directive names are processed.

Other lines in an **ada.lib** file are the result of a compilation (or attempted compilation) in the library and have the following form.

unit_name:*type*:*value*:

There is one entry for each unit's specification or body; *unit_name* is always recorded in lower case. The *value* field is a concatenation of the file name containing the unit, and a 2-character sequence incremented for each Ada library entry from a particular file. The *type* field contains a combination of letters categorizing the unit, the meanings of which are listed below. (**a.ls** can be used to present this information in user readable form.)

- B no body needed
- E **pragma** ELABORATE
- T has a task
- M could be a 'main' program
- U uncompiled
- D contains inlines
- S shared generic instantiation
- I unshared generic instantiation
- N non-generic
- G generic
- L library unit
- S subunit
  - P package
  - S subprogram
  - T task
  - B subprogram body

Entries for generic instantiations have a slightly different form giving the generic names and a sequential number concatenated into INSTXX in the *value* field. A typical entry for an instantiation would be similar to that shown below.

```
float_io$text_io$27:SLPB:INST28XX:
```

This indicates a particular instantiation of the FLOAT_IO package from the **package** TEXT_IO rather than the generic body itself.

Although the file **gnrx.lib** is composed of readable lines, there is no user-useful information contained therein.

18.19.3 Supported INFO and LINK Directive Names

The VADS tool **a.info** is used to examine the INFO and LINK directives of the **ada.lib.** It can also be used to add or delete those directives from the **ada.lib** or to display or change the library search list.

18.19.3.1 INFO Directive Names

The directives listed and described below are supported.

TARGET:INFO:*target_processor_name*:	name of target processor
VADS:INFO:VADS_*location*:	pointer to release area
VERSION:INFO:*version_number*:	current version of VADS

TARGET — The TARGET directive informs the compiler and other tools the name of the target processor for which code is being generated.

TARGET:INFO:*target_processor_name*:

For all self-target releases, *target_processor_name* is SELF_TARGET.

VADS — The VADS directive provides a pointer to the directory containing a particular version of VADS. Its purpose is to provide VADS tools with a base location for the release area.

VADS:INFO:*VADS_location*:

VERSION — The VERSION directive specifies which version Ada library this is and what version of other VADS tools have been working in this Ada library.

VERSION:INFO:*version_number*:

See also *LIBRARY MANAGEMENT TOOLS, 4.7.3* **a.info**, page 4–6.

18.19.3.2 LINK Directive Names

There is one directive that is specific to the VADS prelinker **a.ld.** This directive controls a set of general items that should be linked into the Ada program.

`LIBRARY:LINK:`*file_name*`:`	name of runtime system library for programs without tasking
`TASKING:LINK:`*file_name*`:`	name of runtime system library for programs with tasking
`WITH`*n*`:LINK:`*string*`:`	string to pass to linker

LIBRARY — The LIBRARY directive provides the name of the file that contains the VADS Runtime System (RTS) library for programs that do not use tasking.

```
LIBRARY:LINK:file_name:
```

TASKING — The TASKING directive provides the name of the file that contains the VADS Runtime System (RTS) library for programs that use tasking. The CPU linker will choose the correct link entry to use.

```
TASKING:LINK:file_name:
```

WITH*n* — The WITH*n* directive permits the user to add files and/or commands to be passed to the linker by the VADS prelinker **a.ld.**

```
WITHn:LINK:string:
```

Any break in the sequence WITH1, WITH2, WITH3, ... causes the prelinker to stop processing WITH*n* directives. If there are directives with the names WITH1, WITH2, WITH4, and WITH5, then only WITH1 and WITH2 would be processed; since WITH3 is missing, WITH4 and WITH5 would never be found.

18.20 RUNTIME SYSTEM

This section is an overview of the VADS Runtime System (RTS) from a user perspective. Is is intended as an aid for developing and tuning applications using VADS. It does not require access to the source code defining the runtime algorithms.

Knowledge of Ada is assumed, particularly in the use of the tasking terms *master, terminate, complete, abort, rendezvous, select, accept,* and *call.* These are described in RM 9.

The term *thread* is used here to denote a thread of execution and represents the execution state (stack and registers) of a task.

For information on the machine-specific aspects of the VADS RTS, **see also** *INSTALLATION GUIDE, RUNTIME SYSTEM, 17.4 CONFIGURING THE VADS RUNTIME SYSTEM,* page 17-7.

18.20.1 Overview

The RTS is divided into two major parts. The first is the kernel, a group of services that are linked into the user program.

The second major part of the RTS is the library, a traditional support archive of routines. These routines might be linked to the user program, as required, to perform specific functions.

Execution normally begins in the kernel, where the underlying data structures for the machine and the Ada services, such as tasking, are initialized. After the kernel initializes itself, it transfers control to the user program at a starting address.

The program normally operates in user mode for the machine architecture. Typically the user program is restricted from using certain supervisor instructions.

Elaboration of the user program consists of a call to each address in a table built by the VADS prelinker. It makes explicit reference to each available service whether required by the user program or not. The next-to-last entry in the vector is the user's main program, the last entry is a suitable exit routine that cleans up the user's resources and returns a result to the kernel.

See also Ada RM 10.5.

Once user program execution begins, there are two routes back to the kernel. One route is by means of a service call generated by the compiler, explicit library or explicit user code. The service call convention must be in agreement among the compiler, library, user program, and kernel. Arguments to the service are loaded into the user call stack, accessible to both the user program and the kernel, and the desired service is identified by a service number, loaded into a register.

The other route to the kernel is via a signal, such as an arithmetic overflow fault or an asynchronous timer interrupt. Interrupt handling may be configured to the needs of the user, but generally, the occurrence of an interrupt transfers control to a kernel handler. The system state, the thread of execution, is saved to enable resumption of the program. The handler usually saves all of the system state, but for some simple interrupts, only the required registers are saved. After the interrupt-specific response is executed by the kernel, execution of the user program resumes.

Interrupts are generally handled by the underlying operating system and arrive at the RTS kernel in the form of signals for UNIX.

18.20.2 Memory Allocation

When user programs are compiled, the compiler calculates the amount of storage each entity needs, or generates a call to a subprogram that calculates the storage needs for dynamically allocated objects. The compiler also generates code using a model for subprogram invocation that specifies where each local entity will be positioned on the current execution stack.

When programs are linked, the linker assigns memory locations for each static entity. An entity is static if its size is known at compile time and if it will conceptually exist throughout the lifetime of the program.

Dynamic entities that are *not* defined local to a specific subprogram must be assigned addresses by the runtime system. Users create these dynamic objects using the Ada allocator construct. For example, the following code might be generated:

```
6       v := new integer (3);
      001b: push      04
      0020: call      00    -> AA_GLOBAL_NEW
      0025: add       esp, 04
      0028: mov       ecx, eax
      002a: mov       -08[ebp], ecx
      002d: mov       [ecx], 03
```

In this example, the compiler was able to determine the size of the allocated object, 4 bytes. It passed the requested size to the runtime memory allocator AA_GLOBAL_NEW. The runtime allocator assigns a block of four contiguous bytes from its supply of unassigned memory (its heap) and returns the address of that memory in register d0. It marks that memory as *in use* so that no other allocation can employ it.

This level of memory allocation is called user level. The activity normally occurs only in user space as a call between the user program and the library utilities. The kernel is not involved.

For more information about address clauses, see also Ada RM 13.5 and for more about allocators, see also Ada RM 4.8.

Memory allocation, however, can involve the kernel in that the AA_GLOBAL_NEW implementation may call kernel services. The kernel provides its own memory allocation service, reached by the following kind of code:

```
with system;
with machine_code; use machine_code;

function alloc_from_krn(size: integer) return system.address is
   result: system.address;
   procedure call_krn is
   begin
      code_1'(push, size'ref);
      code_1'(call, ext("KRN_AA_GLOBAL_NEW"));
      code_2'(add, esp, +4);
      code_2'(mov, eax, result'ref);
   end;
   pragma inline(call_krn);
begin
   call_krn;
   return result;
end;
```

The package KRN_ENTRIES, supplied as part of the standard environment by Verdix, lists the kernel entry numbers of all services supplied by the VADS kernels. Note that not all services are implemented for every target because some are machine-specific.

The kernel, in any case, always executes on its own stack. User program stacks do not have to save space for the kernel. However, in some self-hosted systems, interrupts are processed using the current stack, be it kernel or user. When this is the case, stacks must reserve space for interrupt information.

For any system, access to kernel memory allocation routines is provided by a Verdix package called IFACE_ALLOC. IFACE_ALLOC stands for 'interface to allocation'. To obtain from the kernel an address pointing to four unused bytes, the user can write

```
iface_alloc.aa_global_new(4).
```

18.20.3 Semaphores

The kernel is a critical region, enforcing the rule that no two processes (threads) may be executing within the kernel at the same time. Verdix supplies a standard semaphore service to allow its users to define their own critical regions. These services are defined in **package** IFACE_SEMA.

Like memory allocation, the normal execution of semaphore routines occurs in user memory, supplied by code from the VADS library. Only when a semaphore is blocked will the kernel be called to suspend tasks, and later resume them.

Unlike memory allocation, semaphore calls are *not* generated by the VADS compiler. There is no syntax in Ada for a semaphore-protected simple critical region. For now, users can get semaphores only by direct calls to **package** IFACE_SEMA.

18.20.4 Tasking

From the runtime perspective, the semantics of tasks are complicated. They are implemented by a collection of services in the kernel as well as by code generated in the compiler.

Following is a tasking example. The concept is taken from the RM 9.12. Subsequent to it, are discussions of its runtime interactions including creation, activation, startup **delay** statements, **entry** calls, **accept** and **select** statements, completion, and termination.

```
 1:with text_io;
 2:procedure buffer is
 3:    task buffer is
 4:        entry read(c: out character);
 5:        entry write(c: in character);
 6:    end;
 7:    task producer is
 8:    end;
 9:    task consumer is
10:    end;
11:    task body producer is
12:    begin
13:        ...
```

```
14:   buffer.write(c)
15:      ...
16:   end;
17:   task body consumer is
18:      ...
19:   begin
20:      ...
21:         select buffer.read(c);
22:            ...
23:         else delay(0.001);
24:         end  select;
25:      ...
26:   end;
27:   task body buffer is
28:   begin
29:      ...
30:         select when count < pool_size =>
31:            accept write(c: in character) do ...  end;
32:            ...
33:         or  when count > 0 =>
34:            accept read(c: out character) do ...  end;
35:            ...
36:         or terminate;
37:         end select;
38:      ...
39:   end buffer;
40:begin
41:   ...
42:   while buffer'callable loop
43:      delay(0.001);
44:   end loop;
45:   ...
46:end;
```

18.20.4.1 Task Creation

Tasks come into being in two stages, creation and activation. Tasks are created by the elaboration of declarations of objects that are either defined as tasks or contain an instance of a task type. Both tasks and task type instances are treated identically by the runtime.

In our example, the code generated for lines 3–6, creating the buffer task, is:

```
3       task buffer is
        03d0: call      00    -> TS_INIT_ACTIVATE_LIST
        03d5: mov       -0c[ebp], eax
        03d8: mov       -010[ebp], 00
        03df: call      00    -> TS_CREATE_MASTER
        03e4: mov       -010[ebp], eax
        03e7: push      00
        03ec: push      -0c[ebp]
        03ef: push      02
        03f4: push      01f8    @_A_producer.buffer + 0x1f8
        03f9: push      02800
        03fe: push      00
        0403: push      -010[ebp]
        0406: call      00    -> TS_CREATE_TASK_AND_LINK
        040b: add       esp, 01c
        040e: mov       -08[ebp], eax
```

The call to TS_INIT_ACTIVATE_LIST initializes a task activation list for this scope (for all the tasks created directly within the declarative part of the test procedure.) The task list generally resides in kernel space, necessitating this call. A pointer to the list head is returned.

The call to TS_CREATE_MASTER initializes a master structure. This structure will keep track of all the tasks that must terminate before we can complete this scope (in this example, complete the program.) A pointer to the master structure in the kernel space is returned.

The call to TS_CREATE_TASK_AND_LINK causes the creation of a task control data structure in the kernel. A pointer to this structure is returned. Parameter values are assigned by the compiler, with STACK_SIZE and priority given by default or by Ada pragmas. The task is linked to the activation list but is not activated at this point.

The return value from this task creation function is the task's descriptor. The compiled code stores this descriptor in the storage class implied by the scope of the object declaration. It uses this descriptor as the task's value. Whenever the task must be named by the generated code to the RTS this descriptor is passed to the RTS. This descriptor is really the pointer to the task's record, an RTS data structure. The generated code does not take advantage of this fact. On the embedded systems, the generated code could not use these pointers since the RTS and the generated code are in different memory spaces.

Our VADS debugger will now know of this new task, so the command It elicits the list:

Q#	TASK	ADDR	STATUS
	buffer'2	055d98	not yet active
*	<main program>	045068	executing

The other tasks will be created in a similar fashion, for PRODUCER by the code at line 7 and CONSUMER by the code at line 9. Now **lt** gives:

```
Q#      TASK              ADDR      STATUS
        consumer          05b7f8    not yet active
        producer          058ac8    not yet active
        buffer'2          055d98    not yet active
*       <main program>    045068    executing
```

18.20.4.2 Task Activation

At the **begin** that follows a declarative part that contains tasks, the program emits a call to TS_ACTIVATE_LIST. The activation list is the one being used for the declarative part. The RTS will activate the tasks on the list and return. The compiler calls TS_ACTIVATION_EXCEPTIONS immediately following the call to TS_ACTIVATE_LIST. This will make the activating task wait for the children that were just activated to run and elaborate their declarations. Each child will call TS_ACTIVATION_COMPLETE which will increment a count of successfully activated tasks. The last child to do this will wake up the activating task.

At line 40 we see the following code generated:

```
40  begin
      0481: push      00
      0486: push      -0c[ebp]
      0489: call      00    -> TS_ACTIVATE_LIST
      048e: add       esp, 08
      0491: call      00    -> TS_ACTIVATION_EXCEPTIONS
      0496: mov       ecx, eax
      0498: test      ecx, ecx
      049a: je        04c8
      049c: cmp       ecx, 01
      049f: jne       04a6
      04a1: call      00    -> RAISE_TASKING_ERROR
      04a6: call      00    -> RAISE_PROGRAM_ERROR
```

When all the tasks from the example activate and elaborate correctly, an **lt** command at an instruction breakpoint at 0496 gives this output:

```
Q#      TASK              ADDR      STATUS
D1      consumer          05b7f8    suspended at delay
              on delay queue, until 141 millisecs
R1      producer          058ac8    ready
        buffer'2          055d98    suspended at select (termination possible)
              open entries:  write  read
*       <main program>    045068    executing
```

It is possible for the elaboration of the declarative part of a task body to cause an exception. In this case, TS_ACTIVATION_COMPLETE will be called by an anonymous exception handler with a parameter indicating abnormal activation. The following code is generated as this implicit anonymous handler:

```
04b0: mov       -01c[ebp], eax
04b3: push      -010[ebp]
04b6: call      00    -> TS_EXCEPTION_MASTER
04bb: add       esp, 04
04be: push      -01c[ebp]
04c1: call      00    -> RAISE
04c6: jmp       04ab
```

The handler saves the exception (from register eax) and then calls the TS_EXCEPTION_MASTER service to await termination of any subtasks of the current task (there are none in our example PRODUCER). It then reraises the exception.

This will wake up the activating task (which in this case is the main program) even though some of the sibling activated tasks may not yet have called TS_ACTIVATION_COMPLETE. The newly activated tasks will be completed and terminated by the TS_ACTIVATION_EXCEPTIONS service executing for that activating main program; it returns to the main program with a non-zero value:

```
0  all is ok during activation
1  elaboration error, raising TASKING_ERROR
2  activation error, raising PROGRAM_ERROR
```

An lt command issued from an instruction breakpoint immediately following 0496 in the above shows the following after an exception was raised in the declarative part:

```
Q#      TASK              ADDR      STATUS
D1      consumer          05b7f8    suspended at delay
            on delay queue, until 161 millisecs
        producer          058ac8    terminated
        buffer'2          055d98    suspended at select (termination possible)
            open entries:  write
*       <main program>    045068    executing
```

18.20.4.3 Task Startup

The code for task startup is emitted at the start of the text for the task body. The code for PRODUCER begins:

```
11      task body producer is
      0000: enter     024, 03
      0004: cmp       00, esp    @STACK_LIMIT
      000a: jb        014
      000c: add       esp, 024
      000f: call      00    -> RAISE_STORAGE_ERROR
```

```
        0014: call      00    -> TS_TID
        0019: mov       -010[ebp], eax
        001c: xor       eax, eax

        ... elaboration of producer's declarations here ...

14        begin
        002f: push      eax
        0030: call      00    -> TS_ACTIVATION_COMPLETE
        0035: add       esp, 04
```

As can be seen, the task body code has the same procedure startup as other subprograms, getting stack space, checking for stack limits, and then setting up the addressing environment for reaching entities in out parts of the program (these are copied in from the caller's stack frame.)

The only purely tasking activity before elaboration is the call to TS_TID to get and record the ID of this task. This value is the task data structure address in the kernel space.

After any elaboration for the task is complete, a call to TS_ACTIVATION_COMPLETE informs the parent task of the success of the activation. Were this the last task on the activation list, then the activating task (the main program here) would be placed on the run queue for further execution.

18.20.4.4 Delay Statements

Simple **delay** statements (that is, those not in the **else** part of a **select**) are implemented as calls to the TS_DELAY service:

```
68          delay(0.001);
        04eb: push      01
        04f0: call      00    -> TS_DELAY
        04f5: add       esp, 04
```

The fixed point number is pushed as the parameter to TS_DELAY. The following **lt** command output illustrates the main program after it has requested a delay at line 41:

```
Q#      TASK            ADDR        STATUS
        consumer        05b7f8      in rendezvous
        producer        058ac8      suspended calling buffer'2[055d98].write
*       buffer'2        055d98      executing
              in rendezvous with consumer[05b7f8] at entry entry_0
D1      <main program>  045068      suspended at delay
              on delay queue, until 8441 millisecs
```

Tasks requesting delays are put on a DELAY_QUEUE. Each timer interrupt checks each task on this queue to see if CURRENT_TIME has passed its time to awaken; each such task is then queued on the run queue.

18.20.4.5 Task Entry Calls

The compiler translates unconditional and untimed entry calls into calls to TS_CALL. TS_CALL takes three parameters; the task descriptor of the task being called, the entry ID of the entry being called (when the compiler sees a task specification it assigns integers, starting at one, to the declared entries), and the address of a parameter block for the call (parameter blocks are made to look like the stack memory for subprogram parameters).

task PRODUCER contains a simple entry call that provides characters to **task** BUFFER
The entry call has the following generated code:

```
17              buffer.write(c);
      0070: mov       ecx, -04[ebp]
      0073: mov       ecx, -08[ecx]
      0076: push      [eax]                -- Push C
      0078: mov       eax, esp
      007a: add       eax, -08
      007d: into
      007e: push      eax                  -- address of parameter
 -- block (i.e., C)
      007f: push      02                   -- entry ID
      0084: push      ecx                  -- task descriptor
      0085: call      00    -> TS_CALL
      008a: add       esp, 0c
      008d: add       esp, 04
```

The TS_CALL routine will try to do an immediate rendezvous if possible. Rendezvous is possible if the called task (BUFFER) is suspended at an **accept** or at a **select** statement and is waiting at the entry being called. For immediate rendezvous control is transferred to the called task. This is almost like calling it as a subprogram except that when the rendezvous is completed, both the called task and the calling task will be able to execute.

Following is the **lt** output during a rendezvous:

```
Q#      TASK              ADDR        STATUS
D2      consumer          05b7f8      suspended at delay
              on delay queue, until 17041 millisecs
        producer          058ac8      in rendezvous
*       buffer'2          055d98      executing
            in rendezvous with producer[058ac8] at entry write
D1      <main program>    045068      suspended at delay
            on delay queue, until 17041 millisecs
```

If the call can not be done immediately, it is because the called task is not waiting at an **accept** or **select** or because it is not waiting for a call of the entry that we are calling. The called task becomes suspended on the entry queue, waiting for the called task to accept that entry. The following It command illustrates what happens when PRODUCER suspends waiting for BUFFER to accept a WRITE:

```
Q#      TASK              ADDR      STATUS
D1      consumer          05b7f8    suspended at delay
              on delay queue, until 121 millisecs
        producer          058ac8    suspended calling buffer'2[055d98].write
*       buffer'2          055d98    executing
        <main program>    045068    awaiting activations
```

In our example, eventually BUFFER will get around to accepting PRODUCER. An It output at that time would be identical to an immediate rendezvous except that the status of PRODUCER would show that it is suspended:

```
Q#      TASK              ADDR      STATUS
        producer          058ac8    suspended calling buffer'2[055d98].write
```

Conditional entry calls are implemented with the TS_CONDITIONAL_CALL service. It is the same as a TS_CALL except that if immediate rendezvous is not possible, it returns FALSE. Otherwise it returns TRUE after the rendezvous.

Timed entry calls are implemented with the TS_TIMED_CALL service. Its additional parameter passes a delay duration. If immediate rendezvous is not possible, it suspends after setting up a delay for the requested duration. Whenever a timer interrupt ensues, if this task is still on the DELAY_QUEUE, it can be awakened as though from a delay.

18.20.4.6 Accept and Select Statements

The compiler generates a call to TS_ACCEPT when it sees accept statements outside of select statements. TS_ACCEPT takes the number of the entry being accepted as its only argument. Like TS_CALL, TS_ACCEPT checks to see if immediate rendezvous is possible (i.e., if a task is waiting on the entry queue for the given entry). Otherwise, the task must suspend waiting for some other task to call its entry.

The example has no simple accept statements, but these points are similar to those of the more complex select statement.

Select statement code begins with evaluation of the guards for the accept alternatives. Compiled code builds a list of the open alternatives in a data structure called an entry list. An entry list is an array of integers. The first integer corresponds to the first alternative in the select statement, the second integer corresponds to the second alternative, and so on. If the guard for an entry is closed, a zero is put in the entry list element corresponding to the alternative. If the guard is open, the integer corresponding to the entry being accepted is put in the element corresponding to the alternative.

After building the ENTRY_LIST, a call is generated to one of the TS_SELECT routines:

```
ts_select             simple selects (no else part)
ts_select_terminate   an else part with a terminate
ts_select_else        an else part (conditinal accept)
ts_select_delay       an else part with a delay
```

In our case, TS_SELECT_TERMINATE is called, with the address and length of ENTRY_LIST, a boolean that is true if the terminate alternative is open, and space reserved for two OUT parameters (the result of the select process and the associated parameter block.)

The code for our select statement example illustrates these points.

```
49              select when count < pool_size =>
      0238: lea       edx, -088[ebp]
      023e: lea       edi, -088[ebp]
      0244: xor       eax, eax
      0246: mov       ecx, 02
      024b: cld
      024c: rep stos
      024e: cmp       -078[ebp], 064
      0252: jge       025e
      0254: mov       -088[ebp], 02

55              or  when count > 0 =>
      025e: cmp       -078[ebp], 00

49              select when count < pool_size =>
      0262: jle       026e
      0264: mov       -083[ebp], 01
      026e: mov       eax, 01
      0273: push      ecx
      0274: push      ecx
      0275: movsx     eax, al
      0278: push      eax
      0279: push      02
      027e: push      edx
      027f: call      00    -> TS_SELECT_TERMINATE
      0284: add       esp, 0c
```

```
0287: pop       ecx
0288: pop       edx
0289: cmp       edx, 01
028c: je        032c
0292: jl        0299
0294: cmp       edx, 02
0297: je        02a8
0299: call      00    -> RAISE_PROGRAM_ERROR
```

The TS_SELECT procedure examines its entry queue to see if any task is waiting for any open entry. If so, immediate rendezvous is possible with the first such task. Otherwise, the task will suspend until another task calls an open **entry** (for all **selects**), or until its **delay** expires (for **selects** with **delay** alternatives), or until all other dependent tasks are ready to terminate (for **selects** with open terminate alternatives). **selects** with **else** parts fall immediately to their **else** part unless immediate rendezvous is possible.

The following **lt** command output illustrates our BUFFER task waiting at its select statement:

```
Q#      TASK              ADDR        STATUS
R1      consumer          05b7f8      ready
*       producer          058ac8      executing
        buffer'2          055d98      suspended at select (termination possible)
                open entries: write read
R2      <main program>    045068      ready
```

18.20.4.7 Task Completion and Termination

Tasks come to an end in two stages. First they complete and then they terminate. Once a task is completed it does not run again. A completed task becomes terminated when all tasks that depend on it either complete or agree to terminate.

This code is generated at the end of PRODUCER, a task which has no descendants and simply executes through to completion:

```
19      end;
      00a0: call      00    -> TS_COMPLETE_TASK
```

The TS_COMPLETE_TASK service actually takes tasks all the way through to termination, checking and possibly waiting for any dependent tasks.

In our example, we can breakpoint just after all the sub–tasks have terminated or are waiting at a terminate alternative:

```
Q#      TASK              ADDR        STATUS
        <idle task>       05e528      waiting
        consumer          05b7f8      terminated
```

```
          producer              058ac8      terminated
          buffer'2              055d98      suspended at select (termination possible)
               open entries:    write
*         <main program>        045068      executing
```

Note that BUFFER cannot actually terminate yet because the main program is still executing: it could still call one of the open entries of BUFFER. The code for the main program will try to shut down all the dependent tasks with a call to the TS_COMPLETE_MASTER service:

```
71  end;
      050e: push      -010[ebp]
      0511: call      00    -> TS_COMPLETE_MASTER
      0516: add       esp, 04
      0519: jmp       04ab
      051b: nop
71  end;
      04ab: leave
      04ac: ret
      04ad: nop
```

The main program will now return for completion of the whole program, as is described under program exit.

18.21 RUNTIME SYSTEM PANIC NUMBERS AND MESSAGES

When the runtime system is unable to continue because of an internal inconsistency, a corrupted data structure, an unexpected internal exception, improper entry from a user or interrupt handler, or a tasking deadlock it outputs a panic diagnostic message and exits. The panic message consists of the string 'Error in kernel:' followed by either the panic number or its descriptive text. Some panic situations don't have an associated number. In that case, the message text is always output. A list of the panic messages and their corresponding number (if present) follows.

Note: some of the panic conditions are attributed to an INTERNAL KERNEL ERROR. The most probable cause of such a condition is a corrupted tasking data structure. This can happen if either the kernel or user stack pointer is decremented below its bottom limit into the kernel_heap area (which contains all tasking structures) or if an object of access type points incorrectly into kernel_heap area.

[11] activation complete

internal kernel error: when newly activated task completed its activation, the activation count of its parent task was zero.

[12] call abnormal task

internal kernel error: the caller task's wakeup reason at the end of a rendezvous wasn't ok, delay time-out or exception.

[16] resuming completed task

internal kernel error: a completed task was erroneously put on the run queue.

[19] invalid entry called

The kernel was entered from the user program with an invalid service request ID.

[22] raise called in kernel

A software exception was raised within the kernel, TDM, or an interrupt handler. Note: for cross systems, exception handling isn't supported in the kernel or TDM programs.

[25] called extend stack in kernel

User stack needs to be extended to support exception handling.

[26] terminate list

internal kernel error: a task still on an activation list needing to be terminated isn't in the inactive state.

[27] queued task not suspended at call

internal kernel error: task suspended on another task's entry queue isn't in the suspended_at_call state.

[28] task already on run queue

internal kernel error: a suspended task that is going to be placed on the run queue is already on it.

[29] task on delay queue

internal kernel error: a suspended task that is going to be placed on the run queue is still on the delay queue.

[30] task already on delay queue

internal kernel error: a suspended task that is going to be placed on the delay queue is already on it.

[31] arith fault in rts

A hardware numeric exception (divide by zero, integer overflow) occurred while in the kernel, tdm, or an interrupt handler. Note: interrupt handler packages must be compiled with -S option and explicitly check for any conditions that may lead to a numeric exception. No internal exceptions are allowed or supported in kernel or tdm programs.

[32] calling task not in rendezvous or on run queue
internal kernel error: at the end of a rendezvous the calling task wasn't still in the in_rendezvous state.

[34] accepted task terminated or not in rendezvous
internal kernel error: during termination of the called task while in the in_rendezvous state, the calling task wasn't also in the in_rendezvous state.

[35] deadlock: no delays pending, no tasks ready, no signal entries active
You hung yourself. Re-evaluate your use of Ada tasking for timing problems.

[40] replace vector
internal kernel error: on UNIX, a call to SIGVEC was unsuccessful.

[41] invalid interrupt entry
The handler for an interrupt that occurred while in supervisor state (either in kernel or another interrupt handler) is trying to enter the kernel instead of doing an interrupt return. Interrupt handler should complete by calling the INTR_INTERFACE.COMPLETE_INTERRUPT kernel service which determines if it is possible to enter the kernel.

[42] already in kernel
The kernel has been entered from an interrupt handler package to perform a service that may only be performed for the user program. What probably happened is: a statement in a procedure called by an interrupt handler, when compiled, emitted a service request trap to the kernel (for example a **delay** or **task entry** call statement). Note: kernel services available to an interrupt handler package (REPLACE_VECTOR, CREATE_SIGNAL, POST_SIGNAL, COMPLETE_INTERRUPT) are restricted to the subprograms with specifications in the INTR_INTERFACE package.

[43] previously attached signal
Multiple interrupt entries have mapped to the same signal by using identical values in their for use at address clauses.

[44] invalid return to shadow task
internal kernel error: a shadow task created for an interrupt signal (which simply holds tasking information for doing a call to its interrupt entry) was erroneously put on the run queue.

[45] invalid interrupt to attach
On UNIX: the value in an interrupt entry's for use at address clause

was not in the range 0 to 32, was the SIGALRM value (#14) or was used by another interrupt entry.

[46] invalid interrupt to detach
internal kernel error: when the task containing an interrupt entry terminated, the kernel had an incorrect value for its signal.

[47] bad signal
received signal with no corresponding interrupt entry.

[48] no such signal
the value in an interrupt entry's for use at address clause wasn't within 1..32, was the SIGALRM value (#14) or was used by another interrupt entry.

[50] no coprocessor
NOT USED

[ ] unexpected exception
An unexpected exception has been mapped to the kernel's exception handler.

[ ] out of kernel heap
No more memory is available in kernel heap area for allocating task control records or signal structures. Increase the KRN_HEAP_SIZE kernel configuration parameter or reduce the number of tasks or signals.

[ ] Unhandled interrupt vector: 16#__#
Received an unexpected exception or external hardware interrupt. The hexadecimal number specifies the vector/interrupt table index and not the vector offset. You probably installed your interrupt handler in the wrong vector location or you enabled interrupts for a device without installing its handler.

[ ] panic: hardware exception while in RTS
On UNIX, received SIGSEGV or SIGFPE signal while in kernel.

[ ] exception handler: unexpected fpe_code *n*
On UNIX, The FPE_CODE (in decimal) associated with a SIGFPE signal, wasn't on the list of known hardware exceptions.

[ ] exception handler: unexpected signal *n*
On UNIX, an unexpected signal was mapped to the kernel's exception handler.

[ ] panic: sigstack failed

On UNIX, the call to sigstack for setting up a special stack for signal handling failed.

[ ] exception handler: below bottom of user stack

User stack needs to be extended to support exception handling.

[ ] No coprocessor emulate support

For the 386: a floating point instruction was executed without a 80287 or 80387 coprocessor being present (or the kernel configuration parameter, HW_FLT_ENABLED, was set to FALSE.) Coprocessor emulation software isn't currently provided.

18.22 MEMORY ALLOCATION SUPPORT IN THE VADS RUNTIME

This section describes the support for dynamic memory allocation and deallocation in the VADS Runtime System (RTS) and describes the ways in which memory allocation support can be configured to suit application-specific requirements.

In Ada, memory is allocated at runtime in one of two basic ways: by using a stack or using a heap. Memory is allocated from a stack when a subprogram is activated. Memory is allocated from a heap when a program employs an allocator or when a program directly or indirectly calls a routine that allocates memory. For example, if a program uses tasks, the compiler will generate calls to a RTS task creation routine which allocates task storage from a heap. VADS supports user space allocation (allocators) independently of kernel-space allocation (tasks); this document first describes user-space *library* support and then *kernel* support.

For more information about allocators, see also Ada RM 4.8.

This implementation of the RTS goes beyond the strict Ada requirements in that it also supports pool-based allocation and deallocation of memory for user space. The user may also configure in custom allocation support.

18.22.1 Allocators

Consider a user program that contains an access type statically defined by the following.

```
type record_type is record
  field_one: integer;
  field_two: integer;
  ...
end record;
```

```
type access_to_record is access record_type;
v: access_to_record;
```

A user may create an object of type RECORD_TYPE dynamically at execution time and reference it through the variable V. In Ada, this operation is most commonly done by using an allocator construct, as illustrated below.

```
v := new record_type;
```

Such allocators reserve a contiguous block of heap memory for each new instance of RECORD_TYPE. That memory is presumably not used for anything except that RECORD_TYPE instance while that instance is active.

An object becomes active (allocated) by the above mechanism. It becomes inactive (deallocated) by a user action, typically a call to UNCHECKED_DEALLOCATION instantiated for the appropriate type. The VADS RTS will *not* automatically deallocate objects except in the specific cases of local pools and task objects, as described below.

In the above example, the block of memory must be large enough to provide storage for every field that can legally (after constraint checks) be referenced through variable V. In practice, it may be somewhat larger, with extra space reserved for internal information used by the RTS and/or padding added to small allocations to prevent fragmentation of the heap.

An allocator is implemented as an implicit call to an RTS routine AA_GLOBAL_NEW.

```
new_object_address :=
  AA_GLOBAL_NEW(storage_units_needed_for_record_type);
```

The VADS RTS supplies a default implementation for AA_GLOBAL_NEW, as well as for the other memory allocation utilities discussed below. The VADS default is contained in the VADS library and will be linked with the user program unless the user explicitly supplies an alternative implementation. If the user includes a module that defines the name AA_GLOBAL_NEW, that definition will supersede the default. This mechanism, part of configuration control, will be described in detail later. For the moment, though, it is important only to know that users may redefine the memory allocation implementation and that VADS provides a default implementation (in some cases more than one) described here.

18.22.2 VADS Library Memory Management Semantics

The user-space memory management interface defines a set of subprograms that are called by compiler-generated code and the VADS library itself. This section describes when these subprograms are called and the function each performs. VADS supplies the following specification for its memory management interface.

```
with system; use system;
package aa_global is

  function aa_global_new(size : in integer) return address;
  pragma interface(UNCHECKED, aa_global_new, "AA_GLOBAL_NEW");

  procedure aa_global_free(a: in address);
  pragma interface(UNCHECKED, aa_global_free, "AA_GLOBAL_FREE");

  function aa_local_new(size: in integer; lheap: address)
  return address;
  pragma interface(UNCHECKED, aa_local_new, "AA_LOCAL_NEW");

  procedure aa_local_free(a: in address; lheap: address);
  pragma interface(UNCHECKED, aa_local_free, "AA_LOCAL_FREE");

  procedure init;
  pragma interface(UNCHECKED, init, "AA_INIT");

end;
```

AA_GLOBAL_NEW — A call to AA_GLOBAL_NEW (*size*) returns a pointer (address) to a contiguous block of at least *size* STORAGE_UNITS. The space may be located anywhere in memory, but the storage should not be in use for any other purpose. It is assumed that the *size* may be rounded upwards and that the allocated memory block might be larger than requested.

The compiler generates calls to AA_GLOBAL_NEW to implement most Ada allocators. In general, the compiler generates calls to AA_GLOBAL_NEW for each use of an allocator unless the corresponding access type is defined directly or indirectly within a subprogram *and* a

```
for access_type'storage_size use ...
```

representation clause has been given for the type. (The latter case is implemented by calls to AA_LOCAL_NEW as described below.)

Other utilities in the VADS library call AA_GLOBAL_NEW to allocate task blocks and task stack areas. Task information structures are deallocated explicitly by the kernel, as described below.

AA_GLOBAL_FREE — AA_GLOBAL_FREE supports deallocation of objects allocated by AA_GLOBAL_NEW. Its parameter must be a pointer that was previously returned by AA_GLOBAL_NEW. Passing an arbitrary address to AA_GLOBAL_FREE is erroneous and can lead to unpredictable results.

The predefined generic subprogram, UNCHECKED_DEALLOCATION is implemented with calls to AA_GLOBAL_FREE if allocators for the type call AA_GLOBAL_NEW. **See also** AA_LOCAL_FREE below.

Note: Instances of unconstrained array objects are implemented as an access to the start of the array with a dope vector prepended. If such an object is passed to UNCHECKED_DEALLOCATE, it will look past the dope vector to find the allocation size stored by AA_GLOBAL_NEW. Therefore, *do not use* UNCHECKED_CONVERSION or the address attribute in supplying a value directly to AA_GLOBAL_FREE for these instances.

Access variables must not refer to deallocated objects. The compiler does not verify that access variables refer to allocated objects. The RTS and the compiler expect that any storage freed by AA_GLOBAL_FREE can be reused for other purposes.

AA_LOCAL_NEW — Access types declared locally in a procedure are given special treatment if they have a STORAGE_SIZE representation specification. In these cases, the elaboration of the access type declaration will cause a request for a contiguous block of the required size to be pushed onto the procedure stack along with a descriptor for the pool. The descriptor is defined by the following record.

```
type local_heap_t;
type a_local_heap is access local_heap_t;
type local_heap_t is record
  size: integer;              -- size of the pool in storage_units
  head: address;              -- first storage_unit of pool
  next: address;              -- next available storage_unit of pool
  free_list: address;         -- free list populated by AA_LOCAL_FREE
end record;
```

Allocators for local pool access types are implemented as calls to AA_LOCAL_NEW. The *size* parameter of AA_LOCAL_NEW specifies the size in storage units of the object to be allocated. The LHEAP parameter is a pointer of type A_LOCAL_HEAP to the appropriate local pool descriptor for that access type.

The RTS does not require that a local pool be used, but it will save the indicated amount of storage in any case.

AA_LOCAL_FREE — The VADS compiler will generate calls to AA_LOCAL_FREE for uses of UNCHECKED_DEALLOCATION applied to an access type whose objects are allocated by means of AA_LOCAL_NEW. It is assumed that the object pointer passed to AA_LOCAL_FREE will be a pointer supplied by AA_LOCAL_NEW; this is not checked, and programs that disobey this rule are erroneous.

AA_INIT — In the VADS memory allocation default implementations, a call to AA_INIT must precede *any* call to AA_GLOBAL_NEW. Users may supply their own implementations that may relax this rule. The default RTS does call AA_INIT as essentially its first operation. Note that the AA_INIT routine should *not* involve calls to any other RTS service, as these will not yet be initialized.

18.22.3 VADS-Supplied Memory Management

VADS supplies two implementations of memory management. The default is called HEAP_ALLOC, and the other is called POOL_ALLOC.

18.22.3.1 Simple Allocation: HEAP_ALLOC

The default memory management support is called heap allocation. It is supplied in a package called HEAP_ALLOC.

In heap allocation, all calls to AA_GLOBAL_NEW allocate space from one global heap. For a request of N storage units, AA_GLOBAL_NEW allocates at least N storage units from the global heap. The number of storage units allocated is rounded up to a multiple of four.

A single integer value is also allocated to hold the size of the object being allocated. This value is located adjacent to the allocated object, immediately preceding it. AA_GLOBAL_FREE uses this value to determine the amount of storage to deallocate.

In heap deallocation, AA_GLOBAL_FREE links the deallocated storage onto a free list. The link value is placed in the deallocated object so the minimum size of an object is always rounded up to the size of an access variable.

The minimum size of an allocation is the number of storage units required to hold an integer for the object's size and an access variable for its free list link.

AA_GLOBAL NEW searches the free list for an area large enough to satisfy each storage request as it occurs. A 'first fit' algorithm is used.

Heap allocation uses the user configurable variable ALLOWED_WASTE. If an area in the free list is larger than the user request by no more than ALLOWED_WASTE storage units, the entire object will be given to the user even though the extra storage will be wasted. However, if the waste is larger than ALLOWED_WASTE, the remaining storage will be split into a new free area.

The intent of ALLOWED_WASTE is to reduce memory fragmentation, whereby the free list becomes clogged with numerous free areas too small to satisfy memory requests. It is recommended that ALLOWED_WASTE be configured to be in the range of 64 to 128 storage units. However, in no case may it be smaller than the size of an integer plus the size of an address.

The default value for ALLOWED_WASTE is 31 storage units. To re-configure this value, **package** HEAP_CONFIG in the file **unx_heap_cfg.a** (located in **user_config** directory) must be changed and then linked with the program. After the package is modified and recompiled, it can be included in a program either by a **with** clause in the Ada source or a WITH*n* directive in the **ada.lib** file.

See also *INSTALLATION GUIDE, 17.4 CONFIGURING THE VADS RUNTIME SYSTEM,* page 17-7.

In heap allocation, AA_LOCAL_NEW and AA_LOCAL_FREE are implemented exactly as for AA_GLOBAL_NEW and AA_GLOBAL_FREE above. The only difference is that memory will always come from the assigned storage area on the procedure stack instead of from the global heap.

18.22.3.2 Pool-Based Allocation: POOL_ALLOC

Verdix also supplies POOL_ALLOC, an optional pool-based memory allocation implementation. This implementation is considerably larger and more complex than HEAP_ALLOC and delivers more functionality. The increased functionality is in two areas. First, allocation of small objects can be tuned using 'lists'. A list links together storage elements of the same size. Lists avoid fragmentation and eliminate use of the size integer for each object. Second, memory can be separated into multiple storage pools. All objects allocated from a particular pool can be deallocated simultaneously with a single RTS call.

With the exception of compaction, the additional memory control afforded by the use of global pools does not extend to local pools. The RTS performs compaction on a local pool when its storage becomes exhausted, to combine free storage areas and possibly satisfy the request.

POOL_ALLOC employs a heap. The heap itself is a global pool, from which small object lists, global pools, and other objects may be allocated.

Small Object Lists — Space for small object lists is allocated from the heap. Users can configure up to 10 lists, each with a different element size. Each list will contain elements of the same size. A small object will then be allocated from the list whose elements are just large enough to store the object. For example, if lists are established for objects of size 16, 32, and 128 STORAGE UNITs, requests of up to 16 storage units will come from the first list, requests for between 17 and 32 storage units from the second list, and requests for between 33 to 128 storage units from the third list. Objects larger than 128 STORAGE_UNITs are allocated from the heap.

Small object lists are used for allocators only when space is requested from the heap pool. (The section *Pool Control* below shows how memory requests can be directed to a particular global pool.) They are not used for allocation requests that apply to local pools or other global pools. UNCHECKED_DEALLOCATION is used in the normal way to free space occupied by small objects. The compiler will generate a call to AA_GLOBAL_FREE to implement UNCHECKED_DEALLOCATION. The space allocated to a small object is always returned to the list from which it came.

Small object list memory areas are isolated from each other and from other memory areas. Once memory is reserved for a small object list, it will not be used for any other purpose. If an object is to be allocated from the heap, and a small object list has been defined for objects of its size, but the object list is empty, the RTS automatically extends the size of the object list by allocating additional space from the heap. The initial size of an object list and its granularity of extension is not configurable.

The number of small object lists can be configured to zero, in which case they are not used.

The rationale for using small objects is that, if the user knows in advance that most allocations are of a specific size, allocation can be facilitated for that size. Overall performance can therefore improve both in terms of memory usage. List-based allocation tends to be slower if many individual deallocations are made.

Also, list-based allocation is slower if the sizes of objects are truly random. Thus, it should be used only if the user can know ahead of time that many allocation requests will be of a specific size or group of sizes.

To configure list-based allocation into the RTS, the user needs to include in the Ada library a WITH*n* INFO directive naming the Verdix-supplied **pool_alloc.a** library archive file located in the **standard** library (*VADS_location*/**standard/.objects/pool_alloc.a**). This WITH*n* line will cause the linker to include the pool-based implementation, rather than that provided in the default implementation HEAP_ALLOC.

To configure individual list element sizes, **package** POOL_CONFIG in the file **unx_pool_cfg.a** (located in the **user_config** directory) must be changed and then linked with the program.

After the package is modified and recompiled, it can be included in a program either by a **with** clause in the Ada source or a WITH*n* directive in the **ada.lib** file.

Sample configuration parameters for pool-based memory allocation are shown below. The example assumes that the user will allocate many objects of size 24 and many more of size 40.

package POOL_CONFIG

```
------------------------------------------------------------------------
--  The following entities are used only if list-based storage        --
--  management is used. List management handles small objects         --
--  specially so as to reduce overhead and fragmentation              --
--                                                                    --
--    NUM_SMALL_BLOCK_SIZES: Gives number of small object sizes       --
--       to be handled by lists.                                      --
--                                                                    --
--    SMALL_BLOCK_SIZES: Lists the small block sizes for the          --
--       lists. If num_small_block_sizes = 3, then (8, 16, 128)       --
--       means that the three small object pools will be sliced       --
--       into subobjects of 8 bytes, 16 bytes, and 128 bytes. For     --
--       example, user objects of size 20 would be allocated as       --
--       128 byte small objects. Sizes MUST BE MULTIPLES OF 8.        --
--       Sizes MUST BE IN ASCENDING ORDER.                            --
--                                                                    --
--    ALLOWED_WASTE: For larger objects, gives the size at which      --
--       an over-large space will be broken into a perfect fit        --
--       and a new free storage block.  For example, a user asks      --
--       for 1000 bytes and the next free slot has 3000 bytes; if     --
--       allowed waste < 2000, the user will get exactly 1000         --
--       of the free space, and the remainder (2000 bytes less        --
--       overhead of 4 bytes) will be put back on the free list.      --
--       If allowed_waste >= 2000 then the user gets 3000 bytes.      --
--       This controls fragment sizes.                                --
--                                                                    --
--    GET_MEMORY: Requests more memory for the heap.                  --
--                                                                    --
--  If the pool based allocation mechanism is not used, these         --
--  declarations can be deleted.                                      --
--                                                                    --
------------------------------------------------------------------------
max_small_block_sizes: constant := 10;

num_small_block_sizes: integer := 2;
pragma external_name(num_small_block_sizes, "NUM_SMALL_BLOCK_SIZES");

small_block_sizes: array(1..max_small_block_sizes) of integer :=
```

```
 (24, 40);
pragma external_name(small_block_sizes, "SMALL_BLOCK_SIZES");

allowed_waste: integer := 128;
pragma external_name(allowed_waste, "ALLOWED_WASTE");

do_compact: boolean :=true;
pragma external_name(do_compact "DO_COMPACT")

procedure get_memory(size: integer; first_addr, last_addr: OUT
 address; alloc_sema:semaphores.a_semaphore);
pragma external_name(get_memory, "GET_MEMORY");
-- in body, most bare systems might define this as:
-- begin
--     raise STORAGE_ERROR;
-- end;
```

The GET_MEMORY procedure defined in the configuration package is called by the RTS when the heap memory becomes exhausted. Typically, GET_MEMORY is called when AA_GLOBAL_NEW fails to find sufficient heap space to satisfy a memory request. In this case, the value of the size parameter for GET_MEMORY is set to that for AA_GLOBAL_NEW. It is recommended that GET_MEMORY return a much larger block of memory than is requested. In this way, excessive use of GET_MEMORY is avoided. The block is returned by setting FIRST_ADDRESS to its low address and LAST_ADDRESS to its hightes address.

For versions on host operating systems such as UNIX, GET_MEMORY does a kernel call to obtain more memory from the kernel. The ALLOC_SEMAPHORE parameter to GET_MEMORY is discussed below.

18.22.3.3 Pool Control

Besides list allocation, POOL_ALLOC provides additional facilities aimed at more convenient and efficient control of memory allocation. These pool control facilities enable a program to create and deallocate global pools of memory and to direct its memory requests to specific global pools.

The only appropriate way to deallocate objects allocated from pools is to deallocate the entire pool. Any use of AA_GLOBAL_FREE to free an individual object allocated from a pool will result in STORAGE_ERROR if the compact routine (described below) is called.

CREATE_POOL

```
function create_pool(initial_size, extension_size: integer;
  base: address := no_addr) return a_pool_descr;
  pragma external_name(create_pool, "CREATE_POOL");
```

CREATE_POOL creates an internal data structure for a pool and returns a descriptor or identifier for the pool. Pools generally use contiguous memory where possible to prevent fragmentation. INITIAL_SIZE gives the number of storage units to allocate to the pool initially. Pools will grow as necessary, allocating from the heap in lots of EXTENSION_SIZE STORAGE_UNITS. If EXTENSION_SIZE is zero, the pool will not be extended and exhaustion will raise STORAGE_ERROR. By specifying the base address of a pool, a user can assign a specific region of the heap to the pool, perhaps to *prevent* its use by the normal allocation process. The default NO_ADDR causes the base to be arbitrary, as the pool storage block is obtained from an arbitrary storage area in the heap. The RTS creates the heap pool at startup time in AA_INIT.

DEALLOCATE_POOL

```
procedure deallocate_pool(p: a_pool_descr);
  pragma external_name(deallocate_pool, "DEALLOCATE_POOL");
```

DEALLOCATE_POOL causes all blocks of storage that have been obtained for the named pool to be returned to the heap's free list. Such memory is no longer reserved for that pool and may be used for any heap activity. The heap pool can never be deallocated.

SWITCH_POOL

```
procedure switch_pool(p: a_pool_descr);
  pragma external_name(switch_pool, "SWITCH_POOL");
```

For performance reasons, AA_GLOBAL_NEW does not use an explicit parameter to specify the pool used for allocation requests. Instead, it uses an implicit parameter defined by the CURRENT_POOL. At system startup, the heap is the current pool. However, by calling SWITCH_POOL, a program can select an arbitrary pool as the current pool. *All* allocations using AA_GLOBAL_NEW are obtained from the current pool, even if they arise from calls to 'hidden' routines like the TEXT_IO buffer allocations. A program must therefore be careful that all objects are indeed disposable before freeing the pool.

CURRENT_POOL

```
function current_pool return a_pool_descr;
  pragma external_name(current_pool, "CURRENT_POOL");
```

CURRENT_POOL returns the pool identifier for the current pool.

AA_POOL_NEW

```
function aa_pool_new(pool: a_pool_descr; size: integer) return address;
  pragma external_name(aa_pool_new, "AA_POOL_NEW");
```

AA_POOL_NEW allocates space from the named pool. It is equivalent to the following program fragment.

```
save := current_pool;
switch_pool(pool);
result := aa_global_new(size);
switch_pool(save);
return result;
```

HEAP_POOL

```
function heap_pool return a_pool_descr;
  pragma external_name(heap_pool, "HEAP_POOL");
```

HEAP_POOL returns the pool identifier for the heap pool. One might wish to switch to the HEAP_POOL to allocate objects that must persist.

COMPACT

```
procedure compact;
  pragma external_name(compact, "COMPACT");
```

Unlike simple memory allocation, POOL_ALLOC supports free list compaction. Compaction is expensive: it amounts to a sort of a pool's free list by address, followed by a traversal to coalesce all adjacent memory areas. Compaction can remedy fragmentation. It is called automatically if storage is exhausted. Users may wish to call it explicitly if they want to avoid a random long delay when an arbitrary allocation exhausts memory.

DO_COMPACT

Compaction is expensive and may occur at arbitrary times. The DO_COMPACT boolean is supplied to turn off automatic compaction. Instead of compacting to get more free space for a pool, a new block is allocated to the pool by GET_MEMORY.

18.22.3.4 Suggestions for Use of VADS Memory Allocation

Simple allocation is generally suited to most applications for which deallocation is straightforward or unimportant. Deallocation, in HEAP_ALLOC, requires that the user be able to find every object that requires deallocation and call an UNCHECKED_DEALLOCATION instantiation to free it. If there are many freed objects of small size with many allocations of random size, the free list may become clogged with many small objects, and memory will become fragmented.

The more complicated POOL_ALLOC implementation is generally slower than simple allocation. It can be faster if the process lasts long enough to collect and reuse a lot of deallocated storage. The small object lists can prevent fragmentation but will cost time and space if the user does not know in advance which object sizes to use.

Pool-based memory deallocation has been demonstrated to be one of the easiest memory deallocation mechanisms to use Programmers tend to have a general feel for how long their objects will be needed, and so can define a set of pools for each lifetime. They do not need to be able to walk through their data structures to locate garbage and dispose of it explicitly, object by object. Pool-based allocation should *not* be used where object lifetimes are random or unpredictable.

18.22.3.5 Mutual Exclusion During Allocation

User-space memory allocation, as implemented in the Verdix-supplied HEAP_ALLOC and POOL_ALLOC packages, is protected as a critical region via semaphores. The package defines a semaphore which must be obtained before a thread can enter the allocation/deallocation code. The semaphore is blocked, the task must queue itself on that semaphore's wait list.

The semaphore protection allows allocation to proceed without entering the kernel. The kernel is itself a critical region, as the IN_RTS flag behaves as the semaphore. To prevent deadlock, we assure that the kernel never uses the user memory allocators. Otherwise, a user task could block in allocation and the kernel thread might become deadlocked.

Before return is made from allocation utilities such as AA_GLOBAL_NEW, the allocation semaphore must be released, indicating that other tasks may now queue up to get their allocation. We pass the allocation semaphore to GET_MEMORY so that if memory is truly exhausted and STORAGE_ERROR must be raised, then we can release the allocation semaphore before doing the raise.

18.22.3.6 Memory Management and Underlying Operating Systems

When a program runs on top of an underlying operating system such as UNIX, there may be requests for heap memory from other libraries (e.g., requests for buffers for disk transfers). It is imperative that memory allocations be internally consistent or these unknown buffers could be allocated in active program memory.

On UNIX, a single central allocation service (**malloc**(3), **free**(3)) is used by all programs. The VADS RTS preserves this behavior by implementing GET_MEMORY as a kernel call that in turn calls UNIX **malloc**(3). Ada programs are thus consistent: user allocators call AA_GLOBAL_NEW, which will probably satisfy the request directly and is protected by a semaphore. Otherwise, the get_memory utility calls the kernel AA_GLOBAL_NEW service which is protected by the kernel's prohibition against executing multiple single threads and the request is satisfied by a call to malloc(3).

To protect tasks that call programs written in other languages, we suggest that you use the semaphore mechanism or coding practice of serializing accesses, as appropriate. For example, we suggest that you perform all I/O from a single task. Of particular concern are I/O operations, because they do allocate large blocks of memory, either during file opens or during the first reads and writes to the files. To be absolutely safe, one must use the allocation semaphore to make critical regions around *all* calls to foreign language utilities.

NOTE: VADS does not currently protect its underlying calls to UNIX utilities. It is therefore possible that multi-tasking, time-sliced programs will run into conflicts with unpredictable results if tasks are allowed to suspend while in UNIX. We suggest that time-slicing (or other interrupts) be turned off for these tasks, or that semaphores be used to protect them, or that all such activities be confined to a single task at any time.

18.22.3.7 Replacing User-space Memory Allocation

Some users may have specialized performance needs that are not met by any of the Verdix supplied memory allocation packages. These users may wish to implement their own algorithms. These users should simply compile a new body for the AA_GLOBAL package specifications given earlier. Some considerations:

- If multiple tasks can do memory allocation simultaneously, the critical region of the user algorithms should be protected via semaphores.
- On a host operating system, be aware that standard memory allocators are not protected as critical regions and may be interrupted. Trouble will arise if another task enters the allocator before the interrupted task completes its allocation,. Also, calls to other languages may lead to unprotected calls for memory. For example, if TASK1 opens **file1** and TASK2 opens **file2**, both tasks will probably call the OS to get memory for buffers and so forth. If task1 is suspended during the allocation, unpredictable results may occur. VADS does *not* protect users from this effect.
- Packages in the files **iface_*.a**, are supplied to provide access to helpful library definitions in standard.

18.22.3.8 Memory allocation in the Kernel

The VADS kernel does its memory allocation independent of the user space allocation AA_GLOBAL. Kernel memory allocations are divided into two classes: allocation of private kernel structures by KRN_ALLOC, and allocation of user accessible data structures (such as task stacks) by TS_ALLOC. Verdix intends to allow only source customers to modify these key allocations.

The TS_ALLOC allocation and deallocation primitives may be called as kernel services. Indeed, if performance is no object, then all user space allocations can be done via these services, for minimum code size. In general, we expect only the GET_MEMORY routine to actually call the kernel allocation service.

The **package** IFACE_ALLOC defines the kernel's user-visible allocation primitives.

```
-- Copyright 1986, 1987 Verdix Corporation

-- User interface to kernel's memory allocation services.
-- Note that the compiler does NOT generate calls to these
-- routines, but instead calls AA_GLOBAL_NEW and
-- AA_GLOBAL_FREE.  These calls reach the -- user-accessible
-- memory allocators, and are typically used as a backstop to get
-- more memory in large chunks.

-- Note: krn_aa_global_new returns system.no_addr if
-- allocation was not possible.  It will not raise exceptions.

with iface_types;

package iface_alloc is

    function krn_aa_global_new(size: iface_types.alloc_t)
                return iface_types.user_address_t;
        pragma interface(UNCHECKED, krn_aa_global_new,
        "_ _AA_GLOBAL_NEW");

    procedure krn_aa_global_free(a:iface_types.user_address_t);
        pragma interface(UNCHECKED, krn_aa_global_free,
        "_ _AA_GLOBAL_FREE");

end iface_alloc;
```

These primitives have the same meaning as for user-space utilities except that these allocate under serialization by the IN_RTS kernel semaphore. Only one thread can execute *any* kernel activity at a given time. The heap of memory allocated by these utilities is set up during kernel initialization.

See also *INSTALLATION GUIDE, 17.4 CONFIGURING THE RUNTIME SYSTEM,* page 17-7.

18.23 VADS BASIC BIT MANIPULATION UTILITIES

Bit manipulation – full Chapter 13 support – is provided on every VADS compiler, so there is no real need for this package; we provide it as a service for those who are migrating from other compilers and for those who must deal with porting to other compilers.

This package does not use INLINE pragmas because of this same porting problem: other compilers do not support INLINE.

Verdix recommends that the normal Ada representation control be used to achieve bit manipulations unless portability is required.

18.23.1 The IFACE_BITS Package

The package was formerly called BIT_OPS in earlier VADS versions.

```
package iface_bits is

   function bit_and(l, r : integer) return integer;

   function bit_or(l, r : integer) return integer;

   function bit_xor(l, r : integer) return integer;

   function bit_neg(l : integer) return integer;

end;
```

18.23.2 Semantics of the IFACE_BITS Utilities

Each utility operates on the integer type of the target. The integer type is the size of the general purpose register(s) of the target. The operations yield the obvious logical bit–wise results.

18.24 VADS SEMAPHORE SERVICES

The VADS Runtime system (RTS) supports semaphores so that users can protect their critical code regions in concert with the other ongoing activities of the Ada runtime.

A critical region is an activity that must not be in execution by more than one thread at a time: if **task** A is interrupted while inside the critical region, it must not be possible for **task** B to begin execution of the critical region until **task** A has resumed and completed its execution of the critical region. **task** B must therefore have a means of detecting that **task** A is in the critical region, and have a means of waiting on that region. **task** A must have a means of telling any tasks like B that are waiting to execute the critical region that it has exited the region. The VADS RTS supplies semaphores to meet these needs.

18.24.1 The Semaphore Package

```
with system;
with iface_types;

package iface_sema is

null_queue:constant iface_types.a_task_t
                        := iface_types.null_task;

type semaphore_rec is record
   queue: iface_types.a_task_t
                        := null_queue;
   flag: iface_types.test_and_set_t
                        := iface_types.test_and_set_false;
   others_waiting: iface_types.test_and_set_t
                        := iface_types.test_and_set_false;
end record;
type a_semaphore is access semaphore_rec;

procedure enter(s: a_semaphore);
   pragma interface(unchecked, enter, "__ENTER");
procedure leave(s: a_semaphore);
   pragma interface(unchecked, leave, "__LEAVE");

procedure suspend(s: a_semaphore);
   pragma interface(unchecked, suspend, "__SUSPEND");
procedure resume(s: a_semaphore);
   pragma interface(unchecked, resume, "__RESUME");

end;
```

18.24.2 Semaphore Structures

Semaphores are implemented as data structures created in user space. They have three fields:

QUEUE — points to a list of tasks that are waiting on this semaphore. The form of this field is machine-dependent.

FLAG — the classic 'test-and-set' field, its value is either TEST_AND_SET_TRUE or TEST_AND_SET_FALSE. Ada's boolean values, TRUE and FALSE, are not used because VADS uses machine-dependent atomic test-and-set operations giving machine-dependent values.

OTHERS_WAITING — also a test-and-set field, used to indicate when other tasks are suspended on the queue. This field will be discarded in later VADS releases.

18.24.3 Creating Semaphores

Semaphore data structures are created with an allocator, as with any other data structure in Ada, typically:

```
my_sema := new iface_sema.semaphore_rec;
```

It is unlikely that users would have occasion to supply values other than the defaults for the SEMAPHORE_REC fields—that would mean initializing the semaphore as if some task were already in the critical region.

18.24.4 Defining a Critical Region

A critical region begins with a call to the **procedure** IFACE_SEMA.ENTER and ends with a call to **procedure** IFACE_SEMA.LEAVE. It is important that *every* entrance path to the region calls ENTER, and that *every* exit path from the region calls LEAVE. If a task can get in without calling ENTER, then another task could also get in simultaneously. If a task exits the region without doing a LEAVE then the critical region will appear to be still in use: no other task will be able to do an ENTER. This usually results in a deadlock condition. Exception paths, as well as normal returns, must be handled:

```
-- a typical critical region

... non-critical code

   iface_sema.enter(my_sema);

   ... critical region code

   if bad_things then
      -- tricky exit: free semaphore
      iface_sema.leave(my_sema);
      raise bad_things_happened;
   end if;

   ... more critical region code


   exception
      when others =>
         iface_sema.leave(my_sema);
         raise bad_things_happened;

... more non-critical code
```

18.24.5 Semaphore Semantics

The **procedure** ENTER performs a machine-dependent test-and-set operation on S.FLAG. If S.FLAG is TEST_AND_SET_TRUE then some task is already in the critical region and so the RTS kernel's SUSPEND is called to queue the current task onto S.QUEUE. Priority is unimportant on this queue because all tasks will be reactivated and sorted by priority by the scheduler. Otherwise, S.FLAG becomes TEST_AND_SET_TRUE and the current task continues into the critical region.

Execution of LEAVE assigns TEST_AND_SET_FALSE into S.FLAG, freeing the semaphore for other tasks. It then checks S.OTHERS_WAITING and calls the kernel's RESUME to reactivate all the tasks currently queued for that semaphore.

18.24.6 Kernel SUSPEND and RESUME

The kernel is entered if a task must SUSPEND itself or if S.OTHERS_WAITING is TRUE and RESUME must be called.

SUSPEND and RESUME are not necessarily usable ONLY from semaphores. One could use them for direct SUSPEND/RESUME coprocessing, as long as it is understood that SUSPENDed tasks cannot be reactivated except via a RESUME. For example:

```
my_control: iface_sema.a_semaphore := ...;

task body A is
   ...
   iface_sema.SUSPEND(my_control);      -- task A stops itself

   ... code for execution after event X

end A;

task body B is

   ... event X happens here

   iface_sema.RESUME(my_control);       -- task A (at least) can

   ... code for later execution         -- execute again

end A;
```

The SUSPEND and RESUME flow control model can be implemented using normal Ada task entries, and Verdix recommends that approach. However, SUSPEND and RESUME can be useful in cases where task A and task B do not know about each other, or where there are many task As or task Bs.

18.25 VADS TASK CONTROL SERVICES

The dynamic performance of tasking may be controlled independently of the code in an individual task. Tasks can be time sliced. An execution time limit is set on the length of time that a given task will execute before the scheduler will again be given control.

Each task is given a time slice value and a priority at the time it is created. The time slice value is determined in a runtime system configuration file. That default may be changed dynamically by SET_TIME_SLICING, discussed below.

See also *INSTALLATION GUIDE, 17.4 CONFIGURING THE VADS RUNTIME SYSTEM*, page 17-7.

Initial task priority is set in the calls generated to create each task, that is, at compile-time. It is controlled by the **pragma** PRIORITY. Task priority can be set dynamically by SET_PRIORITY, below.

18.25.1 The IFACE_TASKS Package

```
with IFACE_TYPES;
use  IFACE_TYPES;
with SYSTEM;

package IFACE_TASKS is

-- The following kernel services get/set current task's user
-- field.

  function GET_TASK(TASK_VAR_ADDR : SYSTEM.ADDRESS)
                    return A_TASK_T;
  pragma INTERFACE(UNCHECKED, GET_TASK, "_ _GET_TASK");

  function GET_CURRENT_TASK return A_TASK_T;
  pragma INTERFACE(UNCHECKED, GET_CURRENT_TASK, "_ _GET_CURRENT_TASK");

  function GET_USER_FIELD(A_TASK : A_TASK_T) return SYSTEM.ADDRESS;
  pragma INTERFACE(UNCHECKED, GET_USER_FIELD, "_ _GET_USER_FIELD");

  procedure SET_USER_FIELD(A_TASK : A_TASK_T;
                           NEW_VALUE : SYSTEM.ADDRESS);
  pragma INTERFACE(UNCHECKED, SET_USER_FIELD,"_ _SET_USER_FIELD");


  function GET_PRIORITY(A_TASK : A_TASK_T)
                        return SYSTEM.PRIORITY;

 pragma INTERFACE(UNCHECKED, GET_PRIORITY, "_ _GET_PRIORITY");

  procedure SET_PRIORITY(A_TASK : A_TASK_T;
                         NEW_VALUE : SYSTEM.PRIORITY);
  pragma INTERFACE(UNCHECKED, SET_PRIORITY, "_ _SET_PRIORITY");

  procedure SET_TIME_SLICE(A_TASK : A_TASK_T;
                           NEW_TIME_SLICE : TIME_T);
  pragma INTERFACE(UNCHECKED, SET_TIME_SLICE, "_ _SET_TIME_SLICE");

  function GET_TIME_SLICE(A_TASK : A_TASK_T) return TIME_T;
  pragma INTERFACE(UNCHECKED, GET_TIME_SLICE, "_ _GET_TIME_SLICE");

  procedure SET_TIME_SLICING(ENABLE : BOOLEAN;
                             DEFAULT_SLICE : TIME_T);
  pragma INTERFACE(UNCHECKED, SET_TIME_SLICING, "_ _SET_TIME_SLICING");

  procedure GET_TIME_SLICING(ENABLED       : out BOOLEAN;
                             DEFAULT_SLICE : out TIME_T);
  pragma INTERFACE(UNCHECKED, GET_TIME_SLICING, "_ _GET_TIME_SLICING");

end IFACE_TASKS;
```

function GET_TASK

```
GET_TASK(TASK_VAR_ADDR: SYSTEM.ADDRESS) return A_TASK_T
```

Tasks are represented by private type values. Normally these values are the address of a task control data structure in the kernel address space. The **package** IFACE_TASKS is used to formally obtain a value of this private type, given the address of the task. Thus, given a task or task variable T, then GET_TASK(T'ADDRESS) returns a value of **type** A_TASK_T for T.

function GET_CURRENT_TASK

```
GET_CURRENT_TASK return A_TASK_T
```

The currently executing task value is returned.

function GET_PRIORITY

```
GET_PRIORITY(A_TASK: A_TASK_T) return SYSTEM.PRIORITY
```

Each task maintains a static priority and a dynamic priority. The static priority is the basic priority of that task as initialized by the task creation parameters set up at compile time, or as set at run time by SET_PRIORITY. The dynamic priority is the current execution priority of the task. The current execution priority is the maximum of the static priority of this task and the maximum dynamic priority of all the tasks in rendezvous with this task. The GET_PRIORITY routine returns the dynamic priority.

procedure SET_PRIORITY

```
SET_PRIORITY(A_TASK: A_TASK_T; NEW_VALUE: SYSTEM.PRIORITY)
```

This routine sets both the static and dynamic priority of the given task to NEW_VALUE. Note that the change in priority will not take effect until this task is again queued on the run queue; that is, the run queue is not sorted again to reposition this task. Note that this priority may interfere with Ada semantics. It is therefore not recommended by Verdix, and is given only at the insistence of our customers.

function GET_TIME_SLICE

```
GET_TIME_SLICE(A_TASK: A_TASK_T) return TIME_T
```

Returns the current time slice value for **task** A_TASK. This is the value that is to be used for establishing the next time slice for A_TASK, and has no relation to the time remaining in the current time slice, should A_TASK be the CURRENT_TASK.

procedure SET_TIME_SLICE

```
SET_TIME_SLICE(A_TASK: A_TASK_T; NEW_TIME_SLICE: TIME_T)
```

Time slicing effects the sharing of the CPU(s) among tasks having equal dynamic priorities. The next time this task runs (and TIME_SLICING_ENABLED is true, as described below) an alarm will be scheduled to interrupt this task after NEW_TIME_SLICE MILLISECONDS. Note that the current time slice (if A_TASK is the current task) is not affected. For example,

```
SET_TIME_SLICE(GET_TASK(FOO'ADDRESS), 500)
```

causes task FOO to be time sliced in half-second slices.

procedure GET_TIME_SLICING

```
GET_TIME_SLICING(ENABLED: out BOOLEAN; DEFAULT_SLICE: out TIME_T)
```

If enabled is true, then tasks are currently being time sliced. The DEFAULT_SLICE gives the time slice to be assigned to any new tasks; it is not related to the slice of any existing task, except by history.

procedure SET_TIME_SLICING

```
SET_TIME_SLICING(ENABLE: BOOLEAN; DEFAULT_SLICE: TIME_T)
```

If ENABLE is assigned false, then no further time slicing will be done although the current time slice will expire normally; if true, then time slicing will continue with the next switch to a new task. The value of DEFAULT_SLICE will be used for any future tasks created.

18.25.2 Controlling Tasks While Debugging

These routines may be used to control tasks during debugging, but only if the routines are linked into and available in the executable; it is sufficient to with this package in some active unit of this executable.

See also *LIBRARY TOOLS, 4.7.3* **a.info**, page 4–6.

For debugging, it is recommended that time slicing be disabled whenever possible because it often makes the debugging process more difficult. Specifically, whenever a breakpoint occurs, the time slice will usually expire while the user is interacting with the debugger. As a result, the next go command (or single step) will trigger an immediate task switch.

18.25.3 User Control of Tasking

Verdix recognizes that users may wish to control tasking in unusual ways, and to help in that decision process, VADS allows users to associate arbitrary information with the VADS task control data structures.

function GET_USER_FIELD

```
GET_USER_FIELD(A_TASK: A_TASK_T) return SYSTEM.ADDRESS
```

The most recent value assigned, by SET_USER_FIELD, to A_TASK is returned. If no such value has been assigned, the result is arbitrary.

procedure SET_USER_FIELD

```
SET_USER_FIELD(A_TASK: A_TASK_T; NEW_VALUE: SYSTEM.ADDRESS)
```

The NEW_VALUE is internally bound to the task named by A_TASK. Any previous value is lost.

18.26 VADS EXCEPTION REPORTING

Ada defines a small number of standard exceptions, each of which can be raised by a wide variety of circumstances. Users cannot normally differentiate between the various causes of a particular exception.

The VADS debugger uses additional background information supplied by the runtime system and the compiler. The compiler assigns an integer value representing the circumstance surrounding an exception into a variable called __RTS_MSG. The debugger reads this value after any unhandled exception and uses it as an index into a table of messages. The exception name and the message can then be displayed by the debugger.

18.26.1 Package IFACE_RAISE

The following Ada package gives the correspondence table for exception subcode description numbers used by the debugger. The type XCPT_T is used as an enumeration, but is represented as an array of constants to provide continuity with earlier implementations.

```
with SYSTEM;
package IFACE_RAISE is

-- Exception subcodes used by debugger.
  type XCPT_T is range 0 .. 69;
  for XCPT_T'SIZE use 8;

  UNDEFINED_XCPT          : constant XCPT_T := 0;
  --
  VALUE_LBOUND_XCPT       : constant XCPT_T := 1;
  --  RM 3.5: value less than lower bound of subtype
  VALUE_UBOUND_XCPT       : constant XCPT_T := 2;
  --  RM 3.5: value greater than upper bound of subtype
  INDEX_LBOUND_XCPT       : constant XCPT_T := 3;
  --  RM 4.1.1(4): index value less than lower bound of index
  --  subtype
  INDEX_UBOUND_XCPT       : constant XCPT_T := 4;
  --  RM 4.1.1(4): index value greater than upper bound of index
  --  subtype
  SLICE_LBOUND_XCPT       : constant XCPT_T := 5;
  --  RM 4.1.2(4): lower bound of slice is out of range
  SLICE_UBOUND_XCPT       : constant XCPT_T := 6;
  --  RM 4.1.2(4): upper bound of slice is out of range
  STR_LEN_XCPT            : constant XCPT_T := 7;
  --  RM 4.2(3): lower bound of null string literal has no
  --  predecessor
  AGG_LBOUND_XCPT         : constant XCPT_T := 8;
  --  RM 4.3.2(11): lower bound of array aggregate is out of
  --  range
  AGG_UBOUND_XCPT         : constant XCPT_T := 9;
  --  RM 4.3.2(11): upper bound of array aggregate is out of
  --  range
  CAT_LEN_XCPT            : constant XCPT_T := 10;
  --  RM 4.5.3(6): upper bound of catenation is out of range
  ARRAY_SIZE_XCPT         : constant XCPT_T := 11;
  --  RM 5.2.1(2): arrays have different sizes
  INDEX_MATCH_XCPT        : constant XCPT_T := 12;
  --  RM 3.6.1: bounds of index subtypes don't match
  NO_PRED_XCPT            : constant XCPT_T := 13;
  --  RM 3.5.5(9): value in PRED attribute has no predecessor
  NO_SUCC_XCPT            : constant XCPT_T := 14;
  --  RM 3.5.5(8): value in SUCC attribute has no successor
  VAL_ATTR_LBOUND_XCPT    : constant XCPT_T := 15;
  --  RM 3.5.5(7): result of T'VAL is less than lower bound of
```

```
   --  T'BASE
   VAL_ATTR_UBOUND_XCPT     : constant XCPT_T := 16;
   --  RM 3.5.5(7): result of T'VAL is greater than upper bound of
   --  T'BASE
   DELTA_XCPT               : constant XCPT_T := 17;
   --  RM 3.5.9(13): value of DELTA is not compatible with the
   --  subtype
   DIGITS_XCPT              : constant XCPT_T := 18;
   --  RM 3.5.7(14): value of DIGITS is not compatible with the
   --  subtype
   NEG_EXP_XCPT             : constant XCPT_T := 19;
   --  RM 4.5.6(6): negative exponent in integer exponentiation
   ACCURACY_XCPT            : constant XCPT_T := 20;
   --  RM 11.1(6): universal integer cannot be accurately
   --  represented
   DECL_ELAB_XCPT           : constant XCPT_T := 21;
   --  RM 9.3(6): exception raised during elaboration of task
   TASK_ELAB_XCPT           : constant XCPT_T := 22;
   --  RM 11.1(7): attempt to activate a task before it is
   --  elaborated
   DIV_BY_0_XCPT            : constant XCPT_T := 23;
   --  RM 4.5.5(12): divide by zero
   DSCRMT_XCPT              : constant XCPT_T := 24;
   --  RM 3.7.2(6): discriminant value incompatible with
   --  constraint
   VARIANT_XCPT             : constant XCPT_T := 25;
   --  RM 4.1.3(8): component in variant part cannot be selected
   TAG_XCPT                 : constant XCPT_T := 26;
   --  RM 4.1.3(8): component in variant part cannot be selected
   MATCH_LBOUND_XCPT        : constant XCPT_T := 27;
   --  RM 11.7(10): lower bounds of ranges do not match
   MATCH_UBOUND_XCPT        : constant XCPT_T := 28;
   --  RM 11.7(10): upper bounds of ranges do not match
   NULL_DEREF_XCPT          : constant XCPT_T := 29;
   --  RM 4.1(10): prefix has the access value NULL
   SELECT_ELSE_XCPT         : constant XCPT_T := 30;
   --  RM 9.7.1(11): all select alternatives are closed
   NO_RETURN_XCPT           : constant XCPT_T := 31;
   --  RM 6.5(2): function terminated without a RETURN statement
   AGG_OTHER_XCPT           : constant XCPT_T := 32;
   --  RM 4.3(6): aggregate has too few components
   PROPAGATION_XCPT         : constant XCPT_T := 33;
   --  RM 11.4.1(9): exception raised again for propagation
   RAISE_XCPT               : constant XCPT_T := 34;
   --  RM 11.3(3): RAISE statement raised named exception
   RERAISE_XCPT             : constant XCPT_T := 35;
   --  RM 11.3(3): RAISE statement reraised exception
   REAL_ACCURACY_XCPT       : constant XCPT_T := 36;
   --  RM 11.1(6): universal real cannot be accurately represented
   SUBP_ELAB_XCPT           : constant XCPT_T := 37;
   --  RM 3.9(5): subprogram called before body has been
   --  elaborated
```

```
ACCEPT_XCPT             : constant XCPT_T := 38;
--  RM 11.5(a): exception raised again for propagation
INDEX_RANGE_LBOUND_XCPT : constant XCPT_T := 39;
--  RM 3.6.1(4): lower bound less than lower bound of index
--  subtype
INDEX_RANGE_UBOUND_XCPT : constant XCPT_T := 40;
--  RM 3.6.1(4): upper bound greater than upper bound of index
--  subtype
RANGE_LBOUND_XCPT       : constant XCPT_T := 41;
--  RM 3.5(4): lower bound of range less than lower bound of
--  subtype
RANGE_UBOUND_XCPT       : constant XCPT_T := 42;
--  RM 3.5(4): upper bound of range greater than upper bound of
--  subtype
ENUM_VALUE_XCPT         : constant XCPT_T := 43;
-- RM 3.5.5(13): no literal in the enumeration type matches
-- 'value parameter
ENUM_SYNTAX_XCPT        : constant XCPT_T := 44;
-- RM 3.5.5(13): 'value parameter does not have syntax of
-- enumeration literal
OUT_OF_MEMORY_XCPT      : constant XCPT_T := 45;
--  RM 11.(8): out of heap memory
OUT_OF_LHEAP_XCPT       : constant XCPT_T := 46;
--  RM 11.1(8): out of local heap memory
INT_SYNTAX_XCPT         : constant XCPT_T := 47;
--  RM 3.5.5(13), 2.4: 'value parameter does not have syntax of
--  integer
INT_BASE_XCPT           : constant XCPT_T := 48;
--  RM 3.5.5(13): base is out of range 2..16 ('value parameter)
NEG_INT_EXPONENT_XCPT   : constant XCPT_T := 49;
--  RM 3.5.5(13): 'value parameter has negative exponent
INT_TOO_BIG_XCPT        : constant XCPT_T := 50;
--  RM 3.5.5(13): result of 'value conversion is out of range
INT_DIGIT_BASE_XCPT     : constant XCPT_T := 51;
--  RM 3.5.5(13), 2.4: digit in 'value parameter is too large
--  for base
INT_MAGNITUDE_XCPT      : constant XCPT_T := 52;
--  RM 3.5.5(13): 'value parameter is too large to convert to
--  integer
POS_VAL_XCPT            : constant XCPT_T := 53;
--  RM 3.5.5(7): 'val parameter is out of range
CALL_COMPLETE_XCPT      : constant XCPT_T := 54;
--  RM 9.5(16): called entry of task which has completed
--  execution
CALL_ABNORMAL_XCPT      : constant XCPT_T := 55;
--  RM 9.10(7): called entry of abnormal task
PROPOGATED_XCPT         : constant XCPT_T := 56;
--  RM 11.5(4): propogated exception; original raised in called
--  task
ABORTED_XCPT            : constant XCPT_T := 57;
--  RM 9.10(7): called task has been aborted
COMPLETED_XCPT          : constant XCPT_T := 58;
```

```
   --  RM 9.5(16): called task has completed
   ALL_CLOSED_XCPT          : constant XCPT_T := 59;
   --  RM 9.7.1(11): all select alternatives are closed
   NOT_VALUE_XCPT           : constant XCPT_T := 60;
   --  RM 3.5.4(10): value cannot be represented
   TERMINATED_XCPT          : constant XCPT_T := 61;
   --  RM 9.3(4) and RM 9.5(16): called task has been terminated
   CONVERT_XCPT             : constant XCPT_T := 62;
   --  RM 3.5.4(10): conversion cannot deliver a correct result
   FLT_DIV_BY_0_XCPT        : constant XCPT_T := 63;
   --  Floating point divide by zero
   FLT_UNDERFLOW_XCPT       : constant XCPT_T := 64;
   --  Floating point underflow
   FLT_OVERFLOW_XCPT        : constant XCPT_T := 65;
   --  Floating point overflow
   FLT_SUBRANGE_XCPT        : constant XCPT_T := 66;
   --  Floating point, subscript out of range
   DEALLOCATED_POOL_XCPT    : constant XCPT_T := 67;
   --  Objects in user memory pool cannot be individually freed
   BAD_CLAIM_XCPT           : constant XCPT_T := 68;
   --  Memory requested for claim was not available
   NOEXTEND_POOL_XCPT       : constant XCPT_T := 69;
   --  Out of memory in non-extendible user pool
   INDEX_XCPT               : constant XCPT_T := 70;
   --  RM 4.1.1(4): index value outside range of index subtype
   VALUE_XCPT               : constant XCPT_T := 71;
   --  RM 3.5: value outside range of subtype
   STACK_LIMIT_XCPT         : constant XCPT_T := 72;
   --  RM 11.1(8) subprogram call stack overflow

   --
   -- Predefined ADA exception strings
   --
   CONSTRAINT_ERROR_IMAGE   : STRING(1 .. 1);
      pragma INTERFACE_OBJECT(CONSTRAINT_ERROR_IMAGE,
      "_A_constraint_error.EXCEPTION");
   NUMERIC_ERROR_IMAGE      : STRING(1 .. 1);
      pragma INTERFACE_OBJECT(NUMERIC_ERROR_IMAGE,
"_A_numeric_error.EXCEPTION");
   PROGRAM_ERROR_IMAGE      : STRING(1 .. 1);
      pragma
INTERFACE_OBJECT(PROGRAM_ERROR_IMAGE,"_A_program_error.EXCEPTION");
   STORAGE_ERROR_IMAGE      : STRING(1 .. 1);
      pragma
INTERFACE_OBJECT(STORAGE_ERROR_IMAGE,"_A_storage_error.EXCEPTION");
   TASKING_ERROR_IMAGE      : STRING(1 .. 1);
       pragma
INTERFACE_OBJECT(TASKING_ERROR_IMAGE,"_A_tasking_error.EXCEPTION");

   RTS_MSG                  : XCPT_T;
       pragma INTERFACE_OBJECT(RTS_MSG, "_ _RTS_MSG");
```

```
   -- Raise routine called by the Ada run-time routines.
   -- Msg is an index into a table of diagnostics that the
   -- debugger has.
   procedure RTS_RAISE(IMAGE : in SYSTEM.ADDRESS; MSG   : in XCPT_T);
      pragma INTERFACE(UNCHECKED, RTS_RAISE, "_ _RTS_RAISE");

end IFACE_RAISE;
```

18.26.2 Debugger Exception Output

The debugger uses the value in _ _RTS_MSG to obtain and elucidate the reason for the common Ada exceptions. The comment following the XCPT_T constant above gives the text of the debugger's reason. For example, a program might get a STACK_LIMIT violation and thus execute a call to:

```
rts_raise(constraint_error, index_xcpt);
```

If this exception is not handled in the program, the following error message is displayed when running the program with the debugger.

```
MAIN PROGRAM ABANDONED -- EXCEPTION "constraint_error" RAISED
 (program is now stopped on line where exception was raised)
 stopped 5 instructions after "/vc/sfz/t/ada.tests/storage.a":6 in
storage'2
   6          a(i) := 'a';
      --------^
=> RM 4.1.1(4): index value outside range of index subtype
```

CHAPTER 19

VADS COMMAND REFERENCE

summary

VADS includes the components listed below. The reference manual pages that follow are arranged alphabetically.

ada	invoke the Ada compiler
a.cleanlib	reinitialize library directory
a.das	disassemble object files
a.db	debug Ada source code programs
a.du	summarize disk usage for Ada libraries
a.error	analyze and disperse error messages
a.help	interactive help utility for VADS
a.info	list or change VADS library directives
a.ld	build an executable program from previously compiled units
a.list	produce program listing
a.ls	list compiled units
a.make	recompile source files in dependency order
a.mklib	create an VADS library directory
a.path	report or change VADS library search list
a.pr	format source code
a.report	report VADS deficiencies
a.rm	remove an Ada source file or unit from a library
a.rmlib	remove a compilation Ada library
a.tags	create a source file cross reference of units
a.vadsrc	display available VADS versions and create a default library configuration file
a.which	find a compiled unit

ada — Ada compiler

Syntax

```
ada [options] [ada_source.a]... [linker_options] [object_file.o]...
```

Description

The command **ada** executes the Ada compiler and compiles the named Ada source file, ending with the **.a** suffix. The file must reside in a VADS library directory. The **ada.lib** file in this directory is modified after each Ada unit is compiled.

The object for each compiled Ada unit is left in a file with the same name as that of the source with **01**, **02**, etc. substituted for **.a**. The **-o** option can be used to produce an executable with a name other than **a.out**, the default.

By default, **ada** produces only object and net files. If the **-M** option is used, the compiler automatically invokes **a.ld** and builds a complete program with the named library unit as the main program.

Non-Ada object files (**.o** files produced by a compiler for another language) may be given as arguments to **ada**. These files will be passed on to the linker and will be linked with the specified Ada object files.

Command line options may be specified in any order, but the order of compilation and the order of the files to be passed to the linker can be significant.

Several VADS compilers may be simultaneously available on a single system. Because the **ada** command in any *VADS_location*/**bin** on a system will execute the correct compiler components based upon visible library directives, the option **-sh** is provided to print the name of the components actually executed.

Program listings with a disassembly of machine code instructions are generated by **a.db** or **a.das**.

Options

-a *file_name* — (archive) treat *file_name* as an **ar** file. Since some archive files end with **.a**, **-a** is used to distinguish archive files from Ada source files

-d — (dependencies) analyze for dependencies only. Do not do semantic analysis or code generation. Update the library, marking any defined units as uncompiled. The **-d** option is used by **a.make** to establish dependencies among new files

-e

(error) process compilation error messages using **a.error** and send it to standard output. Only the source lines containing errors are listed. Only one **-e** or **-E** option should be used

-E
-E *file*
-E *directory*

(error output) without a file or directory argument, **ada** processes error messages using **a.error** and directs a brief output to standard output; the raw error messages are left in *ada_source*.**err**. If a file pathname is given, the raw error messages are placed in that file. If a directory argument is supplied, the raw error output is placed in *dir/source*.**err**. The file of raw error messages can be used as input to **a.error**. Only one **-e** or **-E** option should be used

-el

(error listing) intersperse error messages among source lines and direct to standard output

-El
-El *file*
-El *directory*

(error listing) same as the **-E** option, except that source listing with errors is produced

-ev

(error vi(1)) process syntax error messages using **a.error**, embed them in the source file, and call the environment editor ERROR_EDITOR. (If ERROR_EDITOR is defined, the environment variable ERROR_PATTERN should also be defined. ERROR_PATTERN is an editor search command that locates the first occurrence of '###' in the error file.) If no editor is specified, call **vi**(1)

-l*file_abbreviation*

(library) This is an option passed to the UNIX linker, **ld**(1) telling it to search the specified library file. (No space between the **-l** and the file abbreviation.)

For a description of the file abbreviations, **see also** Operating system documentation, **ld**(1)

-M *unit_name*

(main) produce an executable program using the named unit as the main program. The unit must be either a parameterless procedure or a parameterless function returning an integer. The executable program will be left in the file **a.out** unless overridden with the **-o** option

-M *ada_source.a*

(main) like **-M** *unit_name*, except that the unit name is assumed to be the root name of the **.a** file (for **foo.a** the unit is **foo**). Only one **.a** file may be preceded by **-M**

-o *executable_file*

(output) this option is to be used in conjunction with the **-M** option. *executable_file* is the name of the executable rather than the default **a.out**

-O[0-9]

(optimize) invoke the code optimizer (no space before the digit). An optional digit limits the number of passes by the optimizer; without the **-O** option, one pass is made; **-O0** prevents optimization; **-O** with no digit optimizes as far as possible

-R *VADS_library*

(recompile instantiation) force analysis of all generic instantiations, causing reinstantiation of any that are out of date

-S

(suppress) apply **pragma** SUPPRESS to the entire compilation for all suppressible checks

-sh

(show) display the name of the tool executable but do not execute it

-T

(timing) print timing information for the compilation

-v

(verbose) print compiler version number, date and time of compilation, name of file compiled, command input line, total compilation time, and error summary line

-w

(warnings) suppress warning diagnostics

See also *[VADS COMMAND REFERENCE]* **a.das**, page 19-6; **a.db**, page 19-8; **a.error**, page 19-10; **a.ld**, page 19-16; **a.mklib**, page 19-23; and Operating system documentation, **ld**(1).

Diagnostics

The diagnostics produced by the VADS compiler are intended to be self-explanatory. Most refer to the RM. Each RM reference includes a section number and optionally, a paragraph number enclosed in parentheses.

a.cleanlib — reinitialize library directory

Syntax

```
a.cleanlib [options] [VADS_library]
```

Description

a.cleanlib preserves all non-compilation information contained in **ada.lib**, including any additional libraries contained in the library search list and any other directives found in **ada.lib**.

The command will empty the files **ada.lib**, **gnrx.lib**, and **GVAS_table** of all separate compilation information and remove the contents of the directories **.imports**, **.nets**, **.lines**, and **.objects** from the named library or, if no library is specified, from the current library directory. All library directives in the **ada.lib** are preserved by **a.cleanlib**. It will also remove the file **name_lib** if present.

If **a.cleanlib** cannot find every library component, it will abort without removing any information unless the **-f** (force) option is given.

The **-F** option is provided to allow **a.cleanlib** to clean a library having a reserved name (i.e., **standard**, **verdixlib**, **publiclib**).

Options

-F	(force name) allow the cleaning of a VADS library having a reserved name
-f	(force) clean the VADS library structure even if components are missing or if lock files are found

Files

GVAS.lock, gnrx.lock	lock the library while reading or writing special library files
GVAS_table	address assignment file
gnrx.lib	generic instantiation reference file
ada.lib	library reference file
.lines	line number reference files directory
.imports	imported Ada units directory
.nets	Ada network control files directory
.objects	Ada object files directory

Diagnostics

An error is reported if any VADS component is missing, and no action is taken unless the **-f** option is used.

See also *[VADS COMMAND REFERENCE]*, **a.mklib**, page 19-23; **a.rmlib**, page 19-33.

a.das — disassemble object files

Syntax

```
a.das [options][path/]unit_name
a.das -n object_file
```

Description

a.das is an object module disassembler that interleaves Ada source lines and assembler instructions. Unlike disassembling from within **a.db**, it needs no target hardware to operate. **a.das** can disassemble any Ada unit. Without the **-s** or **-b** option, it disassembles the given unit body.

a.das requires only the Ada *unit_name* when run from within the VADS library containing the unit. **a.das** can be run on a unit in a non-local VADS library by prefixing the *unit_name* with the path of the VADS library containing it.

a.das can also be used to disassemble any object file produced by the compiler using the **-n** option and naming the file directly.

The data section (if present) may also be listed in various forms with the **-d** option.

VADS_location/**bin/a.das** is a wrapper program that executes the correct executable based upon directives visible in the **ada.lib** file. This permits multiple VADS compilers to exist on the same host. The **-sh** option prints the name of the actual executable file.

Options

-a (all) add hexadecimal display of instruction bytes to disassembly listing

-b (body) disassemble the indicated subprogram body [default]

-d (data) print the data section (if present) as well. **-d** may be followed by qualifying letters indicating the output format:

B or **b** in bytes

W or **w** in words

L or **l** in longwords

x hexadecimal output

d decimal output

With no qualifying letters the **-d** option defaults to **-dWx**. No space is required between the **-d** and the qualifying letters.

-f *file* (format) alternative format for output

-n (no source) disassemble object files. This requires no VADS library files

-s (spec) disassemble the indicated subprogram specification

-sh (show) display the name of the tool executable but do not execute it

a.db — source level debugger

Syntax

```
a.db [options] [executable_file]
```

Description

a.db is a symbolic debugger for Ada programs and for C programs compiled with the **-go** option for those using 4.2BSD UNIX or the **-g** option on System V UNIX. Detailed descriptions of interactive **a.db** commands and runtimeconfiguration file options are provided in the *DEBUGGER REFERENCE*, which is also available on-line using **a.help** or the debugger's internal **help** command.

VADS_location/**bin/a.db** is a wrapper program that executes the correct executable based upon directives visible in the **ada.lib** file. This permits multiple VADS compilers to exist on the same host. The **-sh** option prints the name of the actual executable file.

Options

-i *file_name*	(input) read input from the specified file
-p *VADS_library*	(program) read program compilation information from the specified VADS library directory (rather than the current directory)
-sh	(show) display the name of the tool executable but do not execute it
-v	(visual) invoke the screen-mode debugger directly

Files

GVAS.lock, gnrx.lock	lock the library while reading or writing special library files
GVAS_table	address assignment file
gnrx.lib	generic instantiation reference file
ada.lib	library reference file
.lines	line number reference files directory
.imports	imported Ada units directory
.nets	Ada network control files directory
.objects	Ada object files directory

See also *DEBUGGER REFERENCE, summary*, page 20-1 for a list of all debugger commands.

a.du — summarize library disk usage

Syntax

```
a.du [options] [VADS_library]
```

Description

a.du lists size in bytes for all compiler-generated files in the specified VADS library. If no library is specified, the current directory is assumed.

VADS_location/**bin/a.du** is a wrapper program that executes the correct executable based upon directives visible in the **ada.lib** file. This permits multiple VADS compilers to exist on the same host. The **-sh** option prints the name of the actual executable file.

Options

-a (address) sort the output by the GVAS address of each unit

-e (erroneous) include information for units with damaged or out-of-date net files

-f (file) sort output by the name of the file containing the unit [default]

-g (GVAS) provide the base address of each unit in the GVAS

-i (imports) include information for imported units

-sh (show) display the name of the tool executable but do not execute it

Files

GVAS_table	address assignment file
.imports	imported Ada units directory
.lines	line number reference files directory
.nets	Ada DIANA net files directory
.objects	Ada object files directory

a.error — analyze and disperse error messages

Syntax

```
a.error [options] [error_file]
```

Description

a.error is generally called from the **ada** command, but it can also be used separately. **a.error** analyzes and optionally disperses diagnostic error messages produced by the VADS compiler. It looks at the specified error file or the standard input, determines the source file and line number to which the error refers, determines whether the error is to be ignored or not, and outputs the associated source line followed by the error line(s).

a.error will also insert the error lines into the source file and invoke the **vi**(1) editor if the **-v** option is given. Error lines placed into files this way are of two types. The first gives the position of the error and the second identifies it. Multiple errors on a single line are referenced by sequential alphabetic characters.

```
     subtype T is range 1..1f;
-----------------^A                                    ###
---------------------------^B                         ###
--### A: syntax error: "identifier" inserted
--### B: lexical error: deleted
```

Because all error lines are flagged with ###, the **vi**(1) editor command **:g/###/d** can be used to delete them. However, any source lines containing ### will also be deleted; consequently, do not use ### in any source with which **a.error -v** may be used.

In the case of source files with multiple links, **a.error** creates a new copy of the file with only one link to it.

Options

-e *editor*	(editor) insert the error messages in the source file and invoke the specified editor
-f	(force) force a listing even if **pragma** LIST(OFF) is encountered
-l	(listing) produce a listing on the standard output
-N	(no) do not display line numbers
-s	(short) display only the error messages and the lines associated with them
-t*number*	(tabs) change tab default setting (8). (No space between **-t** and the following digit.)

-v	(vi) embed error messages in the source file and call the environment editor ERROR_EDITOR. (If ERROR_EDITOR is defined, the environment variable ERROR_PATTERN should also be defined. ERROR_PATTERN is an editor search command that locates the first occurrence of '###' in the error file.) If no editor is specified, call vi(1)
-W	(warnings) ignore warnings

Diagnostics

a.error produces diagnostics indicating 'no errors' if **-v** is used and no errors were detected, and 'no such file or directory' if invoked with an invalid file name.

See also *[VADS COMMAND REFERENCE]*, **ada**, page 19-2.

a.help — interactive help utility

Syntax

```
a.help [options] [subject]
```

Description

On-line help is available for each of the VADS utilities and for debugger commands and concepts. Without a specified subject, **a.help** provides information on use of the help utility and prompts for additional subject names. Use **q** to exit from **a.help**.

Without the **-p** option, **a.help** will use the paging program defined by the environment variable PAGER, requiring the full pathname including surrounding quotes for additional options. If PAGER is not defined, **more** is used.

Reference manual entries *for the compiler and tools only* are available on-line by using the **man** command if the local system administrator has elected to install them. A list of topics can then be obtained with

```
man ada
```

and typing

```
man VADS_command
```

will show the entry for a specific command.

VADS_location/**bin/a.help** is a wrapper program that executes the correct executable based upon directives visible in the **ada.lib** file. This permits multiple VADS compilers to exist on the same host. The **-sh** option prints the name of the actual executable file.

Options

-p *pager*	(pager) use *pager* as the paging program. The complete pathname must be given with surrounding quotes if additional options to the paging program are desired
-sh	(show) display the name of the tool executable but do not execute it

On-line Help from the Debugger

Access on-line help *for the debugger as well as the compiler and tools* during a debugging session by typing

```
help [subject]
```

or

```
:help [subject]
```

while in screen mode.

If the subject is omitted, a list of debugger commands is displayed. This overview can also be obtained by typing **intro** after a help prompt. Help with the **help** command can be obtained by typing **help** at a help prompt.

Files

VADS_location/**sup/help_files/***

a.info — list or change VADS library directives

Syntax

```
a.info [options]
```

Description

a.info is used to examine the INFO and LINK directives of the **ada.lib**. It can also be used to add or delete those directives from the **ada.lib** or to display or change the library search list.

All directives have the format

name : *type* : *value* :

where *name* is the name of the directive, *type* can be the word LINK or INFO, and *value* is usually a file name. (More information on directives can be found in discussions of **a.ld**.)

Without options, **a.info** displays all directives in the current library.

The **-i** option executes **a.info** in interactive mode. In this mode, all command line actions may be performed interactively. **a.info** prompts for and checks that the desired directive names and values are supported.

For a complete list of directive names, **see also** *IMPLEMENTATION REFERENCE, 18.19.3 Supported INFO and LINK Directive Names*, page 18-35.

For a discussion of WITH*n* directives used with the prelinker, **see also** *PROGRAM GENERATION TOOLS, 6.3* **a.ld**, page 6-1 and *[PAEDS COMMAND REFERENCE]*, **a.ld**, page 19-16.

Regular expressions, shown in the options below, are formed by following the operating system documentation.

See also *Operating system documentation*, **ed**(1).

Options

-a	(all) Display all directives in each library on the library search list
-F	(suffix) display LINK directives with the suffix (L) and INFO directives with the suffix (I)
-i	(interactive) operate in interactive mode
-l	(long) display all directives and their corresponding values

-p	(path) print the library search list
-s	(short) show just the INFO and LINK names
-v	(verbose) display maximum information
[+ -]info *name value*	add, delete an INFO directive
[+ -]link *name value*	add, delete a LINK directive
[+ -]link WITH*n value*	add, delete a LINK directive
+link WITH *value*	add a LINK directive having the next number: WITH*n*
[+ -]*number VADS_library*	add, delete *VADS_library* in the *number* position to the library search list
-*number*	removes VADS library in position *number* from the library search list
"*regular_expression***"**	print directives whose first *name* field matches *regular_expression*
-value "*regular_expression***"**	print directives whose last *value* field matches *regular_expression*

Files

VADS_location/**sup/LEGAL**.*target_name* — A list of legal directives for this implementation

a.ld — prelinker

Syntax

```
a.ld [options] unit_name [ld_options]
```

Description

a.ld collects the object files needed to make *unit_name* a main program and calls the UNIX linker **ld**(1) to link together all Ada and other language objects required to produce an executable image in **a.out**. *unit_name* is the main program and must name a non-generic subprogram. If *unit_name* is a function, it must return a value of the type STANDARD.INTEGER. This integer result will be passed back to the UNIX shell as the status code of the execution. The utility uses the net files produced by the Ada compiler to check dependency information. **a.ld** produces an exception mapping table and a unit elaboration table and passes this information to the linker.

a.ld reads instructions for generating executables from the **ada.lib** file in the VADS libraries on the search list. Besides information generated by the compiler, these directives also include WITH*n* directives that allow the automatic linking of object modules compiled from other languages or Ada object modules not named in context clauses in the Ada source. Any number of WITH*n* directives may be placed into a library, but they must be numbered contiguously beginning at WITH1. The directives are recorded in the library's **ada.lib** file and have the following form.

```
WITH1:LINK:object_file:
WITH2:LINK:archive_file:
```

WITH*n* directives may be placed in the local Ada libraries or in any VADS library on the search list.

A WITH*n* directive in a local VADS library or earlier on the library search list will hide the same numbered WITH*n* directive in a library later in the library search list.

Use the tool **a.info** to change or report library directives in the current library.

All arguments after *unit_name* are passed on to the linker. These may be options for it, archive libraries, library abbreviations, or object files.

VADS_location/**bin/a.ld** is a wrapper program that executes the correct executable based upon directives visible in the **ada.lib** file. This permits multiple VADS compilers to exist on the same host. The **-sh** option prints the name of the actual executable file.

Options

-E *unit_name*	(elaborate) elaborate *unit_name* as early in the elaboration order as possible
-F	(files) print a list of dependent files in order and suppress linking
-o *executable_file*	(output) use the specified file name as the name of the output rather than the default, **a.out**
-sh	(show) display the name of the tool executable but do not execute it
-U	(units) print a list of dependent units in order and suppress linking
-v	(verbose) print the linker command before executing it
-V	(verify) print the linker command but suppress execution

Files

VADS_location/**standard/***	startup and standard library routines
.nets	Ada DIANA net files directory
.objects/*	Ada object files
a.out	default output file

See also *Operating system documentation*, **ld**(1).

Diagnostics

Self-explanatory diagnostics are produced for missing files, etc. Occasional additional messages are produced by the linker.

a.list — produce program listing with line numbers

Syntax

```
a.list [-N] ada_source.a
```

Description

a.list provides a convenient way of producing a listing for programs containing no errors that closely resembles the output of **a.error**. The listing is written to the standard output and may be piped or redirected to a file.

Option

-N (no) suppress line numbers

See also *[VADS COMMAND REFERENCE]*, **a.error**, page 19–10; **a.pr**, page 19–26.

a.ls — list compiled programs

Syntax

```
a.ls [options] [unit_name] ... [-f ada_source.a ...]
```

Description

a.ls provides a list of the units compiled in the current VADS directory. Options are provided to give more or less extensive information, to change the format of the list, or to provide a list of compiled units occurring in specified source files. Additionally, *unit_name* can be specified as a regular expression to match groups of units. (If the regular expression contains any of the shell's meta characters, the expression must be quoted.)

With the -l or -v options, a.ls prints output in multiple columns. This can be overridden with the -1 (single) option.

The options -F, -l, and -v (in increasing order of listing detail) are mutually exclusive. If more than one of these three is given, the listing will be that with the most detail.

Options

-a (all) list all units visible in libraries on the library search list

-b (body) limit output to unit bodies

-f *filename*
(file) list only units found in *filename*

-F (suffix) list unit bodies with a trailing #

-l (long) list source file date, net file date, unit, and unit type

-s (specification) limit output to unit specifications

-v (verbose) list source file name, source file date, net file date, and unit

-1 (single) print output in a single column

See also *Operating system documentation for regular expressions in* **ed***(1)*.

a.make — recompile source files in dependency order

Syntax

```
a.make [options] [unit_name]... [ld_options] [-f ada_source.a ...]
a.make [options] [path/unit_name]... [ld_options] [-f ada_source.a ...]
```

Description

This utility determines which files must be recompiled in order to produce a current executable file with *unit_name* as the main unit. It also calls **a.ld** to create the appropriate executable, if and only if *unit_name* is a procedure or an integer function; otherwise, it just ensures that the named unit is up–to–date, recompiling any dependencies if necessary.

The utility uses DIANA net files to determine the correct order of compilation and elaboration.

Unless the **–f** option is used, **a.make** will have no knowledge of any source file (**foo.a**) until that file has been compiled in a way that changes the program library. This requires that **foo.a** be compiled 'by hand' at least once. Unless the **–U** or **–D** option is given, the file must compile successfully or else the program library will remain unchanged. In any case, syntax errors must be corrected before the file will be 'seen' by **a.make.**

VADS_location/**bin/a.make** is a wrapper program that executes the correct executable based upon directives visible in the **ada.lib** file. This permits multiple VADS compilers to exist on the same host. The **–sh** option prints the name of the actual executable file.

Supplied names and unknown options are passed to **a.ld.**

Options

–A *VADS_library* [**–A** *VADS_library*] ...
(add) bring the listed libraries up to date if necessary

–All (all) bring all libraries on the library search path up to date

–C "*compiler*" (compiler) use the string *compiler* in recompiling the required units. This option is normally used to provide specific options to the compiler. For example, to call the compiler with the optimizing option and invoke **vi**(1) on compilation errors, use a command of the following type

```
a.make -C "ada -ev" [other commands]
```

-D	(dependencies) list the file-to-file dependencies
-f *ada_source.a ...*	(files) treat remaining non-option arguments as file names in the current VADS library to consider for compilation. All units in these files will be brought up to date; **-f** may be used with one of the other options to print actions or dependencies without executing them, but must be the last option given
-F *file_name*	(named file) consider for compilation the source files listed in *file_name*. This option is similar to **-f** option except that listing the source files in *file_name* allows a list that might otherwise exceed the UNIX command-line limit. In *file_name*, multiple file names may be listed on a single line with any number of blanks or tabs separating them, however, only the first 2048 characters of each line will be read.
-I *ada_source.a*	(if) list actions that would be taken if *ada_source.a* were changed
-i	(ignore errors) do not suppress compilation if file is dependent upon another file that did not successfully compile
-L "*linker*"	(linker) use the string *linker* in linking the required units. This option can be used to provide unusual options to **a.ld** when using **a.make**
-O[0-9]	(optimize) invoke the code optimizer (no space before the digit). An optional digit limits the number of optimization passes; without the **-O** option, one pass is made; **-O0** prevents optimization; **-O** with no digit optimizes as far as possible
-S	(suppress) apply **pragma** SUPPRESS to the entire compilation
-sh	(show) display the name of the tool executable but do not execute it
-U	(units) list the list of dependent units in order, but do not link
-v	(verbose) list the recompilation commands as they are executed
-V	(verify) list the recompilation commands that would be executed, but do not execute them

Files

GVAS.lock, gnrx.lock	lock the library while reading or writing special library files
GVAS_table	address assignment file
gnrx.lib	generic instantiation reference file
ada.lib	library reference file
.lines	line number reference files directory
.imports	imported Ada units directory
.nets	Ada network control files directory
.objects	Ada object files directory

a.mklib — make library directory

Syntax

```
a.mklib [-F -f -v  -i] [new_VADS_library] [parent_VADS_library]]
```

Description

a.mklib creates and initializes a new VADS library directory, creating three files (**GVAS_table, ada.lib,** and **gnrx.lib**) and four directories (**.lines, .imports, .nets,** and **.objects**). It constructs library pointers in **ada.lib** to all libraries available from the parent library and to the parent library itself. As a result, Ada units in the new library can reference all Ada units defined by the parent library and all units that were accessible from the parent library.

NOTE: Verdix recommends invoking **a.mklib** with the **-i** (interactive) option.

If *parent_VADS_library* is unspecified, the default libraries are **verdixlib** and **standard.**

The tool **a.vadsrc** may also be used to create a local configuration file called **.vadsrc** either in the current directory or in the user's $HOME directory so that future libraries created in a directory below the current or $HOME directory will reference a particular VADS version.

If *new_VADS_library* is unspecified, the current working directory is initialized.

The **-f** option will force initialization of the VADS library structure, overwriting any existing components and deleting any existing lock files.

Without the **-F** option, **a.mklib** cannot create libraries named **standard, verdixlib,** or **publiclib.**

A list of available targets can be obtained with the **-i** option or with the tool **a.vadsrc.**

Options

-f (force) create VADS library structure even if some components are already present

-F (force name) allow creation of VADS library with a restricted name

-i (interactive) display all versions of VADS installed on the system and prompt for selection of VADS version unless modified with the -t option. This is the recommended way to use **a.mklib**

-v (verbose) display the library search list and target directives

Example

If the user is positioned at the directory **/usr2/babbage/code** and the VADS library **olddir** exists below it in the UNIX hierarchy, the command

```
a.mklib newdir olddir
```

creates the library directory **/usr2/babbage/code/newdir** and provides access to the Ada compilation units previously compiled in the **olddir** library directory. Any units available to **olddir** from other libraries are now available from **newdir** as well.

Files

.vadsrc	local default configuration file
/etc/VADS	VADS version reference file

Diagnostics

An error is reported and no action is taken (without the **-f** option) if *new_VADS_library* contains any VADS components or lock files or if the name specified exists but is not a directory.

See also *[VADS COMMAND REFERENCE]*, **a.cleanlib**. page 19-5; **a.rmlib**, page 19-33; and **a.vadsrc**, page 19-36.

a.path — report or change VADS library search list

Syntax

```
a.path [options] [VADS_library1 [VADS_library2]]
```

Description

A list of libraries to be searched during compilation is maintained in the current VADS library directory in the file **ada.lib**. **a.path** changes or reports the list of library names contained there. During compilation, any program units not found in the current library will be searched for in the VADS libraries listed on the search list. If the unit is not found in the first VADS library, it is searched for in the second, and so on in listed order. When **a.path** is used with no options, it reports the contents of the current library search list, one library to a line.

Removing a library name from the library search list does not remove compilation information from the referenced libraries.

Maximum length of the library search list is 2048 characters.

Options

-a *VADS_library1* [*VADS_library2*]
: (append) append *VADS_library1* after *VADS_library2*. With a single argument, append *VADS_library1* to the end of the library search list

-i *VADS_library1* [*VADS_library2*]
: (insert) insert *VADS_library1* before *VADS_library2*. With a single argument, insert *VADS_library1* at the beginning of the list

-r *VADS_library1*
: (remove) remove *VADS_library1* from the library search list

-v
: (verbose) display path as it is changed

-t
: (target) display library search list and target information

-x *VADS_library1*
: (except) remove all except *VADS_library1* from the list

a.pr — format source code

Syntax

```
a.pr [options] [ada_source.a]
```

Description

a.pr reformats Ada source code according to options specified in a runtime configuration file with the name **.prrc**. This allows users to tailor **a.pr** for individual Ada coding standards. The configuration file may be located either in the user's current working directory or the home directory.

Additionally, options may be specified on the command line that override those in the configuration file. The options are listed below.

Invoked without a file name, **a.pr** reads its input from standard input.

Error and warning messages are written to standard error.

.prrc Configuration File Options

(default shown in brackets)

align_cmts *where*	align comments to the right of the longest line (**line**) or the longest line containing a comment (**comment**) [**comment**]
chars *number*	specify maximum number of characters of code per line including comment and indentation; any line extending over this limit will be continued on the next line; valid range is from 20 to 500 [132]
comment *case*	print all comments in the specified case: **upper**, **lower**, **same** [**same**]
ident *case*	print all identifiers in the specified case: **upper**, **lower**, **same** [**upper**]
indent *number*	specify amount of indentation between levels; valid range is from 1 to 8 [8]
lines *number*	specify maximum number of lines allowed on a page; valid range is from 1 to 1000 [55]
margin *number*	specify starting margin for top–most level; valid range is from 0 to 15 [0]
no_page	paginate only when **pragma** PAGE is encountered [**page**]

no_warning	suppress warning messages regarding line length greater than desired [provide warnings]
page *number*	set page size; perform pagination with blank lines; valid range is from 1 for 1000 [paginate using form feeds]
page_lu	start each library unit (indicated by a **with** clause) on a new page [do not start on new page]
record *where*	print **record** on either the same line (**same**) or on the next one (**next**) [**same**]
reserved *case*	print all reserved words in the specified case: **upper**, **lower**, **same** [**lower**]
tabs *number*	print tabs for indentation whenever the number of spaces needed for indentation is greater than or equal to the specified number; valid range is from 0 to 8; if **tabs 0** is specified, indentation will be performed with blanks [8]

a.pr Command Line Options
(Default shown in brackets)

-ac	(align comment) align comments to the right of the longest line that contains a comment [default]
-al	(align line) align comments to the right of the longest line, regardless of whether it contains a comment [**-ac**]
-c *number*	(characters) specify maximum number of characters of source code allowed on a line. Valid range is from 20 to 500 [132]
-cl	(comments lower) print comments in lower case [**-cs**]
-cs	(comments same) print comments as in source code [default]
-cu	(comments upper) print comments in upper case [**-cs**]
-i *number*	(indent) specify indentation between levels. Valid range is from 1 to 8 [8]
-il	(identifiers lower) print identifiers in lower case [**-iu**]
-is	(identifiers same) print identifiers as in source code [**-iu**]
-iu	(identifiers upper) print identifiers in upper case [default]

-l *number*	(lines) specify maximum number of lines allowed on a page. Valid range is from 1 to 1000 [55]
-m *number*	(margin) specify starting margin for top-most level. Valid range is from 0 to 15 [0]
-nl	(no page library unit) do not start a new page for each library unit [default]
-np	(no pagination) specify no pagination. Pagination will occur only when **pragma page** is encountered [**-pg**]
-nw	(no warnings) suppress warning messages regarding line length [**-w**]
-p *number*	(page) specify page size. Valid range is from 1 to 1000 [**-pg**]
-pg	(pagination) paginate using form feeds [default]
-pl	(page library) start a new page whenever a library unit is encountered [**-nl**]
-rl	(reserved lower) print reserved words in lower case [default]
-RN	(record next) print **record** on the line following **type** or **for** [**-RS**]
-rs	(reserved same) print reserved words as in source code [**-rl**]
-RS	(record same) print **record** on the same line as **type** or **for** [default]
-ru	(reserved upper) print reserved words in upper case [**-rl**]
-t *number*	(tabs) specify tabs for indentation whenever the number of spaces needed is greater than or equal to the specified number. If **-t 0** is specified, indentation will be performed with spaces. Valid range is from 0 to 8 [8]
-w	(warning) provide warning messages regarding line lengths greater than desired [default]

a.report — report deficiency or suggestion

Syntax

```
a.report
```

Description

VADS provides **a.report** for customers contracting with Verdix for Level I or II product support to submit Customer Reports (CR) to Verdix's Ada Product Support Office (APSO). Each CR received by APSO is entered into a database for automated tracking and processing.

A unique Customer Report Number (CRN) is assigned by APSO and an acknowledgment is sent to the originator. This acknowledgment references the customer's local report number, reproduces some of the report text, and gives the CRN and the date the report was received by Verdix. Any subsequent reference to the CR should use the assigned CRN.

a.report automatically captures the date, time, site name, user ID, and VADS configuration information. (See the sample report at the end of this entry.) It then prompts for the name of source files illustrating the problem to be appended to the report and invokes vi(1) or the user's default editor to allow editing of general text that describes the problem. The formatting lines supplied by **a.report** separate parts of the text for APSO's automated processing. Report generation can be aborted at any point using the <INTR> key (usually mapped to <CONTROL>c).

A warning is displayed if the report length is greater than that acceptable by most network mailers. If the warning appears, the user should be certain that there are no intervening systems between Verdix and the report site. If there are, the report should be saved in a file and transmitted to APSO via magnetic tape.

When a report has been completed, the entire file is electronically mailed to persons determined by the system administrator when VADS was installed. This list usually includes the site Ada administrator. The user can add additional mail destinations and/or save a copy of the report by supplying a file name when prompted.

APSO then analyzes the CR to determine if a previously unreported deficiency has been found. If so, a Deficiency Report (DR) is generated by APSO and a unique Deficiency Report Number is assigned for tracking purposes. At this point the Customer Report is closed and the customer is notified of the resolution either directly (Level II only) or through the VADS Bulletin Board System (BBS) described below.

Information about all DRs currently existing against VADS is maintained on the BBS, which is available to registered VADS users. To register, a user logs onto the BBS at (703) 378-8504 and responds to a questionnaire. After information verification, the caller will be granted BBS access commensurate with product support level.

Customer Reports can be submitted to APSO by electronic mail, magnetic tape, or in hard-copy (if absolutely necessary). APSO prefers electronic mail and can usually process these reports more quickly and accurately.

APSO can be reached at the following address.

Ada Product Support Office
Verdix Corporation
Sullyfield Business Park
14130-A Sullyfield Circle
Chantilly, Virginia 22021
(703) 378-7600
INTERNET: apso@vrdxhq.uu.net or vrdxhq!apso@uunet.uu.net
USENET: {uunet}vrdxhq!apso

A **sample customer report** generated by **a.report** is shown on the next page.

```
VADS report number 4
============================================================
Host: MELKOR      User: maf
VADS: Version 5.5 - SUN-3 UNIX
Date/time: Tue Sep 22 16:04:46 PDT 1986 (c)
Hostid: 0      CRN reference: new
Title: internal error for nonstatic slice preceeding another slice
============================================================
Please enter problem description below:
When a nonstatic slice is specified, followed by a range with a
non-static
component, an internal assertion error occurs.  The error listing
follows:

********************  /vc/maf/stanford/test/slice.a  ********************

  17:  S := S(Nonstatic)(1..identity(10));  --  causes internal error
                                            --  w/ 5.4
A --------^
A:internal: assertion error at file static_type.c, line 147
============================================================
--==========++++++++++########## file slice.a
package slice_pkg is
  -- function to "hide" an integer so a non-static integer
  -- value is created
  function identity( i : integer ) return integer ;
end ;
package body slice_pkg is
  function identity( i : integer ) return integer is
  begin
    return i ;
  end ;
end ;


with slice_pkg ; use slice_pkg ;
procedure Slice is
  subtype Nonstatic is POSITIVE range 1 .. identity (10);
  S : STRING (1 .. identity (10)) := (others => 'a');
begin
  S := S(Nonstatic)(1..identity(10));   -- causes internal error w/ 5.4
  S := S(Nonstatic)(S'RANGE);           -- causes internal error w/ 5.4
end ;
--==========++++++++++########## end
============================================================
```

a.rm – remove an Ada source file or unit from a library

Syntax

```
a.rm [options] unit_name
a.rm [options] ada_source.a
```

Description

The **a.rm** command is executed while positioned in a VADS directory. It removes all information associated with the named unit(s) or file(s). When *unit_name* is specified, the corresponding files in **.nets**, **.objects**, and **.lines** are removed and the **ada.lib** entries are deleted.

When a file name *ada_source.a* is given, all net, object, and line number files are removed for each unit defined in the file, and the appropriate entries are deleted from **ada.lib**. A name ending in **.a** is taken to be an Ada source file name unless the **-u** option is given.

Unit names with dotted notation such as *aaa.bbb* or *aaa.bbb.ccc* are taken to be the names of Ada subunits.

Options

-b (body) delete the bodies of the specified units or files

-f (file) remove the Ada source file in addition to the compiler-generated files whenever all units in a file are deleted

-i (interactive) prompt for confirmation before deleting information for any units

-s (specification) remove the compilation information for the specifications of the specified units

-u (unit) force the next name to be treated as a unit even though it ends in **.a**

-v (verbose) list the units as they are removed

-V (verify) list the units that would be removed, but do not remove them

a.rmlib — remove compilation library

Syntax

```
a.rmlib [-f -F] [VADS_library]
```

Description

a.rmlib removes all VADS library components from *VADS_library* or from the current library if no argument is given. It removes three files (**GVAS_table, ada.lib, and gnrx.lib**), four directories (**.lines, .imports, .nets,** and **.objects**), and lock files, if present and the **-f** option is used. The directory itself, all Ada source files, and other files and directories are left untouched.

If *VADS_library* is unspecified, the current VADS library is used.

If **a.rmlib** cannot find every library component or lock files exist, it will abort without removing any files unless the **-f** (force) option is given.

Without the **-F** option **a.rmlib** cannot operate in a library bearing the name **standard, verdixlib** or **publiclib.**

The directory name for the removed library is left in dependent library paths. This blocks compilation in any dependent libraries until **a.path** is used to remove the path entry that specifies this directory. Compilation could also proceed if a VADS library is re-created in the named directory from which the library information was removed.

Options

-f (force) clean VADS library structure even if some components are missing or lock files exist

-F (force name) allow the cleaning of the VADS library structure of a library having a restricted name

Diagnostics

An error is reported and no action is taken (without the **-f** option) if *VADS_library* contains an incomplete set of components or a lock file.

An error message will be issued if any files or directories are not accessible for deletion.

See also *[VADS COMMAND REFERENCE]*, **a.mklib**, page 19-23; **a.cleanlib**, page 19-5.

a.tags — create a tags file

Syntax

```
a.tags [options] ada_source.a ...
```

Description

a.tags makes a **tags** file from the specified Ada source(s). The operation is similar to the UNIX **ctags**(1) command with modifications for Ada-specific features.

Each line of the **tags** file lists the object name, the file in which it is defined, and search patterns for locating each object's definition. UNIX editors such as **vi**(1) or **ex**(1) can use the **tags** file to locate units and, if the **-t** option is used, to locate types as well. Create the **tags** file with the command:

```
a.tags *.a
```

For example, to edit unit END_PROG without specifying the file that contains it, type the following command.

```
vi -t END_PROG
```

Ada allows unit name overloading, and **a.tags** requires special conventions to access different units having the same name. Ada **specifications** are named by prefacing the Ada simple name with **s#**. **Bodies** are named with the unmodified Ada name. Stubs for **separates** are named by prefacing the Ada simple name with **stub#**.

Nested packages, subprograms, types, generics, and task definitions are always listed both with their full Ada expanded name and with any tag prefaces added to the simple name. Simple names for nested units are listed only if the simple name is unique across all other tags. Thus the user may use the simple name if it is unique and may always use the full name.

Fully qualified overloaded names within a file are not differentiated. However, the tag identifies the correct file, and repeated application of the search pattern will find the desired subprogram. The search pattern is generalized to match all versions of the overloaded subprogram; this generalization may cause the pattern to recognize things other than the desired unit. Identical fully qualified names across files are not handled.

The **-x** and **-v** options provide listings on the standard output; all other options refer to the file **tags** generated for use by **ex**(1) or **vi**(1).

Options

-a (append) append to the **tags** file

-B (backward) record backward searching patterns (?)

-F (forward) record forward searching patterns (/). Default

-t (types) create tags for types also

-v (vgrind) generate an index with line numbers for **vgrind**(1) on the standard output

-w (warnings) suppress warning messages

-x (cross) generate an indexed list of all tags on the standard output

See also *Operating system documentation*, **ctags**(1).

Bugs

When using **ex**(1) or **vi**(1) with the **-t** option, the command line must contain the desired unit or type in the same case (upper or lower) as its occurrence in the source file.

a.vadsrc — display available VADS versions and create a default library configuration file

Syntax

```
a.vadsrc [-i] [-v]
```

Description

When multiple VADS targets or versions are present on the same system, **a.vadsrc** is useful to control the default version or target processor for which libraries are created.

With no option, **a.vadsrc** simply reports the installed VADS versions.

If the –i (interactive) option is used, the tool prompts for selection of a VADS version and creates a **.vadsrc** file in the current directory.

Option

–i (interactive) show all versions of VADS installed on the system and prompt for a selection

–v (verbose) print the contents of any **.vadsrc** files in the current directory or in the user's **$HOME** directory.

Files

/etc/VADS VADS version reference file

See also *[VADS COMMAND REFERENCE]*, **a.mklib**, page 19–23.

a.which — determine which project library contains a unit

Syntax

```
a.which [options] [unit_name]
a.which [options] [path/unit_name]
```

Description

a.which lists the name of the source file that defines the version of *unit_name* visible in the current VADS library. The program library search sequence may also be printed. The **-b** (body) option lists the source file location of the unit body. Without this option, the unit's specification is located.

Options

-b (body) give the location of the body

-v (verbose) give the library search list

Bugs

An option is needed so that hidden units can be printed as well to allow programmers to identify unit naming conflicts.

CHAPTER 20

DEBUGGER REFERENCE

summary

Reference entries for each debugger command and concept listed below are arranged in alphabetical order.

a	step one source line over calls
a.db	invoke the debugger from the shell
address	address memory directly
ai	step one machine instruction over calls
assignment (:=)	assign a value to a variable, register or memory location(s)
b	set a breakpoint at a line or beginning of a subprogram
bd	set a breakpoint after the end of the current subprogram
bi	set a breakpoint at a machine instruction
breakpoints	control a program's execution
bx	set a breakpoint when an Ada exception occurs
cb	move to the bottom frame of the call stack
cd	move down on the call stack
command syntax	express debugger commands
cs	display the call stack
ct	move to the top frame of the call stack
cu	move up on the call stack
current frame	current position on the call stack
current position	current position in a source file
d	delete breakpoints
DB_IFACE (cross debugger interface)	interface between cross debugger and target
display memory (@)	display raw memory
e	move to a new source file
edit	invoke the editor on a subprogram or file
examine	display variables, files, debugger parameters, etc.
exit	terminate the debugger session
expressions	arithmetic expressions in the debugger

files	specify files to be used by the debugger
g	continue executing the program
gw	continue executing the program until a variable changes
gx	continue execution and pass the signal to the program
help	print help text
home position	execution point in the current frame
invocation	see **a.db**
l	display part of a source program
lb	list all currently set breakpoints
li	list disassembled instructions
line numbers	move to a specified line
load	download the memory image to the target processor
lt	list all active tasks
overloading	disambiguate overloaded names
p	display the value of a variable or expression
pass	pass a command to the target controlling agent
quit	terminate the debugger session
r	run the program
read	read debugger commands from a file
reg	list the current machine register contents
<RETURN>	re-execute the most recently repeatable debugger command
s	step one source line, into subprograms
sh	spawn a UNIX shell
screen mode	screen-oriented debugger interface
search (? /)	search forward/backward in the current file for a pattern
select	choose a new task to be the current task
set	set debugger parameters
si	single step one machine instruction into program
stop	stop the debugger or program
sync	set/reset the debugger's knowledge of the target
task	display the address of the current task
terminal control	catching program input/output
vi	switch the debugger into screen mode
visibility rules	determine which identifiers are visible at a breakpoint
w	list a group of source lines surrounding a line
wi	list machine instructions surrounding an instruction

a — (advance) step one source line over calls

Syntax

a	<RETURN> repeats

Description

a single steps to the next source line for which the compiler generated code, stepping *over* subprogram calls.

Other stepping commands are **ai**, **s**, **si**, and **gw**. The **s** command steps one source line *into* called subprograms.

If the program has not yet started or if it has terminated, **a** starts the program, stepping one source line. For Ada programs, this steps over all the library unit elaborations.

Note that currently the **a** command does not advance over entry calls, only procedure calls.

In Screen Mode

The **a** command is directly supported in screen mode: type **a** to single step one source line over subprogram calls. It is unnecessary to press <RETURN> in screen mode.

With **safe** set **on**, the screen mode command becomes **aa**.

See also *[DEBUGGER REFERENCE]*, **set**, page 20–70.

In instruction submode, **a** is interpreted as **ai**.

See also *[DEBUGGER REFERENCE]*, *screen mode*, page 20–63.

a.db — execute the debugger from the shell

Syntax

```
a.db [options] [executable_file]
```

Options

-i *file_name*
: (input) read command input from specified file. When the debugger commands in this file have all been read and executed, the debugger switches to reading from standard input (unless the file contains a **quit** or **exit** command). The commands can be any **set** commands or line-mode commands.

-p *VADS_library*
: (program) read program compilation information from the specified VADS library directory.

-sh
: (show) display the name of the debugger executable but do not execute it

-v
: (visual) invoke the screen-mode debugger.

< file_name
> file_name
>& file_name
: Normally, the debugger reads from the terminal. You may redirect any or all of standard input, standard output, and standard error to a file. The debugger may also be run in background by appending '&' to the invocation line. For example:

```
a.db my_prog < debug.in >& debug.out &
```

There are two restrictions:

1. You cannot use screen mode when debugging input is a file.
2. Your program cannot share the debugger's input file. Use **set input** to set the input file for your program. A sample debug.in file might be:

```
set input my_prog.in
r
quit
```

See also *[DEBUGGER REFERENCE], screen mode,* page 20-63.

Description

In addition to the invocation line, the user can supply parameters to **a.db** using a **.dbrc** file. During debugger initialization, **a.db** checks for **./.dbrc**. If that does not exist, it checks for **$HOME/.dbrc**. The **.dbrc** file may contain only **set** commands. These commands are executed before any other commands, including those in the input file specified on the command line.

If *executable_file* is not specified, **a.db** searches for **a.out**. If only a root file name is given (**foo**, as opposed to **/usr2/bohunk/foo**), **a.db** searches the directories on the $PATH environment (exported) variable for an executable file **foo** just as the shell does.

If a **.dbrc** file is present, the debugger will also search directories specified in a **set source** command for the first directory that contains an **ada.lib** file. That directory is used by the debugger to obtain the DIANA net files and the line number files produced by the compiler.

The **quit** command is used to leave the debugger and return to the shell.

The debugger establishes the debugging environment when it starts up. Certain key parameters are displayed on the screen to verify what and where it is debugging.

```
Debugging: /vc/sbq/tst3/ent
VADS_library: /vc/sbq/tst3
library search list:
       /vc/sbq/tst3
       /bc/vads/standard
       /bc/acvc/report
```

The first line shows the name of the program being debugged. The *VADS_library* is the name of the VADS library directory being used for this debugging session. The library search list is derived from the **ada.lib** file in the *VADS_library*. It shows the VADS library directories that are searched when the debugger looks for an Ada unit. The search list is printed out in the same order that the debugger searches it. In the above example, **/vc/sbq/tst3** is searched first.

For more information about VADS libraries, **see also** *LIBRARY TOOLS, 4.7.5* **a.mklib**, page 4-8; *VADS DEBUGGER, 8.3 INVOKING THE DEBUGGER,* page 8-2; and *8.3.2 Ada Library Directory:* **-p**, page 8-3.

address — address memory directly

Description

Some commands (**li, wi, bi**) can take an address as a parameter, which is expressed as a hexadecimal number (with a leading 0).

In addition, an address can be typed in using the format *@number*. Here, @ is treated like a unary operator that means 'use the following number as an address'. The number following the @ can be either decimal or hexadecimal.

To display memory at a specific address, **see also** *[DEBUGGER REFERENCE], display memory*, page 20-29.

ai — (advance instruction) step one machine instruction over calls

Syntax

```
ai                                          <RETURN> repeats
```

Description

ai single steps one machine instruction *over* call instructions.

Other stepping commands are **a**, **s**, **si**, and **gw**. The **si** command single steps one instruction *into* called subprograms.

If the program has not yet been executed or if it has terminated, **ai** starts the program, stepping one instruction. This relocates the current position from the main subprogram to the actual starting subprogram that precedes the user's program.

In Screen Mode

Precede this command with : and follow with <RETURN>.

:= — assign a value to a variable, register, or memory location(s)

Syntax

name := *expression*
address [, *number*] := *expression*

Description

This command is used to modify memory. *name* can be any Ada expression that identifies a scalar or a register name (preceded by a dollar sign).

Note: if *name* is a debugger keyword, it must be preceded by a backslash or it will result in a syntax error (e.g. b := 3 will result in a syntax error because *b* is a debugger keyword. Debugger keywords are listed on pages 20–1 and 20–2).

expression can be any scalar expression.

address can be either a hexadecimal number (with a leading 0) or of the form *@number* where *number* (of *@number*) is hexadecimal or decimal. *number* (of the above syntax example) indicates how many bytes are to be modified. If the value of *expression* is an integer, *number* must be 1, 2, 3, or 4. If the value of *expression* is a floating point number, *number* must be either 4 or 8. If absent, *number* defaults to 4.

On cross compilers that support floating point coprocessors, the coprocessor registers may be assigned values directly. The number of bits in the coprocessor registers differ between implementations. Values assigned in this way are approximations only and the conversion routines are slow at present, particularly for wide coprocessor registers.

For further discussion of expressions, **See also** Ada RM 4.1 and *[DEBUGGER REFERENCE], expressions,* page *20–36.*

After the memory is modified, a line of the form

address: *new/old*

is printed. *address* is where the value was written, *new* is the value that was written, *old* is the value that used to be there. For variable names, *address* is not printed.

CAUTION: No type checking is performed between the name and the expression.

In Screen Mode

Precede this command with : and follow with <RETURN>.

b — (break) set a breakpoint at a line or beginning of a subprogram

Syntax

```
b [line|subprogram] [of task] [begin commands end]
b [line|subprogram] [of task] when expression
b [line|subprogram] [of task] if expression then commands
   [else commands] end [if]
b [line|subprogram] [of task] if expression else commands end [if]
```

Description

The basic form of the command to set a breakpoint is the first one shown above. This sets a breakpoint at a specified line in the current file or at the beginning of a specified subprogram. *line* is typically given as a decimal number; however, all the options specified in the **line number** section of this reference are supported. To set a breakpoint at line 537 of the current source file, use the following command.

```
b 537
```

subprogram is the name of an Ada subprogram, task body, or package. It is not necessary for the subprogram name to be directly visible. The name is first looked up as a normal Ada name. If it is not directly visible, all subprogram names in the program are searched. Expanded names as well as simple identifiers can also be used (starting with STANDARD if desired). To set a breakpoint at the subprogram SORT_STAMPS, the following command is used.

```
b SORT_STAMPS
```

See also *[DEBUGGER REFERENCE], visibility rules*, page 20–80 and for a discussion of expanded names, Ada RM 4.1.3(13).

If the *subprogram* name has multiple definitions of the same simple name within the context in which it is looked up, it is is overloaded. The debugger prints a diagnostic showing the alternatives. Retype the **b** command attaching a **'1, '2, ...** suffix (matching an alternative shown in the diagnostic) to the subprogram name to disambiguate it, or use an expanded name.

See also *[DEBUGGER REFERENCE], overloading*, page 20–53.

If *subprogram* or *line* is not specified, a breakpoint is set at the current position by using the line part of the current position three-part identifier (file, line number, and instruction address). In line mode, the current position is marked with <.

To set a breakpoint in another file or subprogram, use the **e** (enter) command to locate the correct source file.

See also *[DEBUGGER REFERENCE], current position*, page 20-25.

When the **of** *task* clause is used, the breakpoint is announced only for the specified task. *task* is a task number, obtained using the **lt** command. Currently the debugger does not verify that the task number is valid. **Be careful**, using an invalid task number can result in debugger errors.

Command Blocks and Conditional Breakpoints

A block of commands can be automatically executed when the breakpoint is reached by following the breakpoint with a command of the following format.

```
begin commands end
```

The breakpoint can be made conditional by appending the **when** *expression* clause. *expression* is checked for correctness at the time the breakpoint is set. This check takes place in the environment of the location of the breakpoint.

See also *[DEBUGGER REFERENCE], visibility rules*, page 20-80.

expression is evaluated each time the breakpoint is reached. If the *expression* is FALSE (0), the breakpoint is not announced and the program continues. If the *expression* is TRUE (non-zero), the breakpoint is announced.

Use the **if** statement to set a conditional breakpoint that also conditionally executes debugger commands when the breakpoint is reached. *expression* is evaluated each time the breakpoint is reached. If it is TRUE, the breakpoint is announced to the user and any commands in a **then** clause are executed. If the *expression* is FALSE, commands in an **else** clause are executed. The program then continues on its way (it is not breakpointed).

commands is a sequence of one or more debugger commands. These can be entered on the same line separated by semicolons, or each command can be entered on its own line as long as the first **begin**, **then**, or **else** is on the same line as the **b** keyword.

The second method is recommended. As each command of *commands* is entered on its own line, the debugger prompts with **??** for each new command until the sequence is terminated with **end** (or an **else** in the case of an **if...then...else**).

Each breakpoint set with a **b**, **bd**, **bi**, or **bx** command is given a number. This number is displayed with the **lb** command and is used with the **d** command to delete the breakpoint.

In Screen Mode

The current position is the line in the source window that contains the cursor. When the cursor is located in the source window, typing **b** is the same as a line mode **b** command without any parameters. That is, a breakpoint is set on the line containing the cursor. The screen-mode **b** command is immediately acknowledged by the appearance of = to the left of the line on which the breakpoint is set.

To set a breakpoint at the beginning of a subprogram while in screen mode, position the cursor on top of any letter of any occurrence of the name of a subprogram and press **B**. An acknowledgment message is displayed at the bottom of the screen to indicate that the breakpoint has been set.

To set a conditional breakpoint in screen mode, type : and enter the line-mode command.

In instruction submode, **b** is interpreted as **bi**.

bd — (break down) set a breakpoint after the end of the current subprogram

Syntax

```
bd [of task] [begin commands end]
bd [of task] when expression
bd [of task] if expression then commands [else commands] end [if]
bd [of task] if expression else commands end [if]
```

Description

bd sets a breakpoint in the subprogram that called the current subprogram (i.e., one frame down from the current frame). The breakpoint is reached *immediately* when the current entity returns. The current subprogram is the one represented by the current frame. *Immediately* means that the **bd** breakpoint is set in the first machine instruction following the current subprogram's return. This may not be on a source statement boundary. To get to the beginning of the next statement, use the **a** command.

The **bd** *command is case sensitive.* If **bd** is given, the breakpoint is removed automatically when it is reached. If **bD** is given, the breakpoint remains until explicitly removed. **bD** breakpoints reached are indicated with a hyphen next to the breakpoint number when listed by the **lb** command. The hyphen also briefly appears when the breakpoint is reached.

bd is usually used for stopping at the end of the current subprogram after having entered it by mistake.

The simplest form of this command, **bd**, is most often used. But, as with the **b** and **bi** commands, the **bd** command can be specified for a particular *task* using the **of** *task* clause. A set of commands can be automatically executed at the breakpoint by specifying a **begin** *commands* **end** block. Set a conditional **bd** breakpoint by using a **when** or **if** statement.

For details on conditional breakpoints, **see also** *[DEBUGGER REFERENCE]*, **b**, page 20–9.

In Screen Mode

Precede this command with : and follow with <RETURN>.

bi — (break instruction) set a breakpoint at a machine instruction address

Syntax

bi [*instruction*] [of *task*] [begin *commands* end]
bi [instruction] [of task] when expression
bi [instruction] [of task] if expression then *commands* [else commands] end [if]
bi [instruction] [of task] if expression else *commands* end [if]

Description

bi sets a breakpoint at a specific machine instruction. The instruction address should be given as a hexadecimal number with a leading 0. **li** or **wi** can be used to get a display of the instructions (disassembly) to indicate exactly where to set the instruction breakpoint.

bi breakpoints can be set for a particular task using the **of** *task* option, where *task* is a task number obtained using the **lt** command.

Execute a block of debugger commands automatically whenever the breakpoint is reached by appending a **begin–end** block. A conditional **bi** breakpoint can be set by using a **when** or **if** statement.

See also *[DEBUGGER REFERENCE]*, **b**, page 20–9, for a discussion of *Command Blocks and Conditional Breakpoints.*

Each breakpoint set with **b**, **bd**, **bi** or **bx** is given a number. This number is displayed by the **lb** command and is used with the **d** command to delete individual breakpoints.

In Screen Mode

Precede this command with : and follow with <RETURN>.

breakpoints — control a program's execution

Description

A breakpoint is a location (a point) in a program where the debugger is instructed to suspend (to break) the program's execution. The debugger has four commands that set breakpoints: **b**, **bi**, **bd** and **bx**. When execution commands are given (**g**, **gx**, or **r** commands), the debugger ensures that when execution reaches set breakpoints, the program 'breaks' — that is, the program stops executing.

While the program is running, the debugger does not accept commands, but input to the program or the debugger can be typed ahead. When the program reaches a breakpoint, its execution is suspended. In line mode, the debugger announces the breakpoint as shown in the following example.

```
[2] stopped at "/vc/sbq/tst3/hs.a":95 in check
95  i : integer := 256;
```

The first line of the announcement begins with a breakpoint number in brackets ([2] in the example above). The remainder of the announcement message pinpoints the location of the breakpoint. The name in quotes is the name of the source file and the number following the colon is the line number in that file where the program stopped. The name following the word *in* is the name of the subprogram (package or task) that contains the source line.

See also *[DEBUGGER REFERENCE]*, **lb**, page 20–44 and **d**, page 20–26.

In Screen Mode

Breakpoints are not announced explicitly in screen mode. The debugger scrolls the source window if necessary to insure that the line containing the breakpoint is on the screen. Since the breakpoint is also the current home position, it will be marked with a * and the cursor will be placed on the line.

As usual, the debugger signals that it is waiting for input by putting a * in column 1 or 2 of the dashed line. There is no * in either of these columns while the program is running, and the debugger does not immediately act on new commands.

Typing ahead in screen mode is not recommended: input to the program and input to the debugger are easily confused on the screen, and typing ahead will usually place unwanted command characters in the source window. (Use <CONTROL>**r** to refresh the screen.)

bx — (break exception) set a breakpoint when an Ada exception occurs

Syntax

```
bx [exception] [of task] [begin commands end]
bx [instruction] [of task] when expression
bx [instruction] [of task] if expression then commands [else commands] end [if]
bx [instruction] [of task] if expression else commands end [if]
```

Description

bx sets a breakpoint that is reached when the named *exception* occurs. If the *exception* field is omitted, a breakpoint is announced when any exception occurs.

Like **b**, **bd**, and **bi** breakpoints, **bx** breakpoints can be set for a particular task, using the **of** *task* option, and can be followed by a block of debugger commands to be executed when the breakpoint is reached.

Each breakpoint set with **b**, **bd**, **bi**, or **bx** is given a number. The number is displayed when the breakpoint is reached or when all breakpoints are listed by the **lb** command. The number is used to delete individual breakpoints using the **d** command.

In Screen Mode

Precede this command with : and follow with <RETURN>.

cb — (call bottom) move to the bottom frame of the call stack

Syntax

```
cb
```

Description

cb moves both the current position and the current frame to the bottom or lowest frame on the call stack. For Ada programs, this is the frame corresponding to the main program.

See also Ada RM 10.1(8).

The call stack is represented with the breakpointed subprogram at the top of the stack. Use **cs** to display the call stack.

The debugger prints a one-line display corresponding to the new current frame. This line is the same line that the **cs** command displays for the frame: to the left is the frame number, followed by the name of the subprogram, package or task that the frame represents, followed by the names and values of actual parameters if there are any.

Local variables and parameters in the bottom frame of the call stack can be made visible by using **cb**.

In Screen Mode

Type **cb** to move to the bottom of the stack in screen mode. Pressing <RETURN> afterward is not necessary.

In response to the command in screen mode, the debugger displays the source code surrounding the new home position in the source window. The one-line display mentioned above is shown in the command window.

cd — (call down) move down on the call stack

Syntax

cd [*name*|*number*]

Description

cd moves the current position down one frame on the call stack. If *name* or *number* is specified, the current position moves down to the next frame on the call stack with that name or number. (This moves in either direction — up or down.) The **cs** command displays the contents of the stack, starting with the current frame, and shows the frame numbers.

The call stack is represented with the breakpointed subprogram being at the top of the stack. The call stack is displayed with the **cs** command, starting with the current frame. Each frame is shown with its number.

If *number* is 0, **cd** moves the current position to the home position in the current frame. This is useful after moving away from the current frame, for example, into a new file using the **e** command. The * command is a synonym for **cd 0**.

See also *[DEBUGGER REFERENCE], line numbers*, page 20–46.

After the **cd** command has executed, the new frame becomes the current frame and the line being executed in that frame becomes the home position. The line containing the current home position is always marked with * when it is displayed on the screen in screen mode or by a display command (**l**, **li**, **w**, **wi**) in line mode.

After executing the **cd** command, the debugger prints a one-line display corresponding to the new current frame. This line is the same line that the **cs** command would display for the frame: to the left is the frame number, followed by the name of the subprogram, package or task that the frame represents, followed by the names and values of actual parameters, if there are any.

To display the values of local variables and parameters of a procedure currently active on the call stack, move the *current frame* to that frame on the stack. This makes the local variables visible.

See also *[DEBUGGER REFERENCE], current frame*, page 20–24, and *visibility rules*, page 20–80.

In Screen Mode

Type **cd** in screen mode. Pressing <RETURN> afterward is not necessary.

The *name* option can not be used directly in screen mode, but the *number* option is supported. However, the *number* parameter must precede **cd** (unlike line mode where the *number* follows **cd**). For example, to move to stack frame 5 on the call stack in screen mode, type **5cd**.

In response to the command in screen mode, the debugger displays the source code surrounding the new home position in the source window. The one-line display mentioned above is shown in the command window.

command syntax — express debugger commands

Description

Most debugger commands are of the form

keyword *parameters*

A list of commands can appear on a single line if separated by semicolons.

Commands are executed when <RETURN> is pressed. The single exception is a breakpoint command followed by a block of commands as illustrated below. The breakpoint is set after the final <RETURN> after the **end** keyword.

```
> b 200 begin
?? p subprogram()
?? p variable ; g ; end
```

The debugger uses Ada syntax for comments: characters between the double dash (--) and <RETURN> are ignored.

<RETURN> can be used to repeat the most recent of several commands. The **r** (run) and **g** (go) commands clear the <RETURN> memory until a repeatable command is used again. Each command that can be repeated with <RETURN> is marked by the phrase '<RETURN> repeats' at the far right hand of the *Syntax* section of that command's entry in this reference.

A number can be entered in either decimal or hexadecimal with a leading 0. If the leading digit of a number is a zero, the debugger assumes it is a hexadecimal number (**0123** or **0f2**). If a number begins with a decimal digit (0–9), but contains a hexadecimal digit (a–f), the debugger interprets the number as a hexadecimal number (**12a3** or **9aaf**). Note that a hexadecimal number with a leading hexadecimal digit (**f2**) must be preceded by zero (**0f2**) or it is interpreted as an identifier.

When a name can be either a debugger keyword or a variable name, the debugger interprets it as a keyword. Precede the name with a backslash (\) to force the debugger to interpret it as a variable.

The documentation for **a.db** frequently refers to the INTERRUPT key (<INTR>). The system command **stty** allows this function to be assigned to any convenient key (often <CONTROL>**c**). <INTR> halts the program being debugged, if it is running, or the debugger's current operation. The debugger immediately responds to <INTR> with a prompt for the next command.

Control characters (e.g., <CONTROL>**z**) have their usual meaning.

See also *Operating System Documentation*, **stty**(1), **tty**(4) and *[DEBUGGER REFERENCE]*, **stop**, page 20–75.

cs — (call stack) display the call stack

Syntax

```
cs [reg] [number] [of task]
```

Description

cs displays the call stack, starting with the current frame. *number* is used to specify the topmost number of frames to be displayed (0 means all). The **of** *task* clause is used to display the call stack for a specific task with the *task_number* obtained using the **lt** command.

The leftmost column of the display contains a number for each frame. This number can be used with the **cd** and **cu** commands.

Other stack commands are **cb**, **cd**, **cu**, and **ct**.

If the keyword **reg** is used, an extra three-line hexadecimal dump is provided per frame. The first line shows the values of the PC, FP, and AP registers. (On most machines, the FP and AP registers are the same register). The second line displays a row of 32-bit words, starting with the word pointed to by FP and then moving down to lower addressed words. The third line displays a row of 32-bit words, starting with the word pointed to by the AP and then moving up, to higher addressed words.

In Screen Mode

Type **cs** in screen mode; pressing <RETURN> afterward is not necessary. However, in screen mode the *number* parameter precedes the **cs** (unlike line mode where the number follows **cs**). For example, to see the top three frames of the call stack in screen mode, type **3cs**.

ct — (call top) move to the top frame of the call stack

Syntax

```
ct
```

Description

ct moves the current frame and current position to the top of the stack that is also the breakpointed frame. When a process breakpoints, the current position is initialized to be the top of the stack. Use **cs** to display the call stack.

The debugger prints a one-line display corresponding to the new current frame. This line is the same line that the **cs** command displays for the frame: to the left is the frame number, followed by the name of the subprogram, package, or task that the frame represents, followed by the names and values of actual parameters if any.

Local variables and parameters of the subprogram at the top of the call stack are made visible with **ct**.

In Screen Mode

Type **ct** to move to the top of the stack in screen mode; pressing <RETURN> afterward is not necessary.

In response to the command in screen mode, the debugger displays the source code surrounding the new home position in the source window. The one-line display mentioned above is shown in the command window.

cu — (call up) move up on the call stack

Syntax

cu [*name*|*number*]

Description

cu moves up one frame on the call stack. If *name* or *number* is provided, **cu** moves up to the next frame on the call stack with that name or number.

The new frame becomes the current frame, and the line being executed in that frame becomes the current home position, marked with *.

The debugger prints a one-line display corresponding to the new current frame. This line is the same line that the **cs** command displays for the frame: to the left is the frame number, followed by the name of the subprogram (package or task) that the frame represents, followed by the names and values of actual parameters if there are any.

Use "**cu 0**" or "*" to move to the home position in the current frame. This is helpful after moving into a new file using the **e** command.

See also *[DEBUGGER REFERENCE], line numbers*, page 20–46.

In Screen Mode

Type **cu** in screen mode; pressing <RETURN> afterward is not necessary. The *name* option can not be used directly in screen mode, but the *number* option is supported (however the *number* parameter precedes **cu** instead of following it as in line mode). For example, to move to stack frame 5 on the call stack in screen mode, type **5cu**.

In response to the command in screen mode, the debugger displays the source code surrounding the new home position in the source window. The one-line display mentioned above is shown in the command window.

current frame — current position on the call stack

Description

The current frame is the current position on the call stack. When a breakpoint is announced, the current frame is always set to the breakpointed subprogram, package or task. This frame is the topmost frame of the call stack. In moving up and down the call stack (**cd** and **cu** commands) the current frame is the frame moved to most recently. The current frame is always the topmost frame of a **cs** command listing.

Each frame has a current instruction associated with it. For the topmost frame, this is the instruction that will be executed next. For all other frames it is the call instruction that called the next higher frame. This instruction address, plus the source line and file name that generated it, constitute the home position for that frame. In showing disassembled instructions, the home position (marked with *) is displayed next to the current instruction for the current frame.

The current frame can only be changed using the call stack commands (**cu**, **cd**, **ct**, and **cb**) or by running the program to a new breakpoint.

The current frame plays a central role in establishing what program names are currently visible at any point during a debugging session.

See also *[DEBUGGER REFERENCE], visibility rules*, page 20–80.

Whenever a breakpoint is reached or a call stack command executed, the current position is always set equal to the home position of the current frame. Examining the program source or instructions changes the current position. Return to the current frame by typing

```
*
```

This command is described in the section on **line numbers**. It has the effect of moving the current position back to the home position of the current frame.

See also *[DEBUGGER REFERENCE], current position*, page 20–25.

current position — current position in a source file

Description

The current position is represented by a three-part identifier consisting of file, line number, and instruction address. This identifier represents the present location in the source program. The debugger keeps track of the current position to make certain frequently used commands more convenient. In particular, the commands **b**, **l**, **li**, **w**, and **wi** use the current position as a default parameter. This section explains how the current position is initialized, how it is changed, and how to find out what it is.

When a breakpoint occurs or after a call stack command is executed, the current position is set to the home position of the current frame. For the top frame of the call stack, this corresponds to the breakpoint. For all other frames, the home position is the location of the call to the next higher frame. The current position can be changed using the call stack commands (**cu**, **cd**, **ct**, and **cb**), the **e** command, the line number command, the listing commands (**l**, **li**, **w**, and **wi**), and the searching commands (/ and ?)

The **e** command is the only command that permits changing the current position into an arbitrary file. The **e** command, without any parameters, displays the current position (except the instruction address).

In line mode, if the source line containing the current position is displayed, a < appears to the left of the source line.

In Screen Mode

The current position is always the line in the source window that contains the cursor.

d — (delete) delete breakpoints

Syntax

`d all|`*breakpoint_number* [, *breakpoint_number*]...

Description

d deletes the listed breakpoints. All breakpoints are deleted if **all** is used.

Multiple breakpoint numbers comprising the *breakpoint_number* (obtained using **lb** and displayed in brackets) must be separated by commas in this command as illustrated below.

```
d 1, 2, 3
```

A breakpoint can be deleted at any time.

In Screen Mode

d deletes a breakpoint set on the line in the source window that contains the cursor. If a source line contains a breakpoint, = appears to the left of the line.

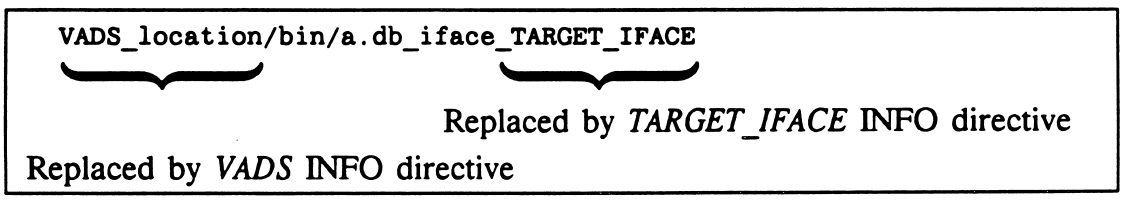
Delete a breakpoint by moving the cursor to a line with = and typing **d**. Pressing <RETURN> is not necessary. The = will disappear, showing that the breakpoint is gone.

DB_IFACE — interface between cross debugger and target [*cross compilers only*]

Description

To enhance portability, the debugger employs a subprocess to communicate with the target processor board.

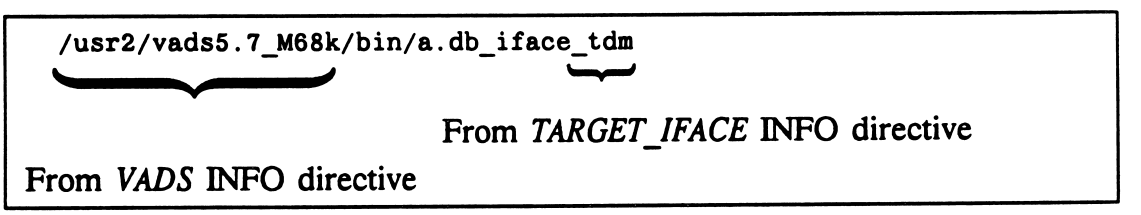
The name of the interface process is determined from the value of the DB_IFACE INFO directive. This value must be the full pathname of the desired interface process. If it is not, a further library search determines the values of the INFO directives VADS and TARGET_IFACE and constructs the DB_IFACE process name in the form:



In the absence of a TARGET_IFACE directive, the value TDM is assumed. For the following visible directives

```
VADS:info:/usr2/vads5.7_M68K:
TARGET_IFACE:info:tdm:
```

the interface process executable name would be:



DB_IFACE Process Startup

The debugger prints out the name of the DB_IFACE process and its invocation before executing it.

```
db_iface process: /usr2/vads5.7_M68k/bin/a.db_iface_tdm
```

If **input** or **output** is set to **pty** (**see set**, page 20–70) , the interface process is invoked using the pseudo terminal drivers. After invocation, the interface process prints

```
This is db_iface!
```

to signal that the process has been correctly started. Information is then exchanged between the process and the target board. Correct communication and initialization is signaled by the message

```
initialized ok
```

and the debugger prompt appears.

Much of the implementation of DB_IFACE is shared by both DB_IFACE and the tool **a.run**. The following discussion really applies to both DB_IFACE and **a.run**.

The DB_IFACE process is also responsible for downloading the memory image to the target. Memory is downloaded in 4K chunks and each chunk will be shown with either a '.' or an '='. A '.' means the memory had to be downloaded. An '=' means that the checksum of the memory on the target was exactly the checksum of the memory to be downloaded, so the download is not performed. This optimization saves a lot of time if TDM is being used and the same program is being run over and over, for example when it is being debugged. In the following example, the image is being downloaded for the first time:

```
>load
downloading constants          20000 ..  203DF
 .
downloading code               203E0 ..  2BE77
 ............
downloading constants          2BE78 ..  2DB5F
 ..
downloading data               2DB60 ..  2DF4F
 .
zeroing memory                 2DF50 ..  2E2EF
>
```

Normally when you re-download, the code portion of the program hasn't changed, so it won't need to be re-downloaded. The same applies to the constants, since they are read-only. The other sections may need to be re-downloaded, however. Here's a second example of a download after the program has been run:

```
>load
downloading constants          20000 ..  203DF
=
downloading code               203E0 ..  2BE77
============
downloading constants          2BE78 ..  2DB5F
==
downloading data               2DB60 ..  2DF4F
 .
zeroing memory                 2DF50 ..  2E2EF
>
```

When DB_IFACE is interfacing between the debugger and an in-circuit emulator, DB_IFACE (and **a.run**) will sometimes require an emulator initialization file. This file is identified by the environment variable VADS_EMU_CONFIG, which should be the name of a file that contains emulator commands to initialize the emulator.

display memory — display raw memory

Syntax

```
@number                          [:display] [number]
hexadecimal_address              [:display] [number]
decimal_address                  :display [number]
name                             :display [number]
```

Description

With a leading @, the number can be either decimal or hexadecimal (with a leading zero).

In the last form, *name* is an Ada object. It can also be a register name of the form **$***name* (i.e., **$eax, $ebp**). *name* is evaluated and the address of the named object is used.

Note: if *name* is a debugger keyword, it must be preceded by a backslash or it will result in a syntax error (e.g. b := 3 will result in a syntax error because *b* is a debugger keyword. Debugger keywords are listed on pages 20–1 and 20–2).

In all four cases, the value to the left of the colon is used as the address of where to start dumping memory.

See also *[DEBUGGER REFERENCE]*, **p**, page 20–54.

display consists of one or two characters from the following list that determine the length and format of each value to be displayed. Either or both can be used. If both are used, the length character must be first. The default is shown in brackets.

B	8-bit [default number base established by **set obase**]
D	64-bit floating point
F	32-bit floating point
L	32-bit [default number base established by **set obase**]
W	16-bit [default number base established by **set obase**]
a	show the address of the item
c	ASCII character
d	decimal [32 bits]
f	floating point [32 bits]
m	one line of STORAGE_UNITS, first in hexadecimal, then as ASCII characters

n	like **m** but bytes are interpreted in reverse order
o	octal [32 bits]
p	hexadecimal pointer [32 bits]
r	reverse-map the address to a procedure name
s	null-terminated (C-style) string
x	hexadecimal [32 bits]

The *number* field determines how many values are to be displayed. The address is advanced by the corresponding amount for the letter being used. For example, to display 8 16-bit decimal values starting at address **01ac4**, type the following command.

```
>10a9a:Wx 8
010a9a: 2074  6865  204e  2071  7565  656e  7320  7072
>
```

The default length is 32 bits (except on MIL-STD-1750A targets where it is 16). The default format as determined by the **set obase** command is either decimal, octal, or hexadecimal. The default count is 1. Specifying an address of the form *@number* or *hexadecimal_number* causes 32 bits to be displayed in the current output base.

The following command will display 2 lines of 16 bytes each, beginning at address 01a9a. Each line is displayed first in hexadecimal and then as an ASCII string.

```
>010a9a:m 2
010a9a: 20 74 68 65 20 4e 20 71 75 65 65 6e 73 20 70 72 " the N queens pr"
010aaa: 6f 62 6c 65 6d  a  0  0  0  1  0  0  0  1  0  0 "oblem..........."
>
```

See also *[DEBUGGER REFERENCE]*, **set**, page 20-70, for a discussion of **set obase**.

The **:a** format is useful for finding the address of any name expression. For example, given the array object MY_ARRAY, the command

```
MY_ARRAY(1):a
```

displays the address of the first element. The **:a** format can also be used for expanded names, selected names, etc.

NOTE
Simply typing a plain decimal number moves the current position to that line number. Memory displays require the colon to be followed by a display letter. The **p** command symbolically displays the value of variables or name expressions; a command consisting of only a name expression is a syntax error.

e — (enter) move to a new source file

Syntax

e [*subprogram|ada_source.a*]

Description

e provides a convenient way to move the current position to a new file or subprogram within a file. If a file is specified, the current position becomes the beginning of the file. If a subprogram is entered, the current position becomes the first line of the subprogram in the source file.

The subprogram name need not be directly visible. For purposes of the **b**, **e**, **edit**, and **vi** commands, all subprogram names in the program are searched. Expanded names can also be given.

For a discussion of expanded names, **see also** Ada RM 4.1.3(13).

If the *subprogram* name has multiple definitions of the same simple name within the context in which it is looked up, it is is overloaded. The debugger prints a diagnostic showing the alternatives. Retype the **b** command attaching a **'1**, **'2**, ... suffix (matching an alternative shown in the diagnostic) to the subprogram name to disambiguate it, or use an expanded name.

See also *[DEBUGGER REFERENCE], overloading*, page 20–53.

File Names

File names that contain only alphanumeric characters, dots, and underscores need not be enclosed in quotes, but those containing other characters must be enclosed in quotes. Enclosing a file name in quotes (**"file.a"**) often eliminates errors, for example, if part of the file name collides with a keyword or program variable. In addition, the debugger interprets shell environment (exported) variables if the file name is enclosed in quotes,

```
e "$al/foo.a"
```

In Screen Mode

Invoke the **e** command by positioning the cursor on any character of a subprogram name and typing <CONTROL>]. This has the same effect as typing **e** *ada_source.a* in line mode — the source window is rewritten with the source code corresponding to the named subprogram.

e can also be used in screen mode by preceding the command with : and following with <RETURN>.

edit — invoke the editor on a subprogram or file

Syntax

`edit [`*subprogram*`|`*ada_source.a*`]`

Description

edit invokes the editor with the specified file or the file containing the specified subprogram. If no parameter is given, the file containing the current position is used.

The debugger consults the environment (exported) variable EDITOR for the name of the editor. If EDITOR is not defined, the debugger uses **vi** as the editor.

The parameters are interpreted exactly as they are for the **e** command.

In Screen Mode

Precede this command with : and follow with <RETURN>.

examine — display variables, files, debugger parameters, etc.

Description

variable values	**p**
files of source code	**e, edit, l,** or **w**
machine instructions	**li** or **wi**
current breakpoints	**lb**
active tasks	**lt**
call stack of currently active subprograms	**cs**
current debugger parameter settings	**set all**

exit — terminate the debugger session

Syntax

```
exit
```

Description

exit exits from the debugger. **quit** can also be used to leave **a.db**.

In Screen Mode

Precede this command with : and follow with <RETURN>.

expressions — arithmetic expressions in the debugger

Description

The debugger supports Ada operator syntax for arithmetic expressions shown below.

+	>=
-	<
*	<=
/	/=
**	=
>	

Operands can be numbers (integers or floating point), program variables, or function calls. More than one function call can appear in the same expression, functions calls can be used as parameters to other function calls, etc.

In an expression, the debugger implicitly converts an integer number to a floating point number if the integer is one operand of a binary operation (+, -, *, /) and the other operand is a floating point number.

The comparison operators in the following list return an integer value of either 0 (FALSE) or 1 (TRUE).

>	<=
>=	/=
<	=

Currently, the final value of an expression must have **type** INTEGER or **type** FLOAT. Also, any operand to one of the above operators must be an integer or floating point value. Ada access values are currently converted to 32-bit integers.

files — specify files to be used by the debugger

Description

The **invocation** section explains how to control which executable file is being debugged.

For example, assume *pathname* is the name of a VADS directory. When debugging a program generated from

pathname`/foo.a`

the debugger uses the net files in

pathname`/.nets/foo*`

produced by the Ada compiler to obtain most of the Ada symbolic information for **foo.a**. Other important files are in *pathname*/**.lines/foo***. These are the line number files, containing a mapping between line numbers and instruction addresses for all the object code generated from **foo.a**.

The commands **e**, **vi**, and **edit** accept a file name. The file name can be contained in double quotes. Simple file names, containing only alphanumeric characters, dots, and underscores need not be surrounded by quotes — they can be typed in directly (**foo.a**, **baz.exp_cmd.a**). For the **e**, **vi**, and **edit** commands, the debugger interprets shell environment (exported) variables for file names, but only if the file name is enclosed in double quotes.

The **r**, **set**, and < commands can also contain file names. These should not be enclosed in quotes. The debugger also interprets tilde and environment (exported) variables in file names in these commands.

g — (go) continue executing the program

Syntax

g

Description

g continues executing the program from where it last stopped. If the process is breakpointed because of a UNIX signal, **g** continues the process *ignoring* the signal. Use the signal **gw** to continue the program while watching for a variable value to change. The single stepping commands **s**, **si**, **a**, **ai** also execute the program, but only for one source line or instruction.

If the program has not started execution, the **g** command can be used to run the program, although no invocation processing (processing of I/O redirection, options, and other parameters used by the program) is done. If the program has exited or terminated, the **g** command reruns the program, using the invocation parameters used the last time the program was run. To rerun a program from the beginning at any time, use the **r** command. **r** also accepts **csh**-like invocation parameters.

g cancels the <RETURN> key memory.

See also *[DEBUGGER REFERENCE]*, **<RETURN>**, page 20-61.

In Screen Mode

Typing **g** when in screen mode causes the program to continue; pressing <RETURN> is not required. With **set safe on**, the screen mode command becomes **gg**.

See also *[DEBUGGER REFERENCE]*, **set**, page 20-70.

gw — (go while) continue executing the program until a variable changes

Syntax

gw name|address [, number]

Description

gw causes the program to execute until the value of the named variable changes. If an address (*@number*) is used, the specified number of bytes (*number*) at that address is watched for change (4 is the default). *number* can be from 1 to 16 inclusive.

This breakpoint is quite useful; however, it can be quite slow since the value being watched is checked after the execution of each machine instruction.

In Screen Mode

Precede this command with : and follow with <RETURN>.

gx — (go signal) continue execution and pass the signal to the program

Syntax

```
gx
```

Description

When a UNIX signal occurs in a program being debugged, it is passed first to the debugger. The debugger announces the signal and the location at which it occurred and stops, waiting for commands. To continue execution of the program as though the signal had not occurred, use **g**, **a**, **ai**, **s**, or **si**. To continue execution and pass the signal to the program, use the **gx** command. This is useful in debugging programs that do explicit signal handling.

See also *[DEBUGGER REFERENCE]*, **set**, page 20–70, for a discussion of set signal.

This command cancels the <RETURN> key memory.

See also *[DEBUGGER REFERENCE]*, **<RETURN>**, page 20–61.

In Screen Mode

Precede this command with : and follow with **<RETURN>**.

help — print help text

Syntax

```
help [subject]
```

Description

On-line help is available for debugger commands and concepts and other VADS tools.

Access on-line help during a debugging session by typing

```
help [subject]
```

If the subject is omitted, a list of debugger commands for which help is available will be printed. This overview can also be obtained by typing **intro** after a help prompt. Otherwise, information on the specified subject is shown.

The pager defined by the environment (exported) variable PAGER will be used to display the text of on-line help. If PAGER is not defined, the default is **pg** for System V UNIX. **See also** UNIX **pg**(1).

In Screen Mode

When using the debugger in screen mode, the command

```
:help [subject]
```

will provide the same aid.

An additional help tool is available in screen mode by typing

```
H
```

which prints a line of information at the bottom of the screen. This is the first line of several single lines of help text that provide a brief summary of the screen-mode commands. Press **H** again for the next line of help text.

home position — execution point in current frame

Description

The home position is the execution point in the current stack frame: the next instruction to be executed. The line containing the home position is marked with an asterisk.

When program execution moves to a new frame, the home position changes as well. The initial home position in a new frame is always a **call** instruction since each frame on the call stack, except the top frame, is calling the next higher frame.

The asterisk is recognized in line mode as being a valid line number meaning 'the line containing the home position'. An asterisk can be used as a command by itself to move the current position to the home position.

See also *[DEBUGGER REFERENCE], line numbers*, page 20–46, and VADS *Debugger, 8.8.1 Call Stack, Home and Current Positions*, page 8–7.

l — (list) display part of a source program

Syntax

l [*line*] [, *number*] <RETURN> repeats

Description

l lists a specified number of lines of the current source file starting at the specified *line*. The default value for *number* is 10, but this can be changed using the set command. The default line is the current line, which is marked with < to the left.

The possible forms of *line* are shown below.

number	move to specified line
+number	move *number* lines forward
–number	move *number* lines backward
*	move to home position

See also *[DEBUGGER REFERENCE], line numbers*, page 20–46.

After the l command is executed, the current line is the last line displayed. Consequently, typing l without parameters continues listing where the last l command stopped.

If * appears after the line number, that line is the current home position for this file. If = appears after the line number, there is a breakpoint set for that line.

If – appears after the line number, there is a breakpoint set in the code generated for this line, but not on the first instruction. If both = and – apply to the same line, the one most recently set is displayed.

In Screen Mode

Precede this command with : and follow with <RETURN>.

lb — (list breakpoints) list all currently set breakpoints

Syntax

```
lb
```

Description

lb lists all currently set breakpoints. To the left of each breakpoint is a number in brackets, which can be used to delete the breakpoint.

In Screen Mode

Precede this command with : and follow with <RETURN>.

See also *[DEBUGGER REFERENCE]*, **d**, page 20–26.

li — (list instructions) list disassembled instructions

Syntax

`li` [*line|instruction*] [, *number*] <RETURN> repeats

Description

li lists a specified *number* of lines including disassembled machine instructions interspersed with source lines. If a decimal line number is given, disassembly starts with the first line of code generated by or after that line. *instruction* stands for an instruction address expressed as a hexadecimal number with a leading zero. If *instruction* is specified, the listing starts with the instruction at that address.

If * appears next to the instruction's address, that instruction is the home position for the current frame.

If an equals sign appears after the address, there is a breakpoint at that instruction.

If [*line|instruction*] is omitted, the listing begins with the line or instruction following the most recently **l**'d or **li**'d line or instruction, or the home line or instruction if no **l** or **li** command has been given for this file. Listing begins with the home position if *line|instruction* is *.

The display contains program source lines interspersed among disassembled machine instructions. The first instruction is preceded by the source line that generated it except when no source is available or disassembly begins mid statement.

In Screen Mode

Precede this command with : and follow with <RETURN>. Disassembly can also be shown in the upper window with *Instruction Submode*.

In screen mode, the upper window (source window) can be toggled to display either source code or disassembled machine instructions. This screen mode command, **I**, performs this toggling.

See also *[DEBUGGER REFERENCE], screen mode*, page 20–63, for a discussion of the command **I**.

line numbers — move to a specified line

Syntax

number	move to specified line
+number	move *number* lines forward
–number	move *number* lines backward
*	move to home position

Description

Some debugger commands accept *number* as a parameter (**b, l, w, li, wi**). In addition, in line mode, typing *number* by itself is the debugger command to move that line.

A line number can be specified as a decimal number, representing that line in the current file. Typing *+number* or *–number* adds or subtracts the specified number from the current position. For example, typing **+5** or **–5** as a command causes the debugger to change the line portion of the current position by adding or subtracting 5, respectively. Typing a * as a line number means 'use the line number corresponding to the current home position'.

When the debugger displays the current home position (in response to an **l, li, w,** or **wi** command), * appears to the left of the line that corresponds to the home position.

In line mode, the debugger displays the current position by showing < to the left of the line corresponding to the current position. The **e** command with no parameters lists out the current file, subprogram, and line corresponding to the current position.

In Screen Mode

Move to a new line in screen mode by using the command shown below.

```
[number]G
```

The **G** must be upper case. This command causes the debugger to move the source window so that it displays the line in the current file whose number was specified. If *number* is not specified, the source window moves to the end of the file. Thus, in screen mode both of the following commands would move to line 102.

```
102G
:102
```

By default the debugger displays source with associated line numbers, (controlled with the **set number** command).

See also *[DEBUGGER REFERENCE]*, **set**, page 20–70, **set number**.

load — download the memory image to the target processor [*cross debuggers only*]

Syntax

```
load
```

Description

The **load** command is for cross debuggers only. Its purpose is to give the user more precise control over when the download operation takes place. **load** downloads the executable file named in the debugger invocation.

Since the runtime system kernel is now an independent executable, it must be downloaded separately to the target using **a.run** before debugging or running a user program.

After downloading a program, the PC is set to the kernel starting address given in the KERNEL_START INFO directive in the **ada.lib** file.

An alternative to the **load** command is the **sync** command. This will attempt to synchronize the cross debugger with the current state of the target processor.

When the self-target debugger is invoked, it will automatically set up a program to run. When the cross debugger is invoked, it does not automatically download a program. It waits for the the user to use the **load** command. As a result, right after debugger invocation many commands cannot be used until either the **load** or **sync** command is given.

lt — (list tasks) list all active tasks

Syntax

```
lt [all | task]
```

Description

If neither **all** nor a task is specified, **lt** lists all active tasks and gives a brief status for each one. *task* can be specified as *, which represents the current task.

If **all** is specified, **lt** gives a detailed display for all tasks.

If *task* is supplied, **lt** gives a detailed display for that single task.

The simple **lt** command displays four columns of information for each task. The columns are labeled Q#, TASK, ADDR, and STATUS.

The Q# column (queue numbers) may have one of several values: R*n* indicates that the task is on the *run queue* in the *n*th position. R1 will run next. D*n* means that the task is on the *delay queue* in the *n*th position. The delay for D1 will expire next.

* indicates the current task (the breakpointed task or the task most recently **selected**).

If no value appears in the Q# column, the task is in some transition or wait state. It could be waiting on a semaphore or it could have been accepted for an immediate rendezvous and will be placed on the run queue when that rendezvous is complete. VADS does not distinguish these states.

See also *[DEBUGGER REFERENCE]*, **select**, page 20–69.

The TASK column contains either the name of the task or a **T** followed by the name of the task type that declared the task. Note that the parent task has no name and is listed as **<main program>**. Also, the runtime itself defines a task used when all other tasks are suspended and the scheduler is waiting for an interrupt event. This task is listed as **<idle task>** when present.

The ADDR column contains the address the run-time system assigns to the task when it is created. This address is used when setting breakpoints for a particular task with the **select** command. The address is also used with the **lt** command when the task is specified. The address value must be used when identifying instances of a task type, as the task type name is not a unique identifier.

See also *[DEBUGGER REFERENCE]*, **b**, page 20–9 **bd**, page 20–12, **bx**, page 20–15 **select**, page 20–69.

The STATUS column shows the state of each task and additional information for some states. There are 11 possible states for each task.

STATUS	DESCRIPTION
`not yet active`	the task has been created, but not yet activated For a discussion of task activation, **See also** Ada RM 9.3.
`awaiting activations`	the task is activating subtasks
`executing`	the task is executing now
`ready`	the task is ready (normally, it will be on the run queue, but might be suspended on a semaphore, for example)
`waiting`	the task may have been accepted for immediate rendezvous, may be in transition, or may be suspended on a semaphore queue
`suspended at accept`	the task has executed an **accept** on an entry that no task is currently calling, so it is now waiting until some task calls it
	`for entry` *entry_name* — produces the name of the accepting entry
`suspended at select`	the task has executed a **select** but no tasks are calling open entries; it is now waiting until some event allows it to proceed
	`(termination possible)` — means that the select statement has an open terminate alternative
	`OPEN ENTRIES:` *entry_name* — lists each open entry of the select
	`NO ENTRIES OPEN:` — means that no entries are currently open
`suspended calling`	the task has executed an entry call and will remain in this state until rendezvous is completed for the call
	task_name [*task_addr*]*.entry_name* — gives the target task and entry
`suspended at delay`	the task has executed a delay statement
`awaiting interrupts`	the task is a special task that is executing an idle loop until interrupted. Normally, this is **<idle task>**
`completed`	the task has executed all its code body. Upon completion of all subtasks, it will terminate **See also** RM 9.4(5).
`terminated`	the task is now terminated. **See also** RM 9.4(6).

Some additional information may be appended to the status entry.

`in rendezvous with` *task_name*`[`*task_addr*`] at entry` *entry_name*

means that the task has accepted the entry call from the named task at the named entry. This message may be given more than once. It is repeated for each rendezvous resulting from outer nested **accepts** in order from innermost to outermost **accept**.

`on delay queue, until` *day: number sec: number*

means that after the runtime *current_time* reaches the given day and duration seconds, this task will become ready for execution. The task may be on the delay queue because of a simple delay statement, a timed entry call, or because of an open delay alternative of an active **select**.

`about to raise` *exception_string*

means that an exception occurred in the called task during a rendezvous and the named exception will be raised in the calling task.

The **lt** command produces output like that in the abbreviated example below.

```
Q#      TASK              ADDR      STATUS

        <idle task>       07a908    not active
        rand_delay        077bd8    suspended at accept for
                                      entry rand
        T philosopher     072178    suspended in rendezvous
                                      fork[063f88].put_down
R1      T philosopher     06c718    ready
D1      T philosopher     0699e8    suspended at delay
                                      on delay queue, until
                                      367600 millisecs
*       fork              063f88    executing
                                      in rendezvous with T
                                      philosopher[072178] at
                                      entry put_down
        dining_room       058ac8    suspended at select on
                                      delay queue, until day: 0
                                      sec: 10,000 open entries:
                                      allocate_seat
        T output          055d98    suspended at select
                                      (terminate possible)
                                      open entries:  put_cursor

        <main program>    045068    waiting to exit
```

When tasks are dynamically created (i.e., anonymous tasks), only their addresses are listed. The task number can be used with the **select** and **cs** (call stack) commands.

The **lt** *task* form of this command provides additional information about a single task. For example `lt dining_room` produces the following output.

```
Q#      TASK            ADDR        STATUS
*       dining_room     058ac8      in rendezvous
                                     with T philosopher[06c718]
                                     at entry allocate_seat
        ENTRY           STATUS      TASKS WAITING
        allocate_seat   open        T philosopher[072178]
                                      T philosopher[06c718]
                                      T philosopher[0699e8]
        enter           open        - no tasks waiting -
        leave           open        - no tasks waiting -
        static priority  = 3
        current priority = 4
        parent task:    <main program>[045068]
                        at line 18 in /vc/sfz/t/ada.tests/phl.a
```

The first lines are the same as the brief display. The added information includes a table of entry queue status giving the entry name, whether the entry is open for a select, and the ordered lists of the tasks waiting at that entry.

The **static priority** line gives the task priority. Following that is **current priority**. A task executes at the higher of its own static priority and the current priority of any task with which it is in rendezvous.

The **parent task** line describes the master of this task. The master block for this task is at line 18 in the file **phl.a**, in the main program. The task that is executing this master is 045068. With this information it is possible to construct a tree of tasks, linked by their masters to their parent tasks. Note that the task executing the master is not necessarily the task that creates it. For further discussion of a master, **See also** Ada RM 9.4.

NOTES: Breakpoints within the runtime system itself may leave the tasking data structures in an unpredictable state. Output from the **lt** command may then be questionable, particularly for the current task. For example, a commonly used breakpoint is SWITCH_TO, which is called whenever control is transferred to a new task. At this point, in the middle of an accept, rendezvous information may not be consistent.

The system clock may continue to run while the debugger is suspended at a breakpoint waiting for input. This may cause delays to expire immediately when stepping away from the breakpoint and the flow of program control to relate to breakpoints and their timing in an unpredictable fashion.

If time slicing is enabled, a breakpoint is often followed by a time slicing transfer of control. This transfer can be pathologic if a breakpoint is set in the actual time slicing mechanism such as in SWITCH or SWITCH_TO. The time slice response calls SWITCH and reaches another breakpoint. The delay in handling the breakpoint uses up the time slice and so another time slicing interrupt comes just as execution resumes from this breakpoint. For this reason, it is convenient when debugging tasks to configure the runtime system without time slicing enabled

```
v_krn_conf.time_slicing_enabled := false;
```

or to call the subprogram in V_XTASKING to turn off time slicing.

```
v_xtasking.set_time_slicing_enabled(false);
```

See also *Implementation Reference, 18.25 VADS Task Control Services, page 18–72.*

overloading — disambiguate overloaded names

Description

a.db supports a subprogram, task, or package name as the object of the **e** (enter subprogram or file), **vi** (enter screen-oriented mode), and **b** (set breakpoint) commands. When debugging Ada programs, the name supplied to these commands need not be fully qualified (the simple name **sin** could be used in place of **standard.math.sin**).

When a simple name is used for one of these commands, the debugger searches 'program-wide' for this name. This search allows the user to set a breakpoint or enter any subprogram even if its name is not currently directly visible.

If the search finds multiple definitions for the same name, the debugger issues a message displaying each of the visible alternatives. The notation is *simple_name'n*, where *n* is a number (e.g., **sin'2**). An example of the debugger message follows. A breakpoint is being set at a subprogram with the simple name **add** which is overloaded.

```
>b add
=> add is overloaded. Use 'n to select one:
add'1 (integer, integer) return integer
add'2 (integer, integer) return real_a
add'3 (integer, integer) return float
add'4 (integer)
add'5 (float)
>b add'3
```

The suffix *'n* must be appended to the subprogram name to indicate precisely the name to be referenced, where *n* is one of the numbers listed in the overloading message (**b add'3**).

The debugger assigns a number to each occurrence of a simple name, starting with 1. These numbers remain constant through out the debugging session. When the debugger finds overloading, all of the overloadings will be listed.

p — (print) display the value of a variable or expression

Syntax

p *expression*

Description

p is the debugger's command for displaying the value of program variables and for calling subprograms. It also evaluates arithmetic expressions that may contain program variables and/or function calls. If the displayed result is an integer, it is printed in the current output base (default, 10). This default can be changed with the **set obase** command.

Name Expressions

For Ada variables, the debugger currently supports simple variable names (MARK, R_A) selected components and expanded names (MARK.LINE), and indexed components (X(Y), Z(1,2)). These can also be used in combination (X(M.Z), B.X(1)). Entry calls and slices are not yet supported.

Ada subprograms and functions can be called. The results of a function can be further used in the expression. There are some restrictions: only parameters of mode **in** are supported. **in out** or **out** parameters are not supported. Default parameter values are not yet supported. The debugger will not call a function that returns an array or a record (functions returning access values *are* supported). A parameterless function must be called using empty parentheses: **p foo()**. When referencing an Ada subprogram, it may be necessary to resolve overloading.

See also *[DEBUGGER REFERENCE], overloading*, page 20–53.

The debugger uses visibility rules similar — but not identical — to Ada for looking up variable names. The current frame determines what names are visible. The visible names are the same names that would be visible if a statement were added to the program at the point in the source text corresponding to the current frame.

See also *[DEBUGGER REFERENCE], visibility rules*, page 20–80.

Expressions

The **p** command also calculates the value of an expression and prints the result.

See also *[DEBUGGER REFERENCE], expressions*, page 20–36.

In Screen Mode

The **p** command has additional support in screen mode. To print the value of a variable on the screen (either window), position the cursor on top of any letter of a variable name and press **p**. In response to the **p**, the name of the variable is displayed in the command window followed by the variable's value. This is handy when single stepping since the cursor is usually near an instance of variables of interest.

To display an entire name expression (**foo(i).link**), position the cursor on top of **foo** and press **P**. This moves the cursor to the **f** of **foo** and overwrites the last letter of **foo** with **@**. The cursor and the **@** delimit the expression. Type **P** again to move the **@** right again to the last character of the next part of the name expression, the right parenthesis of **(i)**. Continue typing **P** in this way until the entire expression to be displayed is delimited by the cursor and the **@**. At this point, a **p** will display the value of the delimited expression.

pass – pass a command to the target controlling agent
[*cross debuggers only*]

Syntax

```
pass text
```

Description

The cross debuggers are implemented by utilizing the support of what is called a *target controlling agent*. A microprocessor emulator, or the Target Debug Monitor (TDM) are target controlling agents. These agents are often powerful debugging stations in their own right and are capable of providing facilities that complement those provided by the VADS debugger.

The **pass** command allows the user to send commands to the controlling agent. All the text following the **pass** keyword is sent as a command to the agent. If any output is generated by the agent, it is displayed on the screen.

Some caution must be exercised when using the **pass** command. When the target processor reaches a breakpoint, the debugger reads state information that it keeps resident on the host computer. A **pass** command could make this information invalid. For example, if the emulator's command to single step were passed to the emulator, the target's real state would be inconsistent with the state the debugger has saved away.

Use the **sync** instruction to synchronize the debugger with the current state of the target processor and the controlling agent.

quit — terminate the debugger session

Syntax

```
quit
```

Description

quit exits from the debugger. **exit** can also be used to leave **a.db**.

In Screen Mode

Precede this command with : and follow with <RETURN>.

r — run the program

Syntax

r [*shell_arguments*]

Description

r runs or reruns the program. If *shell_arguments* is specified, r runs the program as if it were being executed from the shell. The debugger supports a subset of shell command arguments. It supports the following arguments for I/O redirection: >, <, >>, and >& (for redirecting both standard and error output). The debugger supports substitution for shell environment (exported) variables. Additionally, when argument strings contain dollar signs ($), backquotes (‘), or globbing meta symbols (*, ?, []), they are passed to the shell for evaluation. The debugger uses **sh**(1) on System V UNIX or the shell defined in the environment(exported) variable, SHELL. Strings enclosed in double quotation marks are passed as a single argument after removing the quotes. Other parameters are passed (**-o**, **-Pfoo**) just like the shell.

This command starts or restarts the program from the beginning. To continue execution from a breakpoint or step, use **g**.

For more information on restarting a loaded program, **See also** *[DEBUGGER REFERENCE]*, **set**, page 20–70, **set run**.

In Screen Mode

Typing **r** when in screen mode starts or restarts the program executing from the beginning. (Pressing <RETURN> is not required.) In ‘safe’ mode, the command becomes **rr**.

To establish the invocation parameters in screen mode, first type a colon to get a line-prompt and then a full **r** command with I/O redirection, invocation parameters, etc. Alternatively, use **set run** after the colon. The debugger remembers I/O redirection and invocation parameters, so subsequent **r** commands without parameters re-use the parameters of the previous **r** command.

Typing **set run** without any parameters causes the **r** command to ‘forget’ its I/O redirection and invocation parameters.

See also *[DEBUGGER REFERENCE]*, **set**, page 20–70, **set run**.

read — read debugger commands from a file

Syntax

```
read file_name
```

Description

read causes the debugger input source to switch from the keyboard to the named file. Additional **read** commands can occur in the file, but can be nested only to a maximum of four levels. After executing the file, commands are again read from the keyboard (unless the command file contains an **exit** or **quit** command).

Warning: Commands occurring after a **read** command on the same line will not be executed. For example, the **p foo** command in

```
read x_dump; p foo
```

will not be executed.

In Screen Mode

Precede this command with : and follow with <RETURN>.

reg — list the current machine register contents

Syntax

```
reg [all|f]
```

Description

reg lists the contents of registers as they were when the program stopped. Sample output is shown below.

```
   1  -- taskpr1.a
   2
   3  with TEXT_IO; use TEXT_IO;
   4
   5
*-----------------------------------------------taskpr1.a--
EAX:   402eb4   EBX:     d764   ECX:   4029a0   EDX:       1e
EBP: 7ffffdc0   ESI:       14   EDI:   402eb8   ESP: 7ffffdb8
CS:        17    SS:       1f   EIP:     d77c
EFLAGS:   297:  c:1 p:1 a:1 z:0 s:1 t:0 i:1 d:0 o:0 iopl: 0 n:0 r:0
vm:0
```

Some processors are commonly paired with floating point coprocessors with additional accessible registers. For debuggers supporting the option,

```
reg f
```

lists the floating point coprocessor registers if a coprocessor is supported in the implementation.

The command

```
reg all
```

displays all registers.

In Screen Mode

Precede this command with : and follow with <RETURN>.

<RETURN> — re-execute the most recently repeatable debugger command

Syntax

<RETURN>

Description

Pressing the <RETURN> key can be used to repeat the most recent **a**, **ai**, **s**, **si**, **l**, **li**, /, or ? command. Debugging a program with **r**, **g**, or **gx** causes the <RETURN> key to lose its memory until one of the repeatable commands is used again.

In safe mode, this feature is disabled for the **a**, **ai**, **s**, and **si** commands.

See also *[DEBUGGER REFERENCE]*, **set**, page 20–70.

In Screen Mode

Use of the <RETURN> to repeat the functions listed above is disabled in screen mode.

In screen mode, the dot (.) will repeat the previous debugger command line.

See also *[DEBUGGER REFERENCE] Screen Mode*, page *20–63, Window Control Commands*

s — (step) step one source line, into subprograms

Syntax

s <RETURN> repeats

Description

s single steps one line of source code, stepping *into* called subprograms. If the debugger is currently on a source line containing subprogram calls, **s** executes the program up to the point where it calls one of the subprograms. The program then stops inside the called subprogram. The **a** command also single steps one line of source code, but steps *over* subprogram calls.

Subprogram here means procedure, function, separate package body, or instantiated generic package.

If the program has not yet started — for example, just after invoking the debugger — the **s** command starts the program, stepping one source line. For Ada programs, this steps *over* all the library unit elaborations. Other stepping commands are **a**, **ai**, **si**, and **gw**.

Hint

A frequent debugging mistake is to use the **s** command to step into a subprogram when the **a** command to step *over* the subprogram was intended. The **bd** command sets a breakpoint at the place where the current subprogram returns. To recover, use **bd** and then **g**. Execution will usually stop very close to where the **a** command would have stopped. Moreover, in using a lower case **d**, the breakpoint is automatically deleted so it will not be encountered again. (To leave the breakpoint set, use an upper case **D**.) When stopped at the the **bd** breakpoint at the middle of a source statement, use **a** to go to the beginning of the next statement.

In Screen Mode

Typing the **s** when in screen mode single steps the program one source line into called subprograms. Pressing <RETURN> is not required. With **set safe on**, the command becomes **ss**.

See also *[DEBUGGER REFERENCE]*, **set**, page 20–70, **set safe on**.

In instruction submode, **s** is interpreted as **si**.

screen mode — screen-oriented debugger interface

Description

The debugger supports a screen-oriented interface in addition to the more traditional line-oriented interface.

To use the screen interface, either invoke the debugger with the -v option,

```
a.db -v myprogram
```

or use the **vi** command from line mode.

To switch from screen mode to line mode, type the screen command **Q**.

Windows

In screen mode the debugger divides the screen into three windows. The top window is called the source window. It goes from the top of the screen to the dashed line and is used for program source text. The window below the dashed line is called the command window. It is used to display command input to the debugger, text input to the program being debugged, and debugger/program output. The last line on the screen is called the error window and is used for error messages from the screen-mode interface, patterns, etc. Output that appears in the error window does not appear in a log file.

For examples of the screen-mode interface, **See also** *[DEBUGGER REFERENCE]*, **search** *(? /)*, page 20-68.

The top two windows each display a portion of a potentially larger area. For example, the source window displays part of a source file. This window can be moved to display a different part of the same source file or it can display part of a different source file.

The debugger keeps the last 750 lines of text that have been displayed in the command window in an internal history buffer. The command window is typically positioned to display the most recent interactions (the end of the history buffer). However, this window behaves just like the source window in that it can be moved to display a different part of the history buffer.

The command window is automatically paged. If output from either a single debugger command or the user's program would scroll off the top of the window, without requesting user input, this output is stopped and

```
--More--
```

is displayed on the last line. At this point the following responses are valid.

Type

<SPACE>

to print the next window of text. Type

q

to *not* display the remaining output up to the next debugger request for input, but to enter it into the history buffer and to log it, if logging is turned on. Type

g

to enlarge the command window to keep the current text in the command window at the expense of the source window. The window sizes are restored at the next debugger command or by the window control command C without a specified number. Type

<RETURN>

to print one more line, scrolling one line off the top of the command window.

When the screen interface is invoked, the source window is 2/3 of the available screen size, the error window has 1 line, and the command window has the remainder of the screen. The size of the source and command window can be changed at any time using the C command. The debugger remembers the most recent settings over subsequent Q and vi commands.

Entering Commands

Screen-mode commands are those commands that are processed directly by the screen interface. These commands are *not* followed by <RETURN> — they are executed immediately when the entire command has been typed. Some screen commands can be preceded by an optional numeric argument.

There are two classes of screen commands. The window control commands allow manipulation of the window interface. The directly implemented debugger commands are recognized by the screen interface and passed directly to the debugger.

The window control commands are applied to the window that contains the cursor. The , command moves the cursor from source window to command window or the reverse.

Instruction and Source Sub Modes

When in screen mode, the debugger supports two sub-modes, *instruction* or *source*. The I command will toggle the sub-mode, switching from one to the other. In source mode, the source window will display program source

code and the **s** and **a** commands will single step at the source statement level.

Source mode is the default. In instruction mode, the source window will contain disassembled machine instructions, interspersed with source code, if available. Although the source window contains machine instructions, it can still be controlled as usual. In this mode, the **s** and **a** debugger commands will be interpreted as their machine instruction counterparts, the **si** and **ai** commands, respectively. The **b** command will be interpreted as **bi**, setting a breakpoint at the machine instruction under the cursor. All searching and window commands are available.

Screen Prompt

Columns 1 and 2 of the dashed line separating the source window from the command window are used to show the screen-mode prompt, *. The prompt alternates between these two columns and is displayed only when the debugger is awaiting a command. Typing input at any other time is treated as type-ahead and/or input to the program under debug.

The * alternates between the two columns to indicate that a change has taken place in certain, otherwise non-changing, situations. For example, if stopped at a breakpoint in a loop, typing **g** can return to the same breakpoint again. The only change on the screen is the position of the *.

Help

The screen mode **H** command displays a line of help text in the error window. This is one of several lines of help text that summarize all screen mode commands. The next help line is displayed each time the **H** key is pressed.

Window Control Commands

Window control commands control the screen interface. The pattern-matching commands / and ? cause the cursor to move to the bottom of the screen and prompt for the pattern. To terminate the pattern, press either <ESCAPE> or <RETURN>.

Some commands can be preceded by an optional number. When *number* is given for a <CONTROL>**d** or <CONTROL>**u** command (scroll up or down), that number will be used with all subsequent <CONTROL>**d** and <CONTROL>**u** commands until a new number is specified. The initial value of *number* is one-half the size of the window.

The window control commands are shown at the end of this entry.

Debugger Commands

The table at the end of this entry shows the directly implemented debugger commands. These commands can be entered without a preceding : or a following <RETURN> in screen mode. In addition, any line-mode debugger command can be entered by typing a colon first. In response to the colon, the screen interface scrolls up the command window one line, positions the cursor on the bottom line of the command window, and prompts with a colon. Any debugger command can now be entered followed by <RETURN>. After each such debugger command, the cursor returns to the source window.

Screen Interface and set output

The **set output** option allows input and output from the program being debugged in screen mode to be displayed without overwriting the source window. With **set output pty**, the program's output is intercepted by the debugger and displayed in the lower window. Terminal input required by the program is also taken from the lower window. With **set output tty**, the output from a program is presented beginning wherever the cursor is located.

When debugging programs that send cursor-positioning strings to the terminal, it is often useful to alternate between screen and line mode, or to clear the bottom portion of the screen of all cursor-positioning strings by typing :<RETURN> as many times as there are lines in the bottom window and then refreshing the display with <CONTROL>**l** or <CONTROL>**r**.

Window Control Commands

<CONTROL>**b**	(backwards) move backward one full window
[*number*]**C**	(change) change the number of lines in the window; no *number* restores original sizes
[*number*]<CONTROL>**d**	(down) scroll down 1/2 window or *number* lines
<CONTROL>**f**	(forward) move forward one full window
<CONTROL>**g**	print the name of file displayed in source window
[*number*]**G**	(go to) move to bottom or specified line of file
[*number*]**h** or ◀	move left one or *number* columns
H	display the next one-line help message
I	Toggle between **instruction** and **source** submodes
[*number*]**j** or ▼	move down one or *number* lines
[*number*]**k** or ▲	move up one or *number* lines
[*number*]**l** or ▶	move right one or *number* columns
n	(next) repeat the previous / or ? search
Q	leave screen mode; enter line mode
[[	move forward in the source file to the next procedure, function, package, task, or declare block.

]]	move back in the source file to the previous procedure, function, package, task, or declare block.
<CONTROL>r	(redraw) redraw all windows
[*number*]<CONTROL>u	(up) scroll up 1/2 window or *number* lines
/ *pattern*	search forward for pattern
? *pattern*	search backward for pattern
:	enter a debugger command
.	repeat the previous debugger command line
,	move to the other window
^ or **0**	move to beginning of line
$	move to end of line
%	move forward or back to matching parenthesis or brace % also finds the matching **end** when the cursor is placed on: **if, loop, for, while, case, record, select, function, procedure, package**, or **task**. Likewise, % will move the cursor back from **end if** to the corresponding **if**, back from **end** *procedure_name* to the corresponding **procedure**, etc.
*	move the cursor to the current home position

Debugger Commands

a	step to next source line over call statements
b	set a breakpoint on line containing cursor
B	set breakpoint at subprogram name under cursor
cb	(call bottom) move current position and frame to bottom of stack
[*number*]**cd**	(call down) move down one frame or *number* frames on call stack
[*number*]**cs**	(call stack) display entire call stack or just *number* frames
ct	(call top) move current frame and current position to top of stack
[*number*]**cu**	(call up) move up one frame or *number* frames on call stack
d	delete breakpoint at line containing cursor
g	(go) continue program execution
p	display value of variable underneath cursor
P	delimit the expression under the cursor (for use with **p**)
r	(run) start or restart program execution
s	(step) step to next source line going into subprograms
<CONTROL>]	execute debugger **e** command for subprogram name under cursor
<CONTROL>**c**	interrupt the current debugger operation

search — search in current file for a pattern

Syntax

/ pattern	<RETURN> repeats
? pattern	<RETURN> repeats

Description

The / and ? commands search forward and backward, respectively, in the current file, starting at the line following the current line, for the specified pattern. If the pattern is found, the line it is on becomes the current line. The search wraps around; that is, if the pattern is not found between the current line and the end of the file, the file will be searched from the beginning to the current line. The case sensitivity of the search can be controlled through the **set case** command.

The searching commands (/ and ?) use **regex**(3) for pattern matching. This is similar to **vi** and is described in UNIX manuals under **ed**(1).

If no pattern is given, the most recently used pattern for either a / or ? command is used. Pressing <RETURN> causes the command to be executed again.

The line containing the matching pattern becomes the new current position.

In Screen Mode

The searching commands (/ and ?) can be used in either the source or command window. Typing a / or a ? moves the cursor to the the last line on the screen and prompts for the search pattern. The prompt is either a / or a ?. The pattern is terminated by pressing <RETURN> or <ESCAPE>. The command is repeated by pressing **n**.

If the pattern is found, the window is moved to display the line containing the pattern.

The **n** command in screen mode repeats the last / or ? search, but the search will start from the cursor's current location within the current line.

select — choose a new task to be the current task

Syntax

```
select new_task
```

Description

select identifies a task to become the current task. *new_task* is a task number obtained using the **lt** command. **select** makes the call stack of the selected task visible. The call stack commands (**cs**, **cd**, **cu**, **cb**, and **ct**) operate on the current task's call stack.

When the **select** command is executed, the current task is de-selected before the specified task on the **select** command line becomes the new current task. The state of the de-selected task is remembered so that if it is re-selected before the program is further executed, the current frame will be the same as it was when the task was de-selected.

In Screen Mode

Precede this command with : and follow with <RETURN>.

set — set debugger parameters

Syntax

```
set [option [value]]
```

Description

set establishes various debugger parameters. The simple command **set** as well as the command **set all** displays the current settings of all the parameters.

set commands can appear in the **.dbrc** (debugger runtime configuration) file read in during invocation. In addition to normal file names, **set** interprets environment variables for the **log**, **run**, and **source** parameters (as does the **run** command). All **set** commands are valid within the **.dbrc** file, from a command input (**-i**) file, or typed directly from the keyboard.

Options

Options are shown in [brackets].

all	Print current settings of all **set** parameters.
case [on|off]	Make / and ? searches case sensitive. Default: **[off]**
input [pty|tty|*filename*]	Set the input device for the program being debugged. If **pty**, input will be passed by the debugger to your program. If **tty**, your program will read directly from the terminal. If *filename*, your program will read from that file. Default: **[pty]**
lines *number*	Change the number of lines in the display produced by the **l**, **li**, **w**, and **wi** commands. *number* is decimal. Default: [10]
log [off|*file_name*]	Start logging to a file. If a file or **off** is not given, logging is restarted to the log file most recently specified. Default: **[off]**
number [on|off]	In screen mode, determine whether lines in the source window are numbered. Default: **[off]**
obase *number*	Set output number base for displaying commands to 8, 10, or 16. Default: [10]

output [pty|tty|*filename*]

Set output device for the program being debugged. If **pty**, output will be passed to the debugger. In screen mode, the debugger can then put the output in the lower window. Output from your program can also be written on your log file. If **tty**, output will be write directly to the screen, whenever the cursor is. If *filename*, output will be written to that file. Default: **[pty]**

persist [on|off]

The debugger retains the previous file in the source window when the current source file is not available. Default: **[off]**

prompt "*new_prompt*"

Change the debugger's prompt to the specified *new_prompt*, which must be in double quotes. Default: [>]

run *shell_arguments*

Set up the invocation arguments for a program (I/O redirection, options, etc.) but do not start it. Without arguments, **set run** resets the arguments list to have no arguments.

safe [on|off]

Require certain single-letter commands (**a**, **g**, **r**, and **s**) to be typed twice in screen mode for safety when debugging difficult constructs or when excessive line noise is experienced. Default: **[off]**

signal [*signal_list*|all] [b|g|gx]

Set the listed UNIX signals to have the behavior specified by the last parameter. The behavior of each signal may be controlled separately so as to ignore some, and propagate others to the program being debugged. *signal_list* may be a list of the standard signal numbers or names. Use the UNIX command **kill -l** to print a list of signal names. The behavior of two signals, ALRM and CONT may not be changed. Care should be exercised in changing certain signals' behavior, since

```
set signal INT g
```

will prevent interruption of the program.

If you type **set signal** without parameters, the debugger will show the current setting for each signal.
Default: **[all b]**

b When the signal occurs it is announced and the program stopped as if it had reached a breakpoint.

g The debugger will not announce the signal, but will continue the program's execution without reasserting the signal.

gx The debugger will not announce the signal, but will continue the program's execution with the signal passed to it. This is especially useful when debugging programs with exception handlers for hardware exceptions.

source *path_list* Establish the directory search path the debugger uses to find source files [current directory].

tabs *number* Set the tab stop to *number* for listing source (**l**, **li**, **w**, **wi** and screen mode). Default: [8]

sh — spawn a UNIX shell

Syntax

```
sh
```

Description

The **sh** command spawns a shell from the debugger. This way you can temporarily suspend debugging and use the editor, list files, etc. When you exit the shell you will be back in the debugger.

The debugger will use the value of the environment variable SHELL as the name of the shell to invoke. If SHELL is not defined, the debugger will use **/bin/sh**.

In Screen Mode

Precede this command with : and follow with <RETURN>.

si — (step instruction) single step one machine instruction into program

Syntax

`si` <RETURN> repeats

Description

si single steps the program one machine instruction and steps *into* a called subprogram. This is in contrast to the **ai** command, which single steps one machine instruction but treats the machine's *call* instruction as a single instruction, stepping *over* it. Other stepping instructions are **s**, **a**, **ai**, and **gw**.

If the program has not yet been executed, for example, right after invoking the debugger, **si** starts the program, stepping one instruction. This relocates the current position from the main subprogram to the actual starting subprogram preceding the user's program.

In Screen Mode

Precede this command with : and follow with <RETURN>.

stop — stop the debugger or program

Description

The program being debugged can be stopped by setting a breakpoint. After the **r**, **g**, or **gx** command, the program executes until it reaches the breakpoint or exits. If the program appears to be in an infinite loop, stop it by pressing <INTR>.

To terminate the debugger session, use **exit** or **quit**.

sync — set/reset the debugger's knowledge of the target [*cross debuggers only*]

Syntax

```
sync
```

Description

The **sync** command causes the debugger to flush all information it currently has about the target and re-read this information from the target. The debugger will assimilate the current target's state in much the same way it does when the target hits a breakpoint.

The purpose of the **sync** command is to better support the **pass** command. Using the **pass** command, it is possible to transparently alter the state of the target processor without the debugger knowing about it. If this happens, the debugger will start to behave incorrectly as values read from registers or memory don't correspond to what the debugger expects to be there. The **sync** command is used to tell the debugger to reset its expectations by reading the current state from the target.

The **sync** command can also be used to connect the debugger to the current state of the target processor just after invoking the debugger. Although this is not generally useful, it can come in handy in certain cases. For example, if a target system being executed without the debugger all of a sudden crashes, the debugger can be used to examine the crashed memory image symbolically.

The **sync** command cannot completely restore a debugging situation. In particular, if there are breakpoints set in a memory image and the debugger is exited and reinvoked, a **sync** command will pick up the correct register values, but although breakpoints remain in the memory image, the debugger can not recognize them. Disassembly over such breakpoints will be incorrect; attempts to continue execution past such breakpoints will fail.

task — display the address of the current task

Syntax

```
task
```

Description

The **task** command displays the number of the address, which is used, for example, in the **cs** command.

See also *[DEBUGGER REFERENCE]*, **lt**, page 20–48.

In Screen Mode

Precede this command with : and follow with <RETURN>.

terminal control — catching program input/output

Description

The principal function of the **set output** command is to prevent the output of a program from overwriting the source window when debugging in screen mode. If you use **set output pty**, output from your program will be printed in the lower window. If you use **set output** *filename*, output from your program will be written to *filename*.

The debugger uses the pseudo terminal drivers (**ptys**) to position itself between the user's terminal and the process being debugged. This is controlled by **set input** and **set output**. If input is set to **pty**, the default standard input (file descriptor 0) for the user's process is a pseudo terminal. Likewise, when output is set to **pty**, the default standard output (file descriptor 1) is the pseudo terminal, not the control terminal. The debugger does all I/O on behalf of the process being debugged. When **input** or **output** is set to **tty** or to a file, the process being debugged has file descriptors for the actual terminal or file (assuming no I/O redirection has been given with the **r** command).

If you are logging your debugging session, and **input** is set to **pty**, all input to your program will appear in the log. And if **output** is set to **pty**, output from your program will appear in the log.

The (default) use of **ptys** should be transparent to most programs. However, when debugging a program that makes use of special video terminal features this handling may not be desirable.

For more information about the pseudo terminal drivers, **See also** *UNIX Programmer's Manual,* **pty***(4)*.

vi — (visual) switch the debugger into screen mode

Syntax

`vi [`*subprogram*`|`*file*`]`

Description

The debugger supports two interactive interfaces: a conventional line-oriented interface, called 'line mode', and a screen-oriented interface known as 'screen mode'. The **vi** command switches from line mode into screen mode. Once in screen mode, the **Q** command switches into line mode.

The debugger starts in line mode by default. Invoking the debugger with the **-v** option starts it in screen mode.

The parameters to **vi** are interpreted exactly like the parameters to the **e** command. When switching to screen mode, the top part of the screen will display source code. This shows the source code that surrounds the current position. If a file or subprogram name is specified, the corresponding source code is displayed instead of the current position.

For a description of the *file* and *subprogram* options, **See also** *[DEBUGGER REFERENCE]*, **e**, page 20-31.

See also *[DEBUGGER REFERENCE]*, *screen mode*, page 20-63.

visibility rules — determine which identifiers are visible at a breakpoint

Description

The debugger's visibility rules determine which identifiers are visible at a given point in the program's execution. The current frame establishes the current visibility rules using the following model.

At any time during a debugging session, the visible program names are those that would be visible upon adding a statement to the program at the point in the source text corresponding to the current home position in the current frame.

If the current frame is an Ada subprogram, package, or task, Ada visibility rules are used.

However, since debugging is not the same as programming, these rules are extended when the debugger is searching for a subprogram name or one of the commands **b**, **e**, **edit**, or **vi**. As described in the section on overloading, if a *simple_name* is used in one of these commands, but is not found using the model of visibility described above, the debugger searches program-wide for the name. That is, it looks at all the subprograms, tasks, and packages in the entire program, for one whose *simple_name* matches the name typed in for the command.

For more information about names, **See also** Ada RM 4.1(2).

w — (window) list a group of source lines surrounding a line

Syntax

w [*line*] [, *number*]

Description

The term 'window' is used here to mean a section of source text and should not be confused with the windows of the screen-mode debugger.

w lists a window of source text around the specified line in the current file. If *line* is not specified, the center of the window is the line portion of the current position. If the line specification is *, the line portion of the current home position is used as the center of the window. If line is given, the current position is moved to that line and the window surrounding that line is displayed.

The window is the specified number of lines large. (Default: 10.) The **set lines** command can be used to change the default.

For more information on the display of lines containing breakpoints, **See also** *[DEBUGGER REFERENCE]*, **l**, page 20-43.

The **w** command resets the <RETURN> key memory as if a **l** were typed instead.

In Screen Mode

Precede this command with : and follow with <RETURN>.

wi — (window instruction) list machine instructions surrounding an instruction

Syntax

`wi` [*line*|*instruction*] [, *number*]

Description

wi prints a window of disassembled code that is interspersed with original source code if line number files and the source code are available. The lines of output are those surrounding the specified *line* (decimal number) or *instruction* expressed as a hexadecimal number with a leading zero. The window is the specified number of lines large. If *number* is absent, the current setting of **lines** is used. If line number files and the source code are available, the disassembly is interspersed with the original source code.

See also *[DEBUGGER REFERENCE]*, **set**, page 20–70, **set lines** *number*.

If either *line* or *instruction* is specified, the current position is moved to this point.

In general, disassembly is always confined to the code generated from a single source file. If **wi** is used near the beginning or end of a source file, the window can be shortened in one direction.

For more information on the display of instructions containing breakpoints, **See also** *[DEBUGGER REFERENCE]*, **li**, page 20–45.

wi resets the <RETURN> key memory as if **li** were typed instead.

In Screen Mode

Precede this command with : and follow with <RETURN>.

In screen mode, the upper window (source window) can be toggled to display either source code or disassembled machine instructions. This screen mode command, **I**, performs this toggling.

See also *[DEBUGGER REFERENCE]*, *screen mode*, page 20–63, **I**.

Index

Symbols

.all, display value of data object with p command, 15-12

.prrc, a.pr, printing options file, 9-3

:=, assign value, **a.db** modify memory command, 8-17, 20-8

'REF, 12-5
  implementation defined attribute, 18-13
  machine code insertions, 18-32

* in the dashed line, debugger awaits input, 20-65
  sample debugger session, 15-4, 15-5

* inside the source window, location of execution, 20-63
  See also home position
  sample debugger session, 15-5

<CONTROL>c, stopping the program being debugged, 8-4, 20-75

<RETURN>, repeat debugger command, 8-6, 20-61

Numbers

80387 coprocessor, machine code, 12-9

A

a, advance, **a.db** single step command, 8-19, 20-3

a.cleanlib, reinitialize library directory, 4-5, 19-5

a.das
  disassemble object files, 7-1, 19-6
  produce an assembler listing, 5-7

a.db
  * in the dashed line, debugger awaits input, 20-65
    sample debugger session, 15-4, 15-5
  * inside the source window, location of execution, 20-63
    See also home position
    sample debugger session, 15-5

a.db (*continued*)
  <RETURN>, repeat debugger command, 8-6, 20-61
  address memory directly, 20-6
  arithmetic expressions, 20-36
  breakpoint
    commands, 8-21
    control a program's execution, 20-14
    sample debugger session, 15-8
  breakpoint command
    b, set breakpoint, 8-21, 20-9
    b, set contitional breakpoint, 8-22, 20-10
    bD, break down, 8-23, 20-12
    bd, break down, 8-23, 20-12
    bi, break at instruction, 8-24, 20-13
    bx, break at exception, 8-24, 20-15
    command blocks, 8-22, 20-10
    d, delete, 8-24, 20-26
      sample debugger session, 15-11
    implicit breakpoints, 8-24, 8-25
  call stack, 20-21
    See also **cs**
  command syntax, 20-19
  commands, list, 20-1
  cross debugging, 8-29
    See also DB_IFACE
  current frame, 20-24
  current line, sample debugger session, 15-4
  current position, 20-25
  DB_IFACE, 8-30, 20-27
  disambiguate overloaded names, 8-8, 20-53
  display
    commands, 8-10, 20-34
    executable, 8-3
    raw memory, 8-18, 20-29
  display command
    cs, call stack, 8-17
      sample debugger session, 15-9
    l, list source statement, 8-10, 20-43
    lb, list breakpoints, 8-11, 20-44
    li, list instructions, 8-11, 8-24, 20-45
      See also instruction and source modes
    lt, list tasks, 8-13, 20-48
    p, print value of variable or expression, 8-16, 20-54
    reg, list register contents, 8-16, 20-60
    task, display current task address, 20-77
    w, display window of source statements, 8-10, 20-81

a.db (*continued*)
wi, display window of disassembled instructions, 8-11, 8-24, 20-82
download the executable, 8-9, 20-47
execution command
a, advance one source statement, 8-19, 20-3
ai, advance instruction, 8-19, 20-7
exit, terminate debugger session, 8-21, 20-35
g, go, 8-20, 20-38
gw, go while, 8-20, 20-39
gx, go with exception, 20-40
gx, go with signal, 8-20
quit, terminate debugger session, 8-21
r, run, 8-20, 20-58
sample debugger session, 15-5
read, read debugger commands from a file, 8-20, 20-59
s, step one source statement, 8-19, 20-62
sh, spawn a UNIX shell, 20-73
si, step instruction, 8-19, 20-74
execution commands, 8-19
files, specify files to be used, 20-37
home position, 8-7
instruction and source modes, 8-28
invocation, 8-2, 19-8, 20-4
invoke screen-mode debugger, 8-27, 20-63
load, download the executable, 8-9, 20-47
modify debugger configuration
set all, 20-70
set case, 20-70
set input, 20-70, 20-78
set lines, 20-70
set log, 20-70
set obase, 20-70
set output, 20-70, 20-71
set persist, 20-71
set prompt, 20-71
set run, 20-71
set safe, 20-71
set signal, 20-71
set source, 20-72
set tabs, 20-72
sample debugger session, 15-4
modify memory
:=, assign value, 8-17, 20-8
:=, assign value, 8-17
names, variables and visibility rules, 8-8
overload resolution, 8-8, 20-53
pass, pass command to target, 20-56
positioning command
cb, call stack bottom, 8-25, 20-16
cd, call stack down, 8-26, 20-17
cs, call stack, 8-26, 20-21

a.db (*continued*)
ct, call stack top, 8-25, 20-22
cu, call stack up, 8-26, 20-23
current frame, 20-24
e, enter, 8-26, 20-31
edit, move current position and edit, 8-27, 20-33
select, select new current task, 8-26, 20-69
specify new position, 8-25, 20-46
positioning commands, 8-25
process startup, 8-31
screen-mode debugger interface, 20-63
screen-mode window commands, 8-29
screen-mode windows, 8-27
command window, 8-27, 20-63
error window, 8-27, 20-63
history buffer, 8-27
source window, 8-27, 20-63
search, 20-68
sync, set/reset debugger knowledge of target, 8-9, 20-76
terminal control, set input, 20-78
terminate debugger session, 8-4, 20-57, 20-75
unhandled exception, sample debugger session, 15-14
variables and visibility rules, 8-8
vi, enter screen-mode, 8-8, 20-79
visibility rules, 20-80
window command syntax, 8-29
window control commands, 20-65
list, 20-66

a.du, summarize disk usage for Ada compiler, 4-5, 19-9

a.error
ada -e, call a.error to process compiler error messages, 5-3
edit output with vi, a.error -v, 5-5
process compiler error messages, 5-3, 19-10

a.help
interactive help utility for VADS, 10-1
interactive help utility for VADS, 19-12

a.info, list or change VADS library directives, 4-6, 19-14

a.ld, VADS prelinker, 6-1, 19-16
linking foreign objects, 6-2

a.ld, VADS prelinker
example, 14-6
resolve references in interface programming, 13-16

a.list, list programs containing no errors, 5-4, 19-18

a.ls, list compiled units, 4-8, 19-19

a.make
recompilations, 6-3
recompile source files in dependency order, 6-3, 19-20

a.mklib
create a VADS library directory, 4-8, 19-23
example, 11-1, 14-1

a.path
display library search list, 4-11, 19-25
example, 14-4
specify changes to search list with qualifiers, 4-12

a.pr, source code formatter, 9-1, 19-26

a.report, report VADS deficiencies, 10-1, 19-29
configuration in installation, 17-3
submitting customer reports, 17-5

a.rm, remove an Ada source file or unit from a library, 4-12, 19-32

a.rmlib, remove a compilation Ada library, 4-13, 19-33

a.tags
create cross reference of units or types, 4-13, 19-34
replace existing tags file, 4-14

a.vadsrc, display available VADS versions and create a default library configuration file, 4-10, 19-36

a.which, find a compiled unit, 4-15, 19-37

AA_GLOBAL_FREE, procedure
memory management, 18-57
pool allocation, 18-60

AA_GLOBAL_NEW, function
memory allocation in RTS, 18-39, 18-55
memory management, 18-56, 18-66

AA_INIT, procedure, memory management, 18-58

AA_LOCAL_FREE, procedure, memory management, 18-58

AA_LOCAL_NEW, function, memory management, 18-57

AA_POOL_NEW, function, memory allocation pool control, 18-63

accept statements, 18-47
task entry calls, 18-46

ada, invoke VADS Ada compiler, 5-1, 6-1, 19-2

Ada
calling Ada objects from other languages, 13-17
compilation units, 4-1
corresponding types between C, Fortran and Ada, 13-3
libraries, 4-1
source file structure, 4-2

ada.lib
contents, 4-4
example, 4-6
initialized by **a.cleanlib**, 4-5
library reference file, 18-33
limit attempts to lock, 18-22

ADAPATH, character limit, 18-22

ADDR column, display address of tasks with debugger **lt** command, 8-14

address clause
supported for objects and entries, 18-19
use SYSTEM.ADDRESS'REF(N) attribute, 18-14

address memory directly, **a.db**, 20-6

addressing mode, machine code insertions, 12-2, 18-31

ai, advance instruction, **a.db** single step command, 8-19, 20-7

align record fields, record representation clauses, 18-16

allocators, memory allocation in VADS Run Time, 18-54

ALLOWED_WASTE, heap allocation, 18-59

already in kernel, panic message, 18-52

Appendix F, Ada RM, Appendix F material included in *Implementation Reference*, 18-1

archive files, 18-33

arith fault in rts, panic message, 18-51

arithmetic expressions in the debugger, 20-36

array types, interface programming, 13-5

assembly language, machine code insertions, 12-1

assign value, :=, **a.db** modify memory command, 8-17, 20-8

assign value, :=, **a.db** modify memory command, 8-17

attribute
'REF, 18-13
SYSTEM.ADDRESS'REF(N), used in place of unchecked conversion, 18-14

attributes

attributes
DURATION, 18-30
FLOAT, 18-29
INTEGER, 18-29
SHORT_FLOAT, 18-29
SHORT_INTEGER, 18-29
TINY_INTEGER, 18-29
types, ranges, and attributes listed, 18-28

avoiding elaboration, use **pragma** SUPPRESS, 13-18

B

b
set breakpoint, **a.db** breakpoint command, 8-21, 20-9
set conditional breakpoint, **a.db** breakpoint command, 8-22, 20-10

bad signal, panic message, 18-53

bd, **bD**, break down, **a.db** breakpoint command, 8-23, 20-12

benchmarking, 16-1
run time performance, 16-2

bi, break at instruction, **a.db** breakpoint command, 8-24, 20-13

bin, place single VADS **bin** on path, installation, 17-2

bin, directory, VADS executables, 18-2

bit manipulation
record representation clauses, 18-17
utilities, 18-68

bit numbering, illustration, 18-18

breakpoint
commands, **a.db**, 8-21
control a program's execution, **a.db**, 20-14
sample debugger session, 15-8

BUILT_IN, pragma, reserved for Verdix, 18-10

bx, break at exception, **a.db** breakpoint command, 8-24, 20-15

C

C
corresponding types between C, Fortran, and Ada, 13-3
debugging C programs, 8-31
edit **/usr/lib/VADS** during installation, 17-3
pragma INTERFACE, 18-25

C_PRINTF, package, 18-7

C_STRINGS, package, variable length string operations, 18-3

CALENDAR, package, 18-3, 18-28

call stack, **cs**, **a.db** positioning command, 8-7, 20-21

called extend stack in kernel, panic message, 18-51

calling Ada from other languages, 13-17

cb, call stack bottom, **a.db** positioning command, 8-25, 20-16

cd, call stack down, **a.db** positioning command, 8-26, 20-17

change
library search list, **a.path**, 4-11
list or change VADS library directives, **a.info**, 4-6, 19-14

change of representation, supported by VADS, 18-20

character set, full textual representation provided, 18-22

CHARACTER_TYPE, package, 18-7

characters in ADAPATH, limits, 18-22

CLOSE_ALL, package, 18-3

code statements, machine code insertions, 12-2, 18-31

CODE_*N* code-statement, place multiple data items in code, 12-8

command blocks, **a.db** breakpoint command, 8-22, 20-10

command file input, **a.db -i**, 8-3

command reference, summary, 19-1

command syntax, debugger commands, 8-5, 20-19

command window, **a.db**, screen-mode windows, 8-27, 20-63

COMMAND_LINE, package, 18-6

COMPACT, function, memory allocation pool control, 18-64

compilation
Ada compilation units, 4-1
compilation order, 6-3
compilation speed, 16-1
separate compilation information, 4-1
units in separate source files, 13-17

compiler
calling conventions, stack, 18-23
invocation, **ada** command, 5-1
limits, 18-22

COMPLEX, package, 18–6

configuring the VADS Run Time System, 17–7

control a program's execution, **a.db**, breakpoint, 20–14

control characters, usual meaning in use with debugger, 8–6

CONTROLLED, pragma, not implemented, 18–11

conversion, **generic function** UNCHECKED_CONVERSION, 18–24

corresponding types, C, Fortran and Ada, 13–3

create
  cross reference of units or types, **a.tags**, 4–13, 19–34
  default library configuration file and display available VADS versions, **a.vadsrc**, 4–10, 19–36
  subdirectory, example, 4–9
  VADS library directory, **a.mklib**, 4–8, 19–23

CREATE_POOL, function, memory allocation pool control, 18–62

critical region, begins with call to **procedure** IFACE_SEMA.ENTER, 18–70

cross debugging, **a.db**, 8–29

cross reference
  Ada types, **a.tags**, 4–13
  Ada units, **a.tags**, 4–13

cs, call stack
  a.db display command, 8–17
    sample debugger session, 15–9
  a.db positioning command, 8–26, 20–21

ct, call stack top, **a.db** positioning command, 8–25, 20–22

cu, call stack up, **a.db** positioning command, 8–26, 20–23

current frame, **a.db**, 20–24

current line, sample debugger session, 15–4

current position, **a.db**, 8–7, 20–42

current priority, display priority of tasks with **lt** *task*, 8–16

CURRENT_POOL, function, memory allocation pool control, 18–63

curses, Ada interface, 13–1

D

d, delete, **a.db** breakpoint command, 8–24, 20–26
  sample debugger session, 15–11

data types, interface programming, 13–2

DATA_1 code-statement, place single data item in code, 12–8

DB_IFACE, **a.db**, 8–30, 20–27

deadlock, panic message, 18–52

DEALLOCATE_POOL, procedure, memory allocation pool control, 18–63

debugger
  array dimensions in a **p** command, limits, 18–22
  as a learning tool, 8–1
  call parameters, limits, 18–22
  commands, list, 20–1
  configuration, 8–9, 20–70
  controlling tasks, 18–75
  display executable, 8–3
  invocation, 8–2, 19–8, 20–4
  run parameters, limit, 18–22
  task creation, 18–42

debugging C programs, 8–31
  a.db screen functions, 8–32
  a.db stack commands, 8–32
  char * variable display, 8–32
  edit **/usr/lib/VADS** during installation, 17–3
  global C variables, 8–32
  names, variables, visibility rules, 8–31
  operations, 8–32
  operators, 8–33
  type checking, 8–32
  warning: structure field name, 8–33

debugging machine code, 12–11
  li, **wi**, **a.db** display instructions, 8–11

delay queue, display position of tasks with **lt**, 8–13, 18–46

delay statements, tasking, 18–45

DIANA intermediate representation, 4–4

DIANA nets, cached to improve performance, 18–9

directives in **ada.lib**
  add a LINK directive, example, 4–7
  delete an info directive, example, 4–7
  INFO
    limit to characters, 18–22
    reported or altered by **a.info**, 4–6
    supported names, 18–35
    TARGET, 18–35

directives in **ada.lib** (*continued*)
VADS, 18-35
VERSION, 18-35
LINK
LIBRARY, 18-36
reported or altered by **a.info**, 4-6
supported names, 18-36
TASKING, 18-36
TASKING, LIBRARY and WITH*n* searched for by **a.ld**, 6-3
WITH*n*, 18-36
limit to characters, 18-22

directory
.imports, 4-4
bin, VADS executables, 18-2
examples, program samples, 18-2, 18-8
info, program samples, 18-2
.lines, 4-4
.objects, 4-4
publiclib, public domain Ada packages, 18-2, 18-7
standard, standard and other predefined packages, 18-2, 18-3
sup, additional executables, 18-2
user_config, RTS configuration files, 18-2
verdixlib, Verdix supported packages, 18-3, 18-6

directory input, **a.db -p**, 8-3

disambiguate overloaded names, **a.db**, 8-8, 20-53

disassemble object files, **a.das**, 7-1, 19-6

disassembly in the debugger, instruction mode, 8-28
See also **li**, **wi**

disk usage information
conserve space in installation, 17-4
corrupted library net files, **a.du -e**, 4-6
estimating VADS requirements, 18-9
summarize disk usage for Ada libraries, **a.du**, 4-5, 19-9

display
a.db display commands, 8-10, 20-34
available VADS versions and create a default library configuration file, 4-10, 19-36
debugger executable, **a.db -sh**, 8-3
library search list, **a.path**, 4-11, 19-25
raw memory, **a.db**, 8-18, 20-29

DO_COMPACT, memory allocation pool control, 18-64

documentation
description, 1-1
how to use this manual, 1-2

documentation (*continued*)
notation
see also's, 1-2
typefaces differentiate code, commands, etc., 1-3
notational conventions, 1-2

dope vector address, parameter passing, 18-23

DURATION, attributes, 18-30

dynamic array types, interface programming, 13-5

dynamic memory allocation, support in VADS Run Time, 18-54

dynamically allocated objects, memory allocation in RTS, 18-38

E

e, enter, **a.db** positioning command, 8-26, 20-31

edit, output of **a.error**, 5-5

edit, move current position and edit, **a.db** positioning command, 8-27, 20-33

edit-compile cycle, **a.das**, 7-1

ELABORATE, pragma, not implemented. *See* Appendix B of Ada RM

elaboration
avoiding, use **pragma** SUPPRESS, 13-18
information gathered by **a.ld**, 6-1

ELABORATION_TABLE, 13-18

enter screen-mode, **vi**, in debugger, 8-8, 20-79

ENTRY_LIST, accept and select statements, 18-48

enumeration types, 18-28

ERRNO, package, defines UNIX **errno** global variable, 18-5

error messages
categories, 5-2
fatal errors, 5-3
general errors, 5-3
internal errors, 5-3
lexical errors, 5-3
run time system, panic numbers and messages, 18-50
semantic errors, 5-3
syntax errors, 5-3
warnings, 5-3

error window, **a.db**, screen-mode windows, 8-27, 20-63

examine, display variables, files, debugger parameters, 20–34
See also display commands

examples, directory, program samples, 18–2, 18–8

exception handler: below bottom of user stack, panic message, 18–54

exception handler: unexpected fpe_code *n*, panic message, 18–53

exception handler: unexpected signal *n*, panic message, 18–53

exceptions
information gathered by **a.ld**, 6–1
reporting, 18–76
unhandled, sample debugger session, 15–14

executable files, 18–33

execution commands, **a.db**, 8–19

exit, terminate debugger session, **a.db** execution command, 8–21, 20–35

expressions, arithmetic expressions in the debugger, 20–36

external subprograms, declare in interface programmng, 13–6

EXTERNAL_NAME, pragma
link with existing libraries, 13–11
program conversion, 13–15
specify *link_name*, 18–11, 18–26

externally compiled units, 4–11

F

file formats, 18–32

file names, quotes required in use with debugger commands, 8–6

FILE_NAMES, package, file name utilities, 18–3

FILE_SUPPORT, package, file activities for standard I/O, 18–4

files, specify files to be used in debugger, 20–37

find a compiled unit, **a.which**, 4–15, 19–37

finding the right object, compile units in separate source files, 13–17

FLOAT, attributes, 18–29

floating point, 12–9

fortran
corresponding types between C, Fortran and Ada, 13–3
pragma INTERFACE, 18–25

free(3), memory allocation service, 18–66

function
AA_GLOBAL_NEW
memory allocation in RTS, 18–39, 18–55
memory management, 18–56
AA_LOCAL_NEW, memory management, 18–57
AA_POOL_NEW, memory allocation pool control, 18–63
COMPACT, memory allocation pool control, 18–64
CREATE_POOL, memory allocation pool control, 18–62
CURRENT_POOL, memory allocation pool control, 18–63
GET_CURRENT_TASK, 18–74
GET_PRIORITY, 18–74
GET_TIME_SLICE, 18–74
GET_USER_FIELD, 18–76
HEAP_POOL, memory allocation pool control, 18–64

G

g, **go**, **a.db** execution command, 8–20, 20–38

generic declarations, 18–14

generic function
UNCHECKED_CONVERSION, 18–24
UNCHECKED_DEALLOCATION, 18–24
memory allocation in VADS Run Time, 18–55
pool allocation, 18–60

GET_CURRENT_TASK, function, 18–74

GET_MEMORY, procedure
calls **malloc**(3), 18–66
pool allocation, 18–62

GET_PRIORITY, function, 18–74

GET_TIME_SLICE, function, 18–74

GET_TIME_SLICING, procedure, 18–75

GET_USER_FIELD, function, 18–76

global variables, interface programming, 13–10

global virtual address space (GVAS), 4–4

gnrx.lib
contents, 4–4
library reference files, 18–33
limit attempts to lock, 18–22

GVAS_table, limit attempts to lock, 18–22

GVAS_table, keeps track of virtual addresses, 4–4

gw, go while, **a.db** execution command, 8–20, 20–39

gx, go exception, **a.db** execution command, 20–40

gx, go with exception, **a.db** execution signal, 8–20

H

halting the program, <CONTROL>c, stopping the program being debugged, 8–4, 20–75

heap memory, memory allocation in VADS Run Time, 18–55

HEAP_ALLOC, package, memory management support, 18–58

HEAP_CONFIG, package, heap allocation, 18–59

HEAP_POOL, function, memory allocation pool control, 18–64

help
  on-line for the debugger, 8–4, 20–41
  on-line help is available, introduction, 1–2

help utility for VADS, **a.help**, 10–1

help utility for VADS, **a.help**, 19–12

HEX, package, 18–4

history buffer, **a.db**, screen-mode windows, 8–27

home position, **a.db**, 8–7, 20–42

HW_FLT_ENABLED, in panic message: No coprocessor emulate suport, 18–54

I

I/O, of concern in memory management, 18–66

IEEE standard floating point, 16–2

iface_*.a, access to library definitions, 18–66

IFACE_ALLOC, package
  interface to memory allocation, 18–4
  memory allocation in the kernel, 18–67
  memory allocation in the RTS, 18–39

IFACE_BITS, package, 18–4, 18–68

IFACE_RAISE, package, 18–77
  interface to memory allocation, 18–4

IFACE_SEMA, package
  definition of semaphore services, 18–40, 18–69
  interface to semaphores, 18–4

IFACE_SEMA.ENTER, procedure, 18–70

IFACE_SEMA.LEAVE, procedure, 18–70

IFACE_TASKOP, package, task control functions, 18–4

IFACE_TASKS, package, interface to task operations, 18–4, 18–73

IFACE_TYPES, package, supplies types for run time entities, 18–4

implementation dependent, VADS compatibility with Ada RM, 18–1

implicit breakpoints, **a.db** breakpoint command, 8–24

IMPLICIT_CODE, pragma, 12–11
  specifies allowed (ON) or disallowed (OFF), 18–11

.imports directory, contents, 4–4

in, mode **in** parameters supported by debugger **p** command, 8–16

in out, mode **in out** parameters not supported in debugger **p** command, 8–16

INFO directive
  delete an info directive, example, 4–7
  example, 4–6
  reported or altered by **a.info**, 4–6

INFO directive
  character limit, 18–22
  supported names, 18–35

info, directory, program samples, 18–2

INLINE, pragma, 12–10
  See also Appendix B of Ada RM
  declare external subprograms, 13–8

INLINE_ONLY, pragma, 18–11

inlines, limits, 18–22

input/output, 18–30

installation
  installation and maintenance guide, 17–1
  location of VADS, 17–1
  net files incompatible with previous versions, 17–4
  requirements, 17–1
  script, 17–3

instruction and source modes, **a.db**, 8–28

INTEGER, attributes, 18–29

INTERFACE, pragma
  supports calls to other languages, 18–11, 18–25
  user must include foreign objects in link, 6–2

interface programming, 13–1, 18–25
  array types, 13–5
  data types, 13–2
  dynamic array types, 13–5
  global variables, 13–10
  linking a non–Ada main program, 13–18
  map parallel data structures, 13–11
  overhead reduction, 13–14
  pointers and address types, 13–6
  program conversion, 13–15
  record types, 13–3
  run time considerations, 13–19
  simple types, 13–2
  string types, 13–6

INTERFACE_NAME, pragma
  declare external subprograms, 13–6, 13–10
  reference other language variables, 18–11
  reference other language variables and subprograms, 18–25

INTERFACE_OBJECT, pragma, reference other language variables, 18–12

intermediate language file, file formats, 18–32

INTERNAL KERNEL ERROR, panic messages, 18–50

interrupt entries on UNIX, task entries and signals, 18–19

interrupt handler packages, in panic message, 18–51

invalid interrupt entry, panic message, 18–52

invalid interrupt to attach, panic message, 18–52

invocation, debugger, 8–2, 19–8, 20–4

invoke
  screen-mode debugger, **a.db**, 8–27, 20–63
  VADS Ada compiler, **ada**, 5–1, 6–1, 19–2

IOCTL, package, interface to UNIX I/O control functions, 18–5

IOCTL_FMT, package, interface to UNIX I/O control functions, 18–5

J

jump table, via branch offsets, 12–8

K

kernel, one of two major parts of RTS, 18–37

KRN_ALLOC, memory allocation in the kernel, 18–67

KRN_ENTRIES, package, memory allocation in the RTS, 18–39

L

l, list source statement, **a.db** display command, 8–10, 20–43

LANGUAGE, package, supplies language interface for UNIX, 18–4

lb, list breakpoints, **a.db** display command, 8–11, 20–44

lexical errors, limits, 18–22

li, list instructions, **a.db** display command, 8–11, 8–24, 20–45
  See also instruction and source modes

LIBC, package, interface to UNIX directory functions, 18–5

LIBRARY, LINK directive, searched for by **a.ld**, 6–3

library
  library hierarchies distinct from OS directory hierarchy, 4–9
  library management tools, 4–1, 4–4
  names changed or reported by **a.path**, 4–11
  remote, 4–11
  search list example, 4–9
  VADS library directories, 4–3
  VADS library directory contents, Illustration, 4–3

library reference file, **ada.lib** and **gnrx.lib**, 18–33

limits
  characters in ADAPATH, 18–22
  characters in INFO directive, 18–22
  characters in WITH*n* directive, 18–22
  compiler and tool limits, 18–22
  debugger array dimensions in a **p** command, 18–22
  debugger call parameters, 18–22
  debugger run parameters, 18–22
  inlines, 18–22
  lexical errors, 18–22
  nested constructs, 18–22
  STORAGE_UNITS, 18–22
  syntax errors, 18–22

line numbers
change number of lines in debugger display, **set lines**, 20–70
specify new position in the debugger, 8–25, 20–46

line-mode, **a.db**, 8–1

.lines directory, contents, 4–4

lines of disassembled instructions
display with **wi**, 8–11
set with **li**, 8–11

LINK directive
add a LINK directive, example, 4–7
reported or altered by **a.info**, 4–6

LINK directive, supported names, 18–36

link with existing libraries, map parallel data structures, interface programming, 13–11

linker options file, 6–3

linking a non-Ada main program, interface programming, 13–18

LIST, pragma. *See* Appendix B of Ada RM

list
compiled units, **a.ls**, 4–8, 19–19
list or change VADS library directives, **a.info**, 4–6, 19–14
programs containing no errors, **a.list**, 19–18

load, **a.db**, download the executable, 8–9, 20–47

lock files, 5–6

lt
list tasks, **a.db** display command, 8–13, 20–48
task activation, 18–44
task creation, 18–42

M

MACH_CONFIG, package, definitions and data structures used by RTS kernel, 18–4

machine code insertions, 18–31
'REF, 18–32
addressing mode, 12–2, 18–31
code statements, 12–2
pragma IMPLICIT_CODE, 18–32
register definitions, 18–31

machine code procedures, disassembled, 7–1

MACHINE_CODE, package, 18–5
listing, 12–11

MACHINE_TYPES, package, definitions of unsigned types, 18–4

main program
as parameter to **a.ld**, 6–1
information gathered on dependent units by **a.ld**, 6–1
restrictions on 'main' programs, 18–14

malloc(3), memory allocation service, 18–66

mathmatical functions, **verdixlib**, directory, 18–6

memory
allocation in run time system (general), 18–38
allocations must be internally consistent, 18–65
display raw memory, **a.db**, 8–18, 20–29
package MEMORY, 18–5

modify debugger configuraion, **set**, 8–9, 20–70

N

name.lib, maps object names, 4–4

names and visibility rules, **a.db**, 8–8

names of library units, 4–10

nested constructs, limits, 18–22

net files
incompatible with previous versions, 17–4
validity check by **a.du**, 4–5

.nets directory, contents, 4–4

new release of VADS, net files incompatible with previous versions, 17–4

no such signal, panic message, 18–53

NO_IMAGE, pragma, suppress generation of image array, 18–12

NOT_ELABORATED, pragma, avoiding elaboration, 13–18

numeric literals, 18–28

O

object
files, 18–33
linking object files with **a.ld**, 6–2
shared object code, 18–15

object disassembler, **a.das**, 7–1

.objects directory, contents, 4–4

on-line debugger help, 8–4, 20–41

optimization, machine code insertions, 12–1

OPTIMIZE, pragma, not implemented, 18–12
OPTIMIZE_CODE, pragma, 12–11
ORDERING, package, 18–6
OS_FILES, package, 18–5
out, mode **out** parameters not supported in debugger **p** command, 8–16
out of kernel heap, panic message, 18–53
overhead reduction, interface programming, 13–14
overload resolution, **a.db**, 8–8, 20–53
overview of VADS system, 3–1

P

p, print value of variable or expression, **a.db** display command, 8–16, 20–54
  .all, display value of data object, 15–12
PACK, pragma
  minimize gaps between components, 18–12
  representation specifications, 18–16
package SYSTEM, specification, 18–27
packages
  C_PRINTF, 18–7
  C_STRINGS, variable length string operations, 18–3
  CALENDAR, 18–3, 18–28
  CHARACTER_TYPE, 18–7
  CLOSE_ALL, 18–3
  COMMAND_LINE, 18–6
  COMPLEX, 18–6
  ERRNO, defines UNIX **errno** global variable, 18–5
  FILE_NAMES, file name utilities, 18–3
  FILE_SUPPORT, file activities for standard I/O, 18–4
  HEAP_ALLOC, memory management support, 18–58
  HEAP_CONFIG, heap allocation, 18–59
  HEX, 18–4
  IFACE_ALLOC
    interface to memory allocation, 18–4
    memory allocation in the kernel, 18–67
    memory allocation in the RTS, 18–39
  IFACE_BITS, 18–4, 18–68
  IFACE_RAISE, 18–77
    interface to memory allocation, 18–4
  IFACE_SEMA
    definition of semaphore services, 18–40, 18–69
    interface to semaphores, 18–4
packages (*continued*)
  IFACE_TASKOP, task control functons, 18–4
  IFACE_TASKS, interface to task operations, 18–4, 18–73
  IFACE_TYPES, supplies types for run time entities, 18–4
  IOCTL, interface to UNIX I/O control functions, 18–5
  IOCTL_FMT, interface to UNIX I/O control functions, 18–5
  KRN_ENTRIES, memory allocation in the RTS, 18–39
  LANGUAGE, supplies language interface for UNIX, 18–4
  LIBC, interface to UNIX directory functions, 18–5
  MACH_CONFIG, definitions and data structures used by RTS kernel, 18–4
  MACHINE_CODE, 18–5
    listing, 12–11
  MACHINE_TYPES, definitions of unsigned types, 18–4
  MEMORY, 18–5
  ORDERING, 18–6
  OS_FILES, 18–5
  PATH, 18–6
  POOL_ALLOC
    memory management support, 18–59
    pool control, 18–62
  POOL_CONFIG, 18–61
  RAW_DUMP, 18–5
  SEQUENTIAL_IO, 18–28
  SHARED_IO, 18–5
    instantiates Ada generic I/O packages, 18–16
  STATUS, supplies definition of UNIX STATUS_BUFFER data type, 18–5
  STRLEN, 18–5
  STRNCPY, 18–5
  SYSTEM
    is predefined Ada **package SYSTEM**, 18–5
    modification, 18–13
    specification, 18–27
  TTY, supplies definition of UNIX tty(4) data structures, 18–5
  TTY_SIZES, supplies definition of UNIX tty(4) data structures, 18–5
  U_ENV, supplies definition of command line argument interface, 18–5
  UNIX, supplies direct interface to UNIX system calls, 18–5
  UNIX_DIRS, supplies interface to UNIX directory utilities, 18–5
  UNIX_PRCS, supplies interface to UNIX process-control utilities, 18–5
  VSTRING, 18–7
PAGE, pragma. *See* Appendix B of Ada RM

panic messages, run time system, 18–50

panic: hardware exception while in RTS, panic message, 18–53

panic: sigstack failed, panic message, 18–54

Parameter Passing, 18–22, 18–23

parameters supported by debugger **p** command, 8–16

parent_VADS_library, 4–9

pascal, **pragma** INTERFACE, 18–25

pass command to target, **a.db**, 20–56

PATH, package, 18–6

pointers and address types, interface programming, 13–6

POOL_ALLOC, package
  memory management support, 18–59
  pool control, 18–62

POOL_CONFIG, package, 18–61

positioning commands, **a.db**, 8–25

postamble code, 18–32

pragmas
  BUILT_IN, reserved for Verdix, 18–10
  CONTROLLED, not implemented, 18–11
  ELABORATE, not implemented. *See* Appendix B of Ada RM
  EXTERNAL_NAME
    link with existing libraries, 13–11
    program conversion, 13–15
    specify *link_name*, 18–11, 18–26
  IMPLICIT_CODE, 12–11
    machine code insertions, 18–32
    specifies allowed (ON) or disallowed (OFF), 18–11
  INLINE, 12–10
    See also Appendix B of Ada RM
    declare external subprograms, 13–8
  INLINE_ONLY, 18–11
  INTERFACE
    supports calls to other languages, 18–11, 18–25
    user must include foreign objects in link, 6–2
  INTERFACE_NAME
    declare external subprograms, 13–6, 13–10
    reference other language variables, 18–11
    reference other language variables and subprograms, 18–25
  INTERFACE_OBJECT, reference other language variables, 18–12
  LIST. *See* Appendix B of Ada RM
  machine code procedures, 12–10

pragmas (*continued*)
  NO_IMAGE, suppress generation of image array, 18–12
  NOT_ELABORATED, avoiding elaboration, 13–18
  OPTIMIZE, not implemented, 18–12
  OPTIMIZE_CODE, 12–11
  PACK
    minimize gaps between components, 18–12
    representation specifications, 18–16
  PAGE. *See* Appendix B of Ada RM
  PRIORITY. *See* Appendix B of Ada RM
  SHARE_CODE, shares code between instantiations, 18–12, 18–15
  SHARED, not implemented, 18–12
  STORAGE_UNIT
    modify **package** SYSTEM, 18–13
    not implemented, 18–13
  SUPPRESS, 12–10
    avoiding elaboration, 13–18
    single parameter form, 18–13
  SYSTEM_NAME, not implemented. *See* **system.a**

preamble code, 18–32

predefined packages and generics, list of those provided in **standard**, 18–26

prelinker, **a.ld**, 6–1, 19–16

previously attached signal, panic message, 18–52

printf, replaced with **ada_put**, 13–15

printing options file, **.prrc**, 9–3

PRIORITY, pragma. *See* Appendix B of Ada RM

procedure
  AA_GLOBAL_FREE
    memory management, 18–57
    pool allocation, 18–60
  AA_INIT, memory management, 18–58
  AA_LOCAL_FREE, memory management, 18–58
  DEALLOCATE_POOL, memory allocation pool control, 18–63
  GET_MEMORY
    calls **malloc**(3), 18–66
    pool allocation, 18–62
  GET_TIME_SLICING, 18–75
  IFACE_SEMA.ENTER, 18–70
  IFACE_SEMA.LEAVE, 18–70
  SET_PRIORITY, 18–74
  SET_TIME_SLICE, 18–75
  SET_TIME_SLICING, 18–75
  SET_USER_FIELD, 18–76
  SWITCH_POOL, memory allocation pool control, 18–63

process compiler error messages, **a.error**, 5–3, 19–10

process stack size, 18–24

process startup, **a.db**, 8–31

program conversion, interface programming, 13–15

publiclib, directory, public domain Ada packages, 18–2, 18–7

Q

quit, terminate debugger session, 8–4, 20–57, 20–75
  a.db execution command, 8–21

R

r, run, **a.db** execution command, 8–20, 20–58
  sample debugger session, 15–5

ranges, types, ranges, and attributes listed, 18–28

RAW_DUMP, package, 18–5

read, read debugger commands from a file, **a.db** execution command, 8–20, 20–59

recompilations
  minimal number performed by **a.make**, 6–3
  source files in dependency order, **a.make**, 6–3, 19–20

record representation clauses, align record fields, 18–16

record types, interface programming, 13–3

reg, list register contents, **a.db** display command, 8–16, 20–60

register definitions, machine code insertions, 18–31

registers, parameter passing, 18–23

reinitialize library directory, **a.cleanlib**, 4–5, 19–5

release directory structure, illustration, 18–2

remote libraries, 4–11

remove
  Ada source file or unit from a library, **a.rm**, 4–12, 19–32
  compilation Ada library, **a.rmlib**, 4–13, 19–33
  information associated with unit or source file, **a.rm**, 4–12

remove (*continued*)
  library components, 4–13

rendezvous
  accept and **select** statements, 18–47
  task entry calls, 18–46

replace existing tags file, 4–14

report vads deficiencies, **a.report**, 10–1, 19–29

representation attributes, 18–21

representation specifications, 18–16

restrictions on 'main' programs, 18–14

RESUME, semaphore services, 18–71

RTS
  configuration, 17–7
  considerations in interface programming, 13–19
  overview from user perspective, 18–37
  panic numbers and messages, 18–50

run queue, display position of tasks with **lt**, 8–13

run time system
  configuration, 17–7
  considerations in interface programming, 13–19
  overview from user perspective, 18–37
  panic numbers and messages, 18–50

S

s, step one source statement, **a.db** execution command, 8–19, 20–62

sample project, 14–1

screen-mode
  a.db, 8–1
  debugger interface, 20–63
    sample debugger session, 15–1
  invocation, **a.db -v**, 8–3
  window commands, **a.db**, 8–29
  windows, **a.db**, 8–27

search, in debugger, 20–68

search list
  displayed or changed by **a.path**, 4–11
  library example, 4–9

select, select new current task, **a.db** positioning command, 8–26, 20–69

select statements, 18–47

semaphore services, 18–68

semaphores, 18–40

separate, changing separates with **a.rm**, 4–13

separate compilation information, 4-1

SEQUENTIAL_IO, package, 18-28

set
  a.pr, commands for **.prrc** printing options file, 9-3
  modify debugger configuration, 8-9, 20-70

set **tabs**, sample debugger session, 15-4

set/reset debugger knowledge of target, **sync**, 8-9, 20-76

SET_PRIORITY, procedure, 18-74

SET_TIME_SLICE, procedure, 18-75

SET_TIME_SLICING, procedure, 18-75

SET_USER_FIELD, procedure, 18-76

SHARE_CODE, pragma, shares code between instantiations, 18-12, 18-15

SHARED, pragma, not implemented, 18-12

shared object-code, 18-15

SHARED_IO, package, 18-5
  instantiates Ada generic I/O packages, 18-16

SHORT_FLOAT, attributes, 18-29

SHORT_INTEGER, attributes, 18-29

simple name, disambiguate overloaded names, 8-8

simple types, interface programming, 13-2

small object lists, space allocated from heap, 18-60

source code formatter, **a.pr**, 9-1, 19-26

source files
  Ada source file structure, 4-2
  must be compiled within Ada directory, 4-2
  remove associated information with **a.rm**, 4-12

source window
  a.db, screen-mode windows, 8-27
  a.db, screen-mode windows,, 20-63

specification of **package** SYSTEM, 18-27
  contained in the file **system.a**, 18-21

specify new position, **a.db** breakpoint command, 8-25, 20-46

stack
  call stack, **cs**, **a.db** positioning command, 8-26, 20-21
  compiler calling conventions, 18-23

standard, directory, standard and other predefined packages, 18-2, 18-3

static priority, display priority of tasks with **lt** *task*, 8-16

STATUS, package, supplies definition of UNIX STATUS_BUFFER data type, 18-5

STORAGE_SIZE, default for a task, 18-22

STORAGE_UNIT, pragma
  modify **package** SYSTEM, 18-13
  not implemented, 18-13

STORAGE_UNITS
  default collection size, 18-22
  limit, 18-22

STRING, type, 18-29

string types, interface programming, 13-6

STRLEN, package, 18-5

STRNCPY, package, 18-5

subdirectory, create example, 4-9

subprogram
  external, declare in interface programming, 13-6
  location of bodies, 4-2
  location of specifications, 4-2

summarize disk usage for ada compilers, **a.du**, 4-5, 19-9

sup, directory, additional executables, 18-2

SUPPRESS, pragma, 12-10
  avoiding elaboration, 13-18
  single parameter form, 18-13

SUSPEND, semaphore services, 18-71

SWITCH_POOL, procedure, memory allocation pool control, 18-63

symbolic debugger, **a.db**, 8-1

sync, **a.db**, set/reset debugger knowledge of taret, 8-9, 20-76

syntax errors, limits, 18-22

SYSTEM, package
  is predefined Ada **package** SYSTEM, 18-5
  modification, 18-13
  specification, 18-27

system crash, lock files, 5-6

system.a
  contains specification of **package** SYSTEM, 18-21
  modification of **package** SYSTEM, 18-13

SYSTEM.ADDRESS'REF(N), attribute, used in place of unchecked conversion, 18-14

SYSTEM.MAX_REC_SIZE, restrict size of unconstrained records and arrays, 18-28

System-Dependent Named Numbers. *See* **package** SYSTEM

SYSTEM_NAME, pragma, not implemented. *See* **system.a**

T

tags, file created by **a.tags**, 4-13—4-15

TARGET, INFO directive, 18-35

task
  activation, 18-43
  completion and termination, 18-49
  control services, 18-72
  creation, 18-41
  entries associated with UNIX signals, 18-19
  entry calls, 18-46
  startup, 18-44

task, display current task address, **a.db** display command, 20-77

TASKING, LINK directive, 18-36
  searched for by **a.ld**, 6-3

tasking, 18-40
  a.db breakpoint commands, 8-25
  knowledge of terms is assumed, 18-37

terminating the debugger session, **quit**, 8-4, 20-75

time slicing
  conflicts, 18-66
  disable for debugging, 18-75
  time slicing enabled affects benchmarks adversly, 16-2

TINY_INTEGER, attributes, 18-29

tools
  a.cleanlib, reinitialize library directory, 4-5, 19-5
  a.das, disassemble object files, 7-1, 19-6
  a.du, summarize disk usage for Ada libraries, 4-5, 19-9
  a.error, process compiler error messages, 5-3, 19-10
  a.help
    interactive help utility for VADS, 10-1
    interactive help utility for VADS, 19-12
  a.info, list or change VADS library directives, 4-6, 19-14
  a.ld, prelinker, 6-1
  a.ld, prelinker, 19-16
  a.ls, list compiled units, 4-8, 19-19
  a.make, recompile source files in dependency order, 6-3, 19-20

tools (*continued*)
  a.mklib, create a VADS library directory, 4-8, 19-23
  a.path, report or change library search list, 4-11, 19-25
  a.pr, source code formatter, 9-1, 19-26
  a.report, report VADS deficiencies, 10-1, 19-29
  a.rm, remove an Ada source file or unit from a library, 4-12, 19-32
  a.rmlib, remove a compilation Ada library, 4-13, 19-33
  a.tags, create cross reference of units or types, 4-13, 19-34
  a.vadsrc, display available VADS versions and create a default library configuration file, 4-10, 19-36
  a.which, find a compiled unit, 4-15, 19-37
  ada, invoke VADS Ada compiler, 5-1, 6-1, 19-2
  limits, 18-22
  list, 3-1, 19-1
  a.list, list programs containing no errors, 19-18

TS_ACCEPT, **accept** and **select** statements, 18-47

TS_ACTIVATE_LIST, task activation, 18-43

TS_ACTIVATION_COMPLETE, task activation, 18-43

TS_ACTIVATION_EXCEPTIONS, task activation, 18-43

TS_ALLOC, memory allocation in the kernel, 18-67

TS_CALL, task entry call, 18-46

TS_COMPLETE_MASTER, task completion and termination, 18-50

TS_COMPLETE_TASK, task completion and termination, 18-49

TS_CREATE_MASTER, task creation, 18-42

TS_CREATE_TASK_AND_LINK, task creation, 18-42

TS_DELAY, tasking, 18-45

TS_INIT_ACTIVATE_LIST, task creation, 18-42

TS_SELECT_TERMINATE, accept and select statements, 18-48

TTY, package, supplies definition of UNIX **tty**(4) data structures, 18-5

TTY_SIZES, package, supplies definition of UNIX **tty**(4) data structures, 18-5

tutorial, 1-1

types
array, interface programming, 13–5
corresponding between C, Fortran, and Ada, 13–3
cross reference with **a.tags**, 4–13, 19–34
data, interface programming, 13–2
dynamic array, interface programming, 13–5
enumeration, 18–28
pointers and address, interface programming, 13–6
record, interface programming, 13–3
simple, interface programming, 13–2
string, 18–29
string, interface programming, 13–6
types, ranges and attributes listed, 18–28

U

U_ENV, package, supplies definition of command line argument interface, 18–5

unchecked
language name, **pragma** INTERFACE, 18–25
programming, 18–21
storage deallocations, 18–21
type conversions, 18–22

unchecked conversion, SYSTEM.ADDRESS'REF(N), used in place of unchecked conversion, 18–14

UNCHECKED_CONVERSION, generic function, 18–24

UNCHECKED_DEALLOCATION, generic function, 18–24
memory allocation in VADS Run Time, 18–55
pool allocation, 18–60

unexpected exception, panic message, 18–53

units
compile units in separate source files, 13–17
cross reference with **a.tags**, 4–13, 19–34
dependencies, 4–2
externally compiled, 4–11
index of locations in **tags** file, 4–13
information gathered by **a.ld**, 6–1
information on imported units provided by **a.du -i**, 4–6
list created by **a.ls**, 4–8
names of library units, 4–10
remove associated information with **a.rm**, 4–12

UNIX, package, supplies direct interface to UNIX system calls, 18–5

UNIX_DIRS, package, supplies direct interface to UNIX directory utilities, 18–5

UNIX_PRCS, package, supplies interface to UNIX process-control utilities, 18–5

unsigned addresses, to place an object, use SYSTEM.ADDRESS'REF(N) attribute, 18–14

unx_heap_cfg.a, contains **package** HEAP_CONFIG, 18–59

user library, one of two major parts of RTS, 18–37

user_config, directory, RTS configuration files, 18–2

V

VADS, INFO directive, 18–35

VADS library directories, 4–3

VADS_location, represents directory of current VADS version, 17–2

variables and visibiity rules, **a.db**, 8–8

verdixlib, directory, Verdix supported packages, 18–3, 18–6

VERSIONS, INFO directive, 18–35

vi, enter screen mode in debugger, 8–8, 20–79

visibility rules, determine which identifiers are visible at a breakpoint in debugger, 20–80

VSTRING, package, 18–7

W

w, display window of source statements, **a.db** display command, 8–10, 20–81

wi, display window of disassembled instructions, **a.db** display command, 8–11, 8–24, 20–82
See also instruction and source modes

window
command syntax, **a.db**, 8–29
control commands, **a.db**, 20–65
control commands (list), **a.db**, 20–66

WITH*n* directive
character limit, 18–22
example, 4–6
LINK directive, 18–36
linking of foreigh objects with **a.ld**, 6–2
searched for by **a.ld**, 6–3